Skilled in the Art of Being Idle: Reducing Energy Waste in Networked Systems

Sergiu Nedevschi[*] [†] Jaideep Chandrashekar[†] Junda Liu[‡] [*] Bruce Nordman[§]
Sylvia Ratnasamy[†] Nina Taft[†]

Abstract

Networked end-systems such as desktops and set-top boxes are often left powered-on, but idle, leading to wasted energy consumption. An alternative would be for these idle systems to enter low-power sleep modes. Unfortunately, today, a sleeping system sees degraded functionality: first, a sleeping device loses its network "presence" which is problematic to users and applications that expect to maintain access to a remote machine and, second, sleeping can prevent running tasks scheduled during times of low utilization (*e.g.,* network backups). Various solutions to these problems have been proposed over the years including wake-on-lan (WoL) mechanisms that wake hosts when specific packets arrive, and the use of a proxy that handles idle-time traffic on behalf of a sleeping host. As of yet, however, an in-depth evaluation of the potential for energy savings, and the effectiveness of proposed solutions has not been carried out. To remedy this, in this paper, we collect data directly from 250 enterprise users on their end-host machines capturing network traffic patterns and user presence indicators. With this data, we answer several questions: what is the potential value of proxying or using magic packets? which protocols and applications require proxying? how comprehensive does proxying need to be for energy benefits to be compelling? and so on.

We find that, although there is indeed much potential for energy savings, trivial approaches are not effective. We also find that achieving substantial savings requires a careful consideration of the tradeoffs between the proxy complexity and the idle-time functionality available to users, and that these tradeoffs vary with user environment. Based on our findings, we propose and evaluate a proxy architecture that exposes a minimal set of APIs to support different forms of idle-time behavior.

[*]International Computer Science Institute
[†]Intel Research
[‡]University of California, Berkeley
[§]Lawrence Berkeley National Laboratories

1 Introduction

Recent years have seen rising concern over the energy consumption of our computing infrastructure. A recent study [19] estimates that, in the U.S. alone, energy consumption for networked systems approaches 150 TWh, with an associated cost of around 15 billion dollars. About 75% of this consumption can be attributed to homes and enterprises, and the remaining 25% to networks and data centers. Our focus in this paper is on reducing the 75% consumed in homes and enterprises. To put this in perspective, this energy (112 TWh) is roughly equivalent to the yearly output of 6 nuclear plants [14]. Of equal concern is that this consumption has grown – and continues to grow – at a rapid pace.

In response to these energy concerns, computer vendors have developed sophisticated power management techniques that offer various options by which to reduce computer power consumption. Broadly, these techniques all build on hardware support for *sleep* (S-states), and frequency/voltage scaling [21] (processor P-states [4]). The former is intended to reduce power consumption during idle times, by powering down sub-components to different extents, while the latter reduces power consumption while active, by lowering processor operating frequency and voltage during active periods of low system utilization.

Of these, sleep modes offer the greatest reduction in the power draw of machines that are *idle*. For example, a typical sleeping desktop draws no more than 5W [2], as compared to at least 50W [2] when on, but idle – an order of magnitude reduction. It is thus unfortunate that sleep modes are not taken advantage of to anywhere close to their fullest potential. Surveys of office buildings have shown that about two thirds of desktops are fully on at night [20], with only 4% asleep. Our own measurements (Section 3) reveal that enterprise desktops remain idle for an average of 12 hours/day – time that could, in theory, be spent mostly sleeping.

Relative to an idle machine, the only loss of functionality to a sleeping machine is twofold. First, since a sleeping computer cannot receive or transmit network messages, it effectively loses its "presence" on the network.

This can lead to broken connections and sessions when the machine resumes (*e.g.,* a sleeping machine does not renew its DHCP lease and hence loses its IP address) and also prevents remote access to a sleeping computer. This loss of functionality is problematic in an increasingly networked world. For example, a user at home might want to access files on his desktop at work, an on-the-road user might want to download files from his home machine to his handheld, system administrators might desire access to enterprise machines for software updates, security checks and so forth. In fact, some enterprises, *require* that users not power off their desktops to ensure administrators can access machines at all times [6]. The second problematic scenario is when users or administrators deliberately want to schedule tasks to run during idle times – *e.g.,* network backups that run at night, critical software updates, and so on. Unfortunately, these drawbacks cause users to forego the use of sleep modes leading to wasteful energy consumption.

The above observations are not new, having been repeatedly articulated (also by some of the authors) in both the technical literature and popular press [13, 16, 19, 10, 7, 15]. Likewise, there have been two long-standing proposals on how to tackle the problem. The first is to generalize the old technology of Wake-on-LAN (WoL), an Ethernet computer networking standard that allows a machine to be turned on or woken up remotely by a special "magic packet". A second, more heavyweight, proposal has been to use a *proxy* that handles idle-time traffic on behalf of a sleeping host(s), waking the sleeping host when appropriate. Thus both problem (wasted energy consumption by idle computers) and proposed solutions (wake-up packets and/or proxies for sleeping machines) have existed for a while now. In fact, the technology for WoL has been implemented and deployed although not widely used (we explore possible causes for this later in the paper). However the recent focus on energy consumption has led to renewed interest in the topic with calls for research [7, 13], calls for standardization [12], and even some commercial prototypes [15]. As yet however, there has been little systematic and in-depth evaluation of the problem or its solutions – what savings might such solutions enable? what is the broader design space for solutions? what, if any, might be the role of standardization? are these the right long-term solutions? *etc.*

In this paper, we explore these questions by studying user behavior and network traffic in an enterprise environment. Specifically, we focus on answering the following questions:

Q1: Is the problem worth solving? Just how much energy is squandered due to poor computer sleeping habits? This will tell us the potential energy savings these solutions stand to enable and hence the complexity they warrant. Also, is proxying really needed to realize these potential savings or can we hope that WoL suffices to maintain network presence while still sleeping usefully?

Q2: What network traffic do idle machines see? Understanding this will shed light on how this idle-time traffic might be dealt with and, consequently, what protocols and applications might trigger wake-up packets and/or require proxying. On the face of it, it would seem like an idle machine ought not to be engaged in much useful activity and hence, ideally, one might hope that a small number of wake-up events are required and/or that a relatively small set of protocols must be proxied to realize useful savings.

Q3: What is the design space for a proxy? In general, the space appears large. Different proxy implementations might vary in the complexity they undertake in terms of what work is handled by the proxy *vs.* waking the machine to do so. In some cases, one might opt for a relatively simple proxy that (for example) only responds to certain protocols such as ARP (specified by the DMTF ASF2.0 standard[1]) and NetBios. But more complex proxies are also conceivable. For example, a proxy might take on application-specific processing such as initiating/completing BitTorrent downloads during idle times and so forth. Likewise, there are many conceivable deployment options – a proxy might run at a network middlebox (*e.g.,* firewall, NAT, *etc.*), at a separate machine on each subnet, or even at individual machines (*e.g.,* on its NIC, on a motherboard controller, or on a USB-attached lightweight microengine). Given this breadth of options, we are interested in whether one can identify a minimal proxy architecture that exposes a set of open APIs that would accommodate a spectrum of design choices and deployment models. Doing so appears important because a proxy potentially interacts with a diversity of system components and even vendors (hardware power management, operating systems, higher-layer applications, network switches, NICs, *etc.*) and hence identifying a core set of open APIs would allow different vendors to co-exist and yet innovate independently. For example, an application developer should be able to define the manner in which his application interacts with the proxy with no concern for whether the proxy is deployed at a firewall, a separate machine or a NIC.

Q4: What implications does proxying have for future protocol and system design? The need for a proxy arises largely because network protocols and applications were never designed with energy efficiency in mind nor to usefully exploit, or even co-exist with, power management in modern PCs and operating systems. While proxies offer a pragmatic approach to dealing with this mismatch for currently deployed protocols and software, one might also take a longer-term view of the problem and ask how we might redesign protocols, applications

or even hardware power management to eventually obviate the need for such proxying altogether.

In this paper, we study the network-related behavior of 250 users and machines in enterprise and home environments, and evaluate each of the above questions in Sections 3 to 6 respectively.

2 Measurement data and methodology

We collected network and user-level activity traces from approximately 250 client machines belonging to Intel corporation employees, for a period of approximately 5 weeks. The machines, running Windows XP, include both desktops and notebooks—approximately 10% are desktops and the rest, notebooks.

Our trace collection software was run at the individual end-hosts themselves and hence, in the case of notebooks, trace collection continued uninterrupted as the user moved between *enterprise* and *home*, enabling us to analyze traffic from both of these environments.

Our packet traces were collected using Windump. To capture user activity, we developed an application that sampled a number of user activity indicators at one second intervals. The user activity indicators we collected included keyboard activity and mouse movements and clicks. Noticeable gaps in the traces occur when the host was turned off, put to sleep, or in hibernation. Thus each end-host is associated with a trace of its network and user activity. We then used BRO [9] to reassemble connection-level information from each packet-level trace.

Thus, for the 5 week duration of our measurement study, we have the following information for each end-host:

- a packet-level (pcap) trace capturing packet headers for the entire duration
- per-second indicators of user presence at the machine
- the set of all connections—incoming and outgoing—as reconstructed by BRO from the packet traces

The result is a 500GB repository of trace data. To process this, we developed a custom tool that extends the publicly available WIRESHARK [3] network protocol analyzer with different function callbacks implementing the additional processing required for our study.

3 Low Power Proxying: Potential and Need

In this section, we estimate the energy wasted by home and office computers that remain powered on even when idle, i.e., even when there is no human interacting with the computer. Subsequently, we investigate whether very simple approaches — *e.g.,* the computer is woken up to process every network packet and then returns to sleep immediately after—would suffice in allowing hosts to sleep more while preserving their network "presence".

How much energy is squandered by not sleeping?

Virtually all modern computers support advanced sleep states, S1 - S4 as defined in the ACPI specification [5].

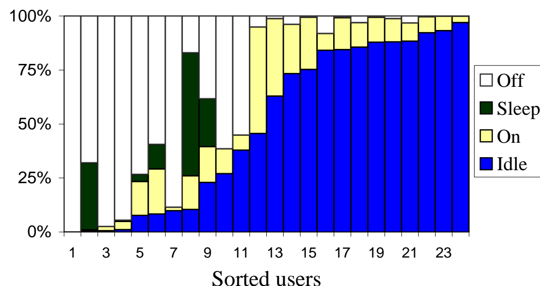

Figure 1: Distribution of the split among off, idle and active periods across users.

These states vary in their characteristics—whether the CPU is powered off, how much memory state is lost, which buses are clocked and so on. However, common to all states, is that the CPU stops executing instructions and hence the computer appears to be powered down. Thus although these sleep states conserve energy, the undesirable side-effect is that a sleeping computer effectively "falls off" the network—making it unavailable for remote access and unable to perform routine tasks that may have been scheduled at particular times. This leads many users to disable power management altogether and instead leave machines running 24/7. For example, studies have shown that approximately 60% of the PCs in office buildings remain powered on overnight and almost all of these have power management disabled [20].

To more carefully quantify the amount of wasted energy (and hence potential savings), we analyzed the trace data collected at our enterprise machines. To determine whether a machine has a locally present and active user, we examine the recorded mouse and keyboard activity for the machine: if no such activity is recorded for 15 minutes, we say that the machine is *idle*. We use 15 minutes because it is the default timeout recommended by EnergyStar for putting machines to sleep, and because it represents a simple (and fairly liberal) approximation for the notion of *idle-ness*, for which a standard definition does not exist. We maintain this definition of idle-ness for the remainder of the paper.

At any point in time, we classify a machine as being in one of four possible states: (a) on, and actively used, we call this *active*; (b) on, but not used, *idle*; (c) in a *sleep* state such as S3 or S4, and (d) powered down, *off*. Note that this notion of "idle" refers here to the *user*, and not the machine, being inactive.

In Figure 1 we present this data for our enterprise desktops. We focus here on the desktops since this represents the potential energy savings an enterprise could garner. Because the bulk of our traces come from mobile users, we have a limited number of desktops. We see that the fraction of time when these machines are active is quite low, falling below 10% on average. Moreover, the average fraction of time when machines are idle is high –

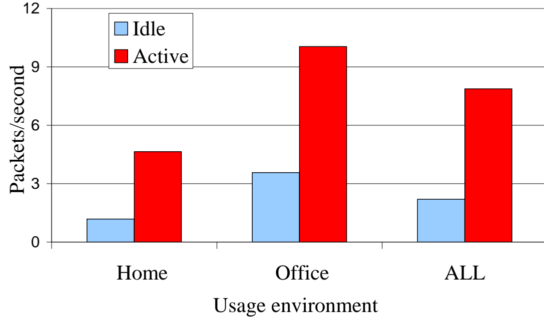

Figure 2: Average number of directed and broadcast/multicast packets received on average by a network host at home and in the office.

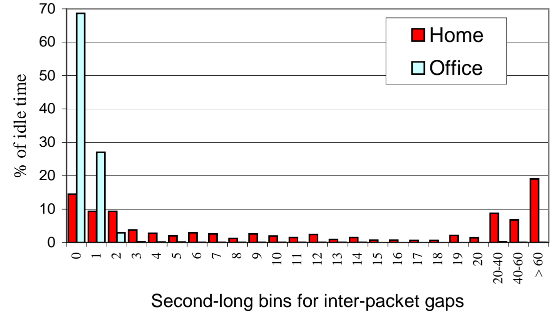

Figure 3: Histogram of the fraction of the idle time made up of inter-packet gaps of different size.

about 50%. Similar to other studies, we note that a small fraction of our desktops (only 5 out 24) use sleep mode at all. Overall, this indicates that there is a tremendous opportunity for energy savings on enterprise desktops. The opportunity on our corporate laptops exists too, but is moderate because we found that our laptop users were more likely to employ aggressive sleeping configurations that come pre-configured on laptops.

While the sample of the desktop machines in our experiments is small, the results are consistent with existing studies [20]. We therefore use these measured idle times to extrapolate the energy that could be saved by sleeping instead of remaining idle. There are estimated to be about 170 million desktop PCs in the US (data summarized in [23]). Assuming an 80W power consumption of an idle PC, and assuming these machines are idle for 50% of the time, this amounts to roughly 60 TWh/year of wasted electricity (or 6 billion dollars, at US$0.10 per kWh).

Is low-power proxying needed? Before developing new solutions to reducing host idle times, we investigate whether very simple approaches like waking up for every packet can deliver these savings while maintaining full network presence. In this approach, which we denote (WoP – wake on packet), the machine is woken up for every packet it needs to receive (directed or broadcast), and put back to sleep after the packet is served. The performance of such an approach depends on whether the inter-packet gap (IPG) is smaller or comparable to the time it takes to transition in and out of sleep. If it isn't then there is no gain over simply leaving the machine in an idle state.

To examine the traffic during idle times, we used both our desktop and laptop machines. We consider both types (even though we're primarily interested in desktops) because this gives us a significantly larger set of samples. We separate the idle time traffic into two categories, office and home. In Figure 2 we plot the average number of packets/sec for idle traffic both in the office and at home. In the office environment, the average number of packets per second is roughly 3, while at home it is roughly 1. This indicates a fairly constant level of background chatter on the network, independent of the user's activity. Because this number is an average, we need to understand if these packets occur in bursts or not. If the packets are bursty most of the time, then there may still be opportunities to sleep as the host can be woken up to service a burst of packets and then be put to sleep for some reasonable period of time (certainly more than a few seconds). If these packets occur fairly evenly spaced, then it is not worth going to sleep unless the time to transition in and out of sleep is very small (on the order of 1 to 3 seconds).

To quantify the burstiness level of our traffic, we group inter-packet gaps into second-long bins (*i.e.,* 0-1s, 1-2s, *etc.*). We then compute the sum of the inter-packet gaps in each of these bins, and finally compute the fraction of total idle time represented by each bin. We present these results in Figure 3, for both home and office environments. In the office, over 90% of the time, the IPG is less than 2 seconds. Although the distribution is more uniformly spread for the home environment, we still see that roughly 70% of the time, the IPG is less than 20 seconds. Overall we observe that: (a) neither of the environments enjoys many long periods of quiet time; (b) we find this distribution to be very different for the two environments. In home networks the distribution has a much heavier tail, the traffic is burstier, and we do see longer periods of quiet time.

We now translate these observations into actual sleep time. In order to perform this computation, we must consider a representative value for the time interval it takes the host to wake up, process the packet and then go to sleep again—we call this the transition time, denoted t_s. Today, typical machines take 3 – 8 seconds to enter S3 sleep, and 3 – 5 seconds to fully resume from S3, as measured in a recent study [6]. Therefore, it is reasonable to assume an average transition time t_s of $10s$.

When a packet arrives, the machine is woken up to serve the packet. After processing a packet, the machine only goes to sleep again if it knows the next packet will not arrive before it transitions to sleep. This idealized test thus assumes that the host knows the future incoming

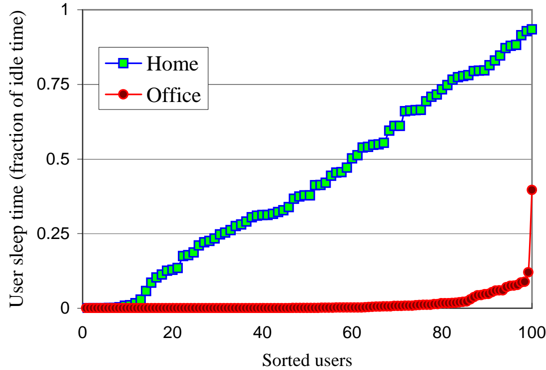

Figure 4: The fraction of idle time users can sleep if they wake up for every packet, across different environments for a transition time $t_s = 10$seconds.

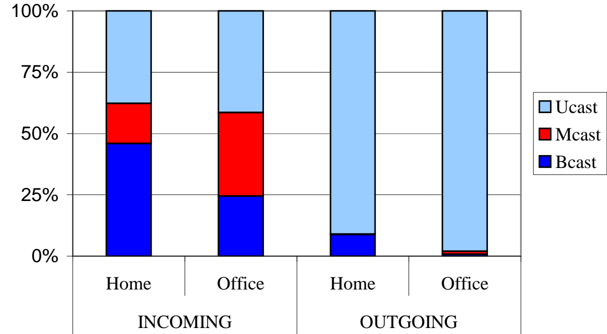

Figure 5: Composition of incoming and outgoing traffic during idle times, for home and office environments, based on communication paradigms

packet stream and captures the best the machine could do in terms of energy savings.

Figure 4 presents the fraction of idle time for which users can sleep, assuming the policy described above. The results are rather dramatically different for across environments. In the office, there is almost no opportunity to sleep for the majority of the users. This indicates that the magic packet-like approach will not succeed in saving any energy for machines in a typical corporate office environment. For the home environment, we see that roughly half the users can sleep for over 50% of their idle times. Thus in these environments, a $10s$ transition time coupled with a WoP type policy can be somewhat effective. However, these estimates assume perfect knowledge of future traffic arrivals and also frequent transitions in and out of sleep—in practice, we expect the achievable savings would be somewhat lower. Nonetheless, this does suggest that efforts to reduce system transition times in future hardware could obviate the need for more complex power-saving strategies in certain environments.

We conclude that while significant opportunity for sleep exists, capitalizing on this opportunity requires solutions that go beyond merely waking the host to handle network traffic; we thus consider solutions based on proxying idle-time traffic in the following sections.

4 Deconstructing traffic

In the previous section we saw that, by just waking up to handle all packets, our ability to increase a machine's sleep time is limited. In particular, we see virtually no energy savings in the dominant office environments. This suggests that we need an approach that is more discriminating in choosing when to wake hosts. This leads us to an alternate solution to the WoL which is to employ a network proxy whose job is to handle idle-time traffic on behalf of one or more sleeping hosts. Packets destined for a sleeping host are intercepted by (or routed to, depending on the proxy deployment model) its proxy. At this point, the proxy must know what to do with this intercepted traffic; broadly, the proxy must choose between three reactions: a) ignore/drop the packet; b) respond to the packet on behalf of the machine; or c) wake up the machine to service it. To make a judicious choice, the proxy must have some knowledge of network traffic—what traffic is safely ignorable, what applications do packets belong to, which applications are essential, and so forth. In this section, we do a top-down deconstruction of the idle-time traffic traces aimed at learning the answers to these questions.

4.1 Traffic Classes by Communication Paradigm

To begin, we look at all packets exchanged during idle periods, and classify each packet as either being a broadcast, multicast or unicast packet. Within these broad traffic classes, we further partition the traffic by whether the packets are incoming or outgoing, for both the home and office environments. We separate incoming and outgoing traffic because we expect them to look different in terms of the proportion of each class in different directions (*e.g.,* most end-hosts ought to send little broadcast traffic). Similarly, we look at different usage environments because it is intuitive that the dominant protocols and applications used in each environment may differ. Since we expect these differences, we treat them as such to avoid mischaracterizations. The breakdown of our traffic according to all these partitions in depicted in Fig. 5.

We note that outgoing traffic is dominated by unicast traffic since, as expected, each host generates little broadcast or multicast traffic. We also find that incoming traffic at a host sees significant proportions of *all* three classes of traffic, and this is true in both enterprise and home environments. This suggests that a power-saving proxy might have to tackle all three traffic classes to see significant savings.

So far, we looked at traffic volumes as indicative of the need to proxy the corresponding traffic type. We now directly evaluate the opportunity for sleep represented by each traffic type. To understand the maximum sleeping

opportunities, we consider for a moment an idealized scenario in which we use our proxy to ignore *all* incoming packets from either or both of the broadcast and multicast traffic classes. A machine always wakes up for unicast packets. Fig. 6 shows the sleep potential in four scenarios: a) ignore only broadcast and wake for the rest; c) ignore only multicast and wake for the rest; c) ignore both broadcast and multicast. For comparison purposes we also include the results for a scenario d) in which we wake up for all packets. This comparison allows us to compare the benefits derived from these four different proxy policies. For each user, we computed the fraction of its idle time that could have been spent sleeping under the scenario in question. We use a transition time of $t_s = 10s$ and the results are averaged over 250 users for both home and office environments.

We make the following observations:

(i) Broadcast and multicast are largely responsible for poor sleep. If we can proxy these, then we can recuperate over 80% of the idle time in home environments. And in the office, where previously sleep was barely possible, we can now sleep for over 50% of the idle time.

(ii) Doing away with only one of either broadcast or multicast is not very effective (we suspect this is due to the periodicity of multicast and broadcast protocols, and evaluate this in later sections).

More generally, the graph clearly indicates a valuable conclusion—if we're looking to narrow the set of traffic classes to proxy, then multicast and broadcast traffic appear to be clear low-hanging fruit and should be our primary candidates for proxying. That said, proxying unicast traffic appears key to achieving higher savings (beyond 50%) in the enterprise and hence should not be dismissed either. We thus continue, for now, to study all three traffic types.

Of course, whether these potential savings can actually be realized depends on whether a particular traffic type can indeed be handled by a proxy without waking the host. This depends on the specific protocols and applications within that class and hence, in the remainder of this section, we proceed in turn to deconstruct each of broadcast (§4.2), multicast (§4.3) and unicast (§4.4) traffic.

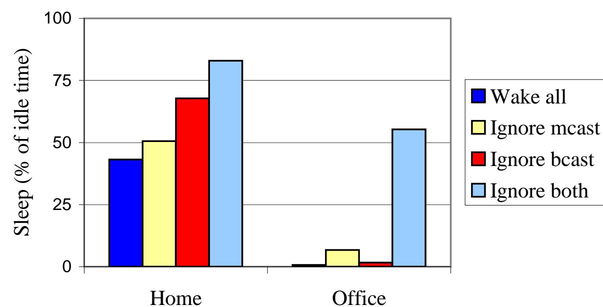

Figure 6: Average sleep opportunity when ignoring multicast and/or broadcast traffic, for different environments

4.2 Deconstructing Broadcast

Our goal in this section is to evaluate individual broadcast protocols, looking for: (1) which of these protocols are the main offenders in terms of preventing hosts from sleeping and, (2) what purpose do these protocols serve and how might a proxy handle them. Answering the first question requires a measure of protocol "badness" with respect to preventing hosts from sleeping. We use two metrics for our evaluation. The first is simply the total **volume of traffic** due to the protocol in question. While high-volume traffic often makes sleep harder, this is an imperfect metric since the (in)ability to sleep depends as much on the precise temporal packet arrival pattern due to the protocol as on packet volumes. Nonetheless, we retain traffic-volume as an intuitive, although indirect measure of protocol badness. Our second metric—which we term the **half-sleep time**, denoted `ts_50` – more directly measures a protocol's role in preventing sleep.

We define the half-sleep time for a protocol (or traffic type) P as the largest host transition time that would be required to allow the host to sleep for at least 50% of its idle time, under the scenario where the machine wakes up for all packets of type P and ignores all other traffic. In effect, `ts_50` quantifies the intuition that, if we ignore all traffic other than that due to the protocol of interest, then a protocol whose packets arrive spaced far enough apart in time is more conducive to sleep since the host has sufficient time to transition in and out of sleep.

In more detail, `ts_50` is computed from our traces as follows. We measure the total time a given host can sleep assuming it wakes up for all the packets of the protocol under consideration and ignores all others. We compute this number for all hosts and take the average. This gives us an upper bound on achievable sleep if the protocol is handled by waking the host. We estimate this sleep duration for different values of the host transition time t_s ranging from 0 seconds (ideal) to 15 minutes. The largest of these transition times t_s that allows the host to sleep for over 50% of its idle time is the protocol's `ts_50` .

Intuitively, `ts_50` indicates the extent to which a protocol is "sleep friendly" since protocols with large values of `ts_50` could simply be handled by allowing the machine to wake up; whereas those with low values of `ts_50` imply that (to achieve useful sleep) the proxy must handle such traffic without waking the host.

For our evaluation, we classify each packet by protocol and rank them by both metrics: traffic volume and the half-sleep time. We begin by measuring traffic volume, we then establish the top ranking protocols by volume, and use these as candidates for our second metric, the half-sleep time. When presenting the top ranking protocols by each of the metrics, we consider : (1) the protocols whose traffic volumes represents more than 1% of the total traffic at the host and (2) the protocols with a

Office		*Home*	
Protocol	**% of traffic**	**Protocol**	**% of traffic**
ARP	46.13	ARP	42.56
NBNS	22.89	SSDP	19.63
IPX	10.12	NBNS	9.48
NBDGM	5.91	CUPS	5.6
LLC	3.28	LLC	4.4
ANS	2.85	UNISTIM	4.07
RPC	2.46	IPX	3.8
BOOTP	2.01	NBDGM	2.3
NTP	1.13	BOOTP	1.02
Other	3.22	Other	7.14
Total	**100**	**Total**	**100**

Figure 7: Protocol composition of incoming broadcast traffic, in both office and home environments, ranked by per-protocol traffic volumes.

Office		*Home*	
All Bcast	**1-2s**	**All Bcast**	**10-20s**
ARP	2-3s	ARP	1-2min
NBDGM	10-20s	NBDGM	2-4min
NBNS	2-4min	NBNS	4-8min
IPX	4-8min		

Figure 8: Protocol composition for broadcast protocols ranked by `ts_50` .

half-sleep time of less than 15 minutes. Table 7 and 8 present our results for broadcast traffic. For completeness, we also present the value of `ts_50` when considering all broadcast traffic together.

In terms of traffic volumes, we see that the bulk of broadcast traffic is in the cause of address resolution and various service discovery protocols (*e.g.,* ARP, Netbios Name Service – NBNS, the Simple Service Discovery Protocol used by UPnP devices – SSDP). These protocols are well represented in both home and office LANs. A second well-represented category of traffic is from router-specific protocols (*e.g.,* routing protocols implemented on top of the IPX).

In terms of the half-sleep time, we see that broadcast as a whole allows very little sleep in the office: achieving 50% sleep would require very fast transitions (between 1 and 2 seconds), not feasible with today's hardware support. The situation in home LANs is significantly better (`ts_50` $= 10s$). In terms of protocols, we see that the greatest offenders are similar to those from our traffic-volume analysis, namely: ARP, Netbios Datagrams (NBDGM) and Name Queries (NBNS), and IPX.

On closer examination, we find that most of these offending protocols could be easily handled by a proxy: for example, IPX is safely ignorable, ARP traffic that is not destined to the machine in question is likewise safely ignorable; for ARP queries destined to the machine, it would be fairly straightforward for a proxy to automatically construct and generate the requisite response without having to wake the host.

Office		*Home*	
Protocol	**% of traffic**	**Protocol**	**% of traffic**
HSRP	59.58	SSDP	94.4
SSDP	24.91	HSRP	2.31
PIM	6.04	IGMP	1.84
IGMP	5.05		
EIGRP	1.88		
Other	2.54	Other	1.45
Total	**100**	**Total**	**100**

Figure 9: Protocol composition for incoming multicast traffic, in both office and home enviroments, ranked by per-protocol traffic volumes.

Office		*Home*	
All Mcast	**0-1s**	**All Mcast**	**1-2min**
HSRP	0-1s	SSDP	4-8min
PIM	8-9s	HSRP	>15min
IGMP	20-30s	IGMP	>15min
SSDP	20-30s		

Figure 10: Protocol composition for incoming multicast traffic, in both office and home environments, ranked by `ts_50` .

4.3 Deconstructing Multicast

Table 9 and 10 present our protocol rankings for multicast traffic. Again, we also present the value of `ts_50` when considering all multicast traffic taken together. We see that, multicast traffic (as a whole) can be a bad offender in enterprise environments with an `ts_50` $= 0-1s$. It turns out that this is largely caused by router traffic—the Hot Standby Router Protocol (HSRP), Protocol Independent Multicast (PIM), EIGRP, *etc.*

This traffic is either absent (*e.g.,* PIM) or greatly reduced (*e.g.,* HSRP) in home environments which explains why multicast is much less problematic in homes, with an `ts_50` $= 1 - 2$ minutes (compared to $10 - 20s$ for broadcast).

The good news is that all router traffic (HSRP, PIM, IGRP) is safely ignorable. In fact, many modern Ethernet cards already include a hardware multicast filter that discards most unwanted multicast traffic.

As with broadcast traffic, we also see significant traffic contributed by service discovery protocols: in this case SSDP, the Simple Service Discovery Protocol used by UPnP devices. Once again, for protocols such as SSDP and IGMP, it is fairly straightforward for a proxy to automatically respond to incoming traffic without waking the host; doing so would require some amount of state at the proxy such as the list of multicast groups the interface belongs to and the services running on the machine.

4.4 Deconstructing Unicast

Finally, we present our protocol ranking for unicast traffic in Tables 11 and 12. Because much of unicast traffic is either TCP or UDP, and this level of classification is unlikely to be informative, we further break each

Transport Protocol	Session Protocol	% of traffic	
TCP		94.73	
	DCE/RPC		24.91
	NBSS		14.85
	HTTP		12.31
	TPKT		3.82
	SSL		2.68
	VNC		2.45
	Other		33.71
UDP		3.75	
	DNS		1
	Other		2.75
ICMP		1.29	1.29
Other		0.23	0.23
Total		**100**	**100**

Figure 11: Protocol composition of incoming unicast traffic in office enviroments, ranked by per-protocol traffic volumes.

Office		*Home*	
All Ucast	**10-20s**	**All Ucast**	**50-60s**
TCP	10-20s	UDP	1-2min
UDP	1-2min	DNS	1-2min
DCE/RPC	1-2min	TCP	8-15min
DNS	2-4min		
SMB	4-8min		
NBNS	4-8min		
HTTP	8-15min		

Figure 12: Protocol composition of incoming unicast traffic in office environments, ranked by `ts_50` .

Port	App	ts_50
TCP keep alives	many	1-2min
UDP 53	DNS	2-4min
TCP 1025	DCE/RPC	2-4min
TCP 445	SMB/CIFS	4-8min
TCP 63422	Bigfix	4-8min
TCP 53	DNS	4-8min
TCP 80	HTTP	8-15min
UDP 63422	Bigfix	8-15min
TCP SYNs	many	> 15min

Figure 13: Protocol composition for unicast traffic based on TCP and UDP ports, ranked by `ts_50`

down by session-layer protocol with an additional mapping from ports in Table 13. Unfortunately, unlike the case of broadcast and multicast, with unicast, it is harder to deduce the ultimate purpose for much of this traffic since even the session or application-level protocol identifiers are fairly generic. (One exception is the "BigFix" application listed in Fig. 13. BigFix is an enterprise software patching service that checks security compliance of enterprise machines; based on the frequency and volume of BigFix traffic we see, it appears to have been configured by an over-zealous system administrator.)

Stymied in our attempts to deconstruct unicast traffic based on whether and how it might be proxied, we try

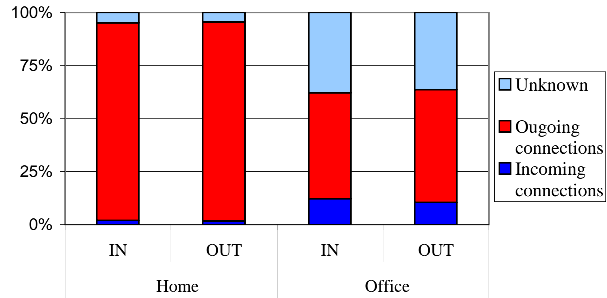

Figure 14: Fraction of packets generated by incoming vs. outgoing connections. For home and office, both received and transmitted packets.

an alternate strategy. We classify TCP and UDP packets based on the connections they belong to and categorize connections as incoming vs. outgoing. Our interest in this classification is because we suspect that a large portion of packets are likely to belong to outgoing connections. And while a host might wake for incoming connections, waking for outgoing connections might well be avoidable (for reasons discussed below). From the results in Fig. 14, we see that outgoing connections do indeed dominate. Now for a sleeping machine, there are three possibilities for these outgoing connections: (1) the connection was initiated by the host before the idle period—in this case, such traffic might not be ignorable if the host/proxy wants to maintain this connection, hence we hope this percentage of traffic is small, (2) the connection was initiated but failed (3) the connection was initiated by the host after the start of the idle period; for a sleeping host, these connections would either simply never have been initiated (if the connection were deemed unncessary) or, the host would be deliberately woken to initiate these connections (if the connection were deemed necessary, as for services scheduled to run during idle times). For the former, the traffic can simply be ignored from our accounting and, in the latter case, such scheduled processing is easily batched and hence needn't disrupt sleep. Hence for all but the first case, waking the machine might be avoidable. We plot this breakdown of outgoing connections in Figure 15. We see that only a relatively small percentage of outgoing connections – always less than 25% – belong to the first category which might require waking the host. Based on this, we speculate that, it might be possible to eliminate or ignore much of even unicast traffic.

Early in this section, we asked whether one might identify a small set of of protocols or proxy behaviors that could yield significant savings. We find that, the answer is positive in the case of multicast and broadcast but less clear for unicast traffic. In the next section we consider the implications of our traffic analysis for proxy design.

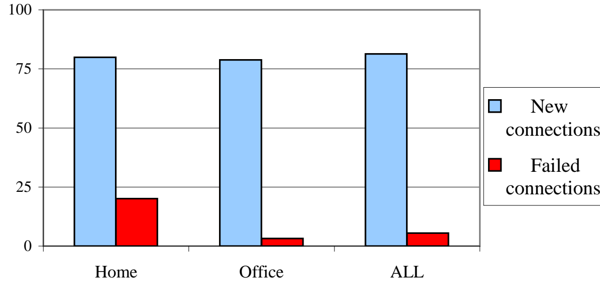

Figure 15: For outgoing connections: the fraction of incoming packets that belong to new connections and failed connection attempts.

5 A Measurement-driven Approach to Proxy Design

Having studied the nature of idle-time traffic, we now apply our findings to the design of a practical power-saving proxy. We start in Section 5.1 by extracting the high-level design implications of our traffic analysis from the previous section. Building on this, in Section 5.2, we illustrate the space of design tradeoffs by considering four specific examples of proxies. In Section 5.3, we distill our findings into a proposal for a core proxy architecture that offers a single framework capable of supporting the broad design space we identify.

5.1 Design Implications

At minimum, a power-saving proxy should: (a) allow the host to sleep for a significant fraction of the time, and (b) maintain the basic network presence of the host by ensuring remote entities can still address and reach the machine and the services it supports. Beyond this, we have a significant margin of freedom in choosing how a proxy might handle the remaining idle-time traffic and applications. Viewed through this lens, our results from Section 4 lead us to differentiate idle-time traffic along two different dimensions. The first classifies traffic based on the need to proxy the traffic in question:

(1) don't-wake protocols: these are protocols that generate sustained and periodic traffic and hence, ideally, would be dealt with (by a proxy) *without* waking the host since otherwise the host would enjoy little sleep. Examples of such protocols identified in the previous section include IGMP, PIM, ARP. Table 1 lists a set of protocols we classify as don't-wake.

(2) don't-ignore protocols: these are protocols that require attention to ensure the correct operation of higher-layer protocols and applications. For example, we must ensure the DHCP lease on an IP address must be maintained and that a machine must respond to NetBIOS name queries to ensure the services it runs over NetBIOS remain addressable. The protocols we identified as don't-ignore are listed in Table 1. Note that the list of don't-wake and don't-ignore protocols need not be mutually

Don't wake	HSRP, ARP, PIM, NBDGM, ICMP, IGMP, SSDP
Don't ignore	ARP (for me), NBNS, DHCP (for me)

Table 1: Protocols that shouldn't cause a wake up (too expensive in terms of sleep), and protocols that should not be ignored (for correctness).

Ignorable	HSRP, PIM, ARP (for others), IPX, LLC, EIGRP, DHCP	
Mechanical Response	Protocol	State
	ARP	IP address
	NBNS	NB names of machine and local services
	SSDP	Names of local plug-n-play services
	IGMP	Multicast groups the interface belongs to
	ICMP	IP address
	NBDGM	NB names of machine and local services. Ignores pkts. not destined to host, wakes host for rest

Table 2: Protocols that can be handled by ignoring or by mechanical response. We classify DHCP as ignorable because we choose to schedule the machine to wake up and issue DHCP requests to renew the IP lease – an infrequent event.

exclusive; for example, ARP traffic is both frequent and critical and hence falls under both categories.

(3) policy-dependent traffic: for the remainder of traffic, the choice of whether and how a proxy should handle the traffic is a matter of the tradeoff the user (or software designer) is seeking to achieve between the sophistication of idle-time functionality, the complexity of the proxy implementation and energy savings. We shall explore these tradeoffs in the context of concrete proxy implementations in Section 5.2.

A complementary dimension along which we can classify traffic is based on the complexity required to proxy the traffic in question:

(A) ignorable (drop): this is traffic that can safely be ignored. Section 4 identified several such protocols and the top ranked. of these are listed in Table 2. Comparing Tables 1 and 2, we see that (fortunately) there is a significant overlap between `don't-wake` and ignorable protocols. Policy-dependent traffic/applications that are deemed unimportant to maintain during idle times could likewise be ignored while `don't-ignore` protocols obviously cannot be.

(B) handled via mechanical responses: this includes incoming (outgoing) protocol traffic for which it is easy to construct the required response (request) using little-to-no state transferred from the sleeping ho.nction is somewhat subjective, based For example, a proxy can easily respond to NetBIOS Name Queries asking about local NetBIOS services, once these services are known by the

proxy. Table 2 lists key protocols that can be dealt with through mechanical responses.

(C) require specialized processing: this covers protocol traffic that, if proxied, would require more complex state maintenance (transfer, creation, processing and update) between the proxy and host. For example, consider a proxy that takes on the role of completing ongoing p2p downloads on behalf of a sleeping host – this requires that the proxy learn the status of ongoing and scheduled downloads, the addresses of peers, *etc.* and moreover that the proxy appropriately update/transfer state at the host once it resumes. In theory, specialized processing would be attractive for `policy-dependent` traffic that is both important and frequently-occurring (since otherwise we could simply drop unimportant traffic and wake the host to process infrequent traffic).

Of course, in addition to the the above (classes A-C), for traffic that a proxy doesn't ignore but doesn't want/know to handle a proxy always has the option of waking the host. Essentially the decision of whether to handle desired traffic in the proxy versus waking the host represents a tradeoff between the complexity of a proxy implementation and the sleep time of hosts.

5.2 Example Proxies

We now present four concrete proxy designs derived from the distinctions drawn above. We select these proxies to be illustrative of the design tradeoffs possible but also representative of practical and useful proxy designs.

proxy_1 We start with a very simple proxy that: (1) ignores all traffic listed as ignorable in Table 2 and (2) wakes the machine to handle all other incoming traffic. Besides clearly ignorable protocols, we choose to also ignore traffic generated by the Bigfix application (TCP port 63422) , which we previously identified (Section 4) to be one of the *big offenders*. We do so because this traffic is a) not representative for non-Intel machines, and b) the application is very badly configured – sending very large amounts of traffic for little offered functionality – making sleep almost impossible.

This proxy is simple – it requires no mechanical or specialized processing. At the same time, because it makes the conservative choice of waking the host for all traffic not known to be safely ignorable, this proxy is fully *transparent* to users and applications, in the sense that the effective behavior of the sleeping machine is never different from had it been idle (except for the performance penalties due to the additional wake-up time).

proxy_2 Our second proxy is also fully transparent, but takes on greater complexity in order to reduce the frequency with which the machine must be woken. This proxy: (1) ignores all traffic listed as ignorable in Table 2, and (2) issues responses for protocol traffic listed in the same table as to be handled with mechanical responses and (3) wakes the machine for all other incoming traffic. Since this proxy needs more state to generate mechanical responses (*e.g.,* the NetBIOS Names of local services, needed to answer NBNS queries), it can also use this extra information to selectively ignore more packets than `proxy_2` (*e.g.,* ignore all NetBIOS datagrams not destined for local services).

proxy_3 Our third proxy generates even deeper savings by only maintaining a small set of applications, (chosen by the user) operable during idle times, while ignoring all other traffic. We use telnet, ssh, VNC, SMB file-sharing and NetBIOS as our applications of interest. This proxy performs the same actions (1) and (2) as implemented by `proxy_2` (ignore and responds to the same set of protocols), but it (3) wakes up for all traffic belonging to any of telnet, ssh, VNC, SMB file-sharing and NetBIOS and (4) drops any other incoming traffic. Relative to our previous example, `proxy_2` is less transparent in that the machine appears not to be sleeping for some select remote applications, but is inaccessible to all others.

proxy_4 All the above proxies implement functionality related to handling incoming packets. In our final proxy, we also consider waking up for scheduled tasks initiated locally. This proxy behaves identically to `proxy_3` with respect to incoming packet, but supports an additional action: (5) wake up for the following tasks (for which we assume that the system is configured to wake up in order to perform them): regular network backups, antivirus (McAfee) software updates, FTP traffic for automatic software updates, and Intel specific updates.

Evaluating tradeoffs In the following we compare the sleep achievable by our 4 proposed proxies, and compare it with the baseline WoP case. We perform this evaluation for both office and home environments, and in each case we evaluate 3 possible values for transition times ts: 5, 10, and 60 seconds. The first of these (5s) is a very optimistic transition time, not achievable today using S3 sleep states, but foreseeable in the near future (today, Microsoft Vista specifications require computers to resume from S3 sleep in under 2s [18]). The second (10s) is representative of the shortest transitions achievable today [6], and the last (1min) is representative of a setting that allows almost a minute for processing subsequent relevant network packets before going to sleep again. The advantage of using a very short timer before going to sleep is the increased achievable sleep. The disadvantage is that the delay penalty for waking the host will be incurred at more packets. In the extreme case of very short sleep timers, this could make remote applications sluggish and un-responsive. For the wake events generated by scheduled tasks, we use a longer transition time (and thus a longer sleep timer value) of 1min, since such tasks usually take longer time to complete.

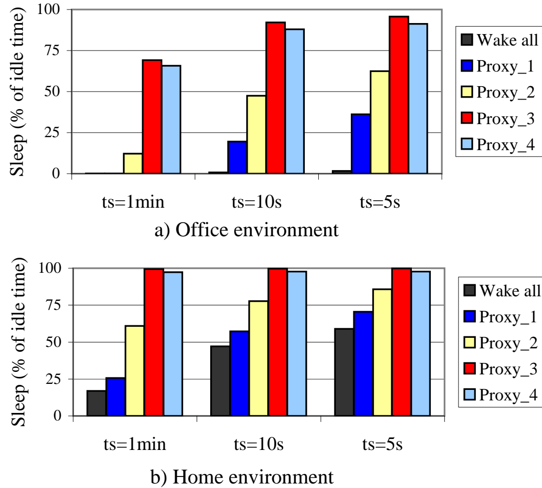

Figure 16: Savings achieved by different proxies in home and office environments.

Examining the performance of our proxies, we make the following high-level observations: *a)* At one end of the spectrum, `proxy_1`(the simplest) is inadequate in office environments, and borderline adequate in home environments. *b)* At the other end of the spectrum we have `proxy_3`, which only handles a select number of applications, but in return achieves good sleep in all scenarios – more than 70% of idle time even in the office and with a transition time of 1minute. *c)* The efficiency of proxy 2 depends heavily on environment. While the additional complexity (compared to `proxy_1`) makes it a good fit in home environments (sleeping close to 60% even for $ts = 1min$), having to handle all traffic makes it a worse fit for the office (sleeping $\approx 12\%$ for the same transition time). This shows that, unless they support a large number of rules, transparent proxies are a better fit for home, but not the office. *d)* The best tradeoff between functionality and savings, and therefore the appropriate proxy configuration, depends on the operating environment. *e)* Since scheduled wake-ups are typically infrequent, the impact they have on sleep is minimal – in our case, `proxy_4` sleeps almost as much as `proxy_3` in all considered scenarios.

5.3 A strawman proxy architecture

Our study leads us to propose a simple proxy architecture that offers a unified framework within which we can accommodate the multiplicity of design options identified above. The proposal we present is a high-level one since our intent here is merely to provide an initial sketch of an architecture that could serve as the starting point for future discussion on standardization efforts.

The core of our proposal is a table—the power-proxy table (PPT)—that stores a list of *rules*. Each rule describes the manner in which a specified traffic type should be handled by the proxy when idle. A rule consists of a *trigger*, an *action* and a *timeout*.

Triggers are either timer events or regular expressions describing some network traffic of interest. When a trigger's timer event fires or if an incoming packet matches a trigger's regular expression, the proxy executes the corresponding action. If the action involves waking the host, the timeout value specifies the minimum period of time for which the host must stay awake before contemplating sleep again. To resolve multiple matching rules, standard techniques such as ordering the rules by specificity, policy, *etc.* can be used. The proxy table must also include a *default* rule that determines the treatment of packets that do not match on any of the explicitly enumerated rules.

We propose the following actions:

- drop: the incoming packet is dropped.
- wake: the proxy wakes the host and forwards the packets to it. Other packets buffered while waiting for the wake will be forwarded as well.
- respond(`template, state`): the proxy uses the specified *template* to craft a response based on the incoming packet and some *state* stored by the proxy. This action is used to generate mechanical responses as described below.
- redirect(`handle`): the proxy forwards the packet to a destination specified by the `handle` parameter. This is used to accommodate specialized processing as described below.

A response template is a *function* that computes the mechanical response based on the incoming packet and one or more *immutable* pieces of state. This means that our function does not maintain or change any state. There is no state carried over between successive incoming packets (such as sequence numbers), and no state transfer between the proxy and the host upon wake-up. We choose to support this functionality because a) it is relatively simple to implement in practice and b) it covers most of the non-application specific traffic, as shown in Section 4, and illustrated in our proxy examples.

To accommodate more specialized processing, we assume developers will write application-specific stubs and then enter a redirect rule into the proxy's PPT, where the `handle` specifies the location to which the proxy should send the packet. Such stubs can run on machine accessible over the network (*e.g.,* a server dedicated to proxying for many sleeping machines in a corporate LAN), or on a low-power micro-engine supported on the local host (*e.g.,* a controller on the motherboard, or a USB-connected gumstick). In all these cases, the handle would be specified by its address, for example a (IP address, port) combination. The redirect abstraction thus allows us to accommodate specialized processing without embedding application-specific knowledge into the core proxy architecture.

The external API to this proxy is twofold: (1) APIs to

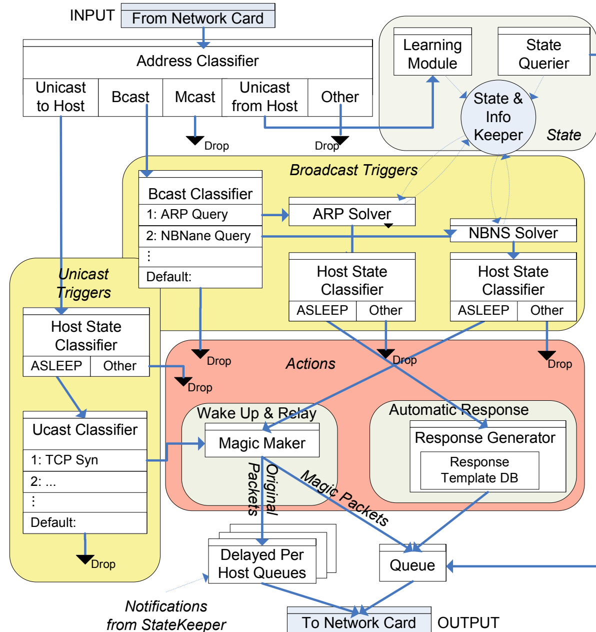

Figure 17: Example Click implementation.

activate/deactivate the proxy as the host enters/exits sleep and (2) APIs to insert and delete rules. The process by which to install and execute stubs is outside of the core proxy specification which only provides the mechanism to register and invoke such stubs. The architecture is agnostic to where the proxy runs allowing implementations in hardware (e.g., at host NICs), in PC software (e.g., a proxy server running on the same LAN) or in network equipment (e.g., a firewall, NAT box).

Finally, the use of timer events to wake a host already exists today. Our contribution here is merely to integrate the mechanism into a unified proxy architecture.

5.4 Proxy Prototype Implementation

To illustrate the feasibility of our architecture, we build a simple proxy prototype using the Click modular router [17]. We choose to deploy the proxying functionality in a standalone machine responsible for maintaining the network presence of several hosts on the same LAN. To allow our proxy (let us call it P) to sniff the traffic for each host, we ensure that P shares the same broadcast domain with these hosts. This can be achieved either by connecting the proxy and the machines to a common network HUB, or by configuring the LAN switch to forward all traffic to the port that serves P.

In our initial design, we don't implement proxies that involve transferring state between the host and the proxy. Instead, P learns the pieces of state required (e.g. the IP address and the Netbios name for each host) by sniffing host traffic and extracting the state exchanged (e.g. ARP and NBNS exchanges). This design circumvent the need for any end-host modifications, and support proxying for machines with different hardware platforms (new and old) and operating systems. The proxy requires minimal configuration (a list of the MAC addresses of the hosts that need to be proxied), and can be incrementally deployed as a low-power stand-alone network box. Once low-power proxying standards are developed [12], the design can be extended to support state transfer, and achieve even deeper energy savings.

Our prototype implements very basic proxying functionality, but the software architecture (presented in Figure 17) can be easily extended to more protocols and use cases. Currently, we support three types of actions: *wake*, *respond* and *drop*. The proxy awakes its hosts for TCP connection requests (incoming TCP SYN packets) and incoming Netbios Name Queries for the host's NB name. If such a "wake packet" for a sleeping host arrives, P buffers the request, sends a magic packet to wake the host, and relays the buffered packet once the host becomes available. The proxy responds automatically to incoming ARP requests, and drops all other incoming packets. In relation to the examples discussed in Section 5.2, this prototype has a *simple* and *non-transparent* design. To determine whether a host is awake, the proxy sends periodic ARP queries to each host; if these queries receive no response, the host is assumed to be asleep. When the proxy attempts to wake a host and fails repeatedly, the host is assumed to be off, rather than just asleep, and the proxy ceases to maintain its network presence.

Figure 17 presents the software architecture of our Click proxy, and highlights the mapping between Click modules and the generic categories of *triggers*, *actions* and *state*, discussed in the strawman proxy architecture.

We test our Click-based proxy implementation by installing it on one of our enterprise desktops, and configuring the proxy to maintain the network presence of several IBM ThinkPad laptops. We use this deployment to measure the delays experienced by applications waking a sleeping host, and find these to be surprisingly low: $2.4s$ on average, and $4s$ at maximum – much lower than the $30s$ TCP SYN timeout. These delays includes the host wake-up delay ($\approx 1.4s$), and the additional time required for the proxy to detect the state change and relay the buffered packet causing the wake ($\approx 1s$). We defer a comprehensive deployment-based evaluation to future work.

6 Power-Aware System Redesign

In this section we consider approaches that might assist in reducing idle-time energy consumption by either simplifying the implementation of proxies or altogether obviating the need for proxying.

6.1 Software Redesign

Our idle traffic analysis shows that solutions relying on Wake-on-LAN functionality face the following chal-

lenges: (i) It is difficult to decide if various packets and protocols warrant a machine wake-up.(ii) Hosts receive many packets even when idle (3 per second on average). (iii) Many protocols exchange packets periodically, preventing long quiet periods when hosts could sleep. These challenges could be dealt with at both application and protocol level:

Power-aware application configuration Today, applications and services are typically designed or configured without taking into account their potential impact on the power management at end-systems. For example, in Section 4.4 we discussed a tool called Bigfix, that checks if network hosts conform to Intel's corporate security specifications. This application was configured to perform these checks very aggressively, continuously generating large amounts of traffic. Under a WoL approach, this application alone would have made prolonged sleep virtually impossible.

This is a perfect example of the behaviour that could be avoided by configuring applications to be more power-aware, and perform periodic tasks less frequently, reducing the volume of network traffic seen by hosts.

Protocol Specification The decision to ignore or wake on a packet can be difficult, and involves protocol parsing, maintaiing a long set of filters and rules, and for some protocols host or application-specific state.

To eliminate the complexity of this decision, and allow hosts to sleep longer even when using very simple rules for waking, protocols could be augmented to carry explicit power-related information in their packets. An example of such information would be a simple bit indicating whether a packet can be ignored.

Protocol Redesign We believe these principles should be followed when designing power-aware protocols.

Consideration when using broadcast and multicast: We saw earlier that broadcast and multicast are mainly responsible for keeping hosts awake. This type of traffic could be substantially reduced by redesigning protocols to use broadcasts sparingly. Some protocols are particularly inefficient in this respect. For example, all NetBIOS datagrams are always sent over Ethernet broadcast frames. These frames are received by all hosts on the LAN, and then discarded by most of them. This ranks NBDGM as one of the top "offenders", yet this could be easily avoided by using unicast transmissions when possible. Another approach is based on the observation that many service discovery protocols have redundant functionality. This redundant functionality could conceivable be replaced by a single service that can be shared by a multiplicity of applications.

Synchronization of periodic traffic: One way to increase the number of long periods of network quiescence would be to identify protocols that use periodic updates/message exchanges, and try to synchronize, or bulk these exchanges together. This would allow machines to periodically wake up, process all notifications and request, and resume sleep.

Complementing soft state: Many protocols (*e.g.,* SSDP, NetBIOS, etc.) maintain and update state using periodic broadcast notifications/ For such protocols (and for similar applicatios), it would be essential to make them disconnection-tolerant, by providing complementary *state query* mechanisms that could be used quickly build up-to-date copies of the soft state upon waking. This would enable ignoring any soft state notifications. Today, such query mechanisms exist only for some of these protocols, and they are often inefficient.

6.2 Hardware Redesign

A general goal of energy saving mechanisms, especially hardware designs, is to lead the industry towards energy proportional computing [8]. If energy consumption of a machine would accurately reflect its level of utilization, the energy would be zero when idle. Sleep states are a step in this direction, P-states (low power active operation) are another. Related to this, it would be very desirable to expose power saving states (S states) that feature better transition times, even if they offer smaller savings. Given the small inter-packet gaps, these states will come in handier than the deep-sleep ones.

7 Related Work

The notion that internetworked systems waste energy due to *idle* periods has been frequently reiterated[14, 13, 16, 19, 10, 7, 15]. Network presence proxying for the purpose of saving energy in end devices was first proposed over ten years ago by Christensen *et al.*; in follow-up work [11] the authors quantify the potential savings using traffic traces from a single dormitory access point and in [13] examine the traffic received at a single idle machine to identify dominant protocols and discuss whether these can be safely ignored. Our work draws inspiration from this early work extending it with a large-scale and more in-depth evaluation of idle-time traffic in enterprise and home environments. A more recent proposal [7]. postulates the notion of "selective connectivity", whereby a host can dictate or manage its network connectivity, going to sleep when it does not want to respond to traffic.

There is an extensive literature on energy saving techniques for individual PC platforms. Broadly, these aim for reduced power draws at the hardware level and faster transition times at the system level. These offer a complementary approach to reducing the power draw of idle machines; if and when these techniques lead us to perfectly "energy-proportional" computers, the idle-time consumption will be less problematic and proxying will

fade in importance. So far however, achieving such energy proportionality has proved challenging.

In parallel work [6], the authors build a prototype proxy supporting BIT-TORRENT and IM as example applications. Our work considers a broader proxy design space, evaluating the tradeoffs between design options and the resultant energy savings informed by detailed analysis of network traffic. In relation to our design space, their proxy supports BT and IM using application stubs.

8 Conclusions

In general, the question of how a proxy should handle the user-idle time traffic presents a complex tradeoff between balancing the complexity of the proxy, the amount of energy saved, and the sophistication of idle-time functionality. Through the use of an unusual dataset, collected directly on endhosts, we explored the potential savings, requirements and effectiveness of technologies that aim to put endhost machines to sleep when users are idle. For the first time here, we dissect the different categories of traffic that are present during idle times, and quantify which of them have traffic arrival patterns that prevent periods of deep sleep. We see that broadcast and multicast traffic constitute a substantial amount of the background chatter due to service discovery and routing protocols. Our data also revealed a significant amount of outgoing connections, generated in part by enterprise applications. We tried to identify which traffic can be ignored and found that most of the broadcast and multicast traffic, as well as roughly 75% of outgoing connections, appears safely ignorable. Handling unicast traffic is more involved because it harder to infer the intent of such traffic, and often needs some state information to be maintained on the proxy.

After having studied our traffic and the sleep potential those patterns contain, we discuss the design space for proxies, and evaluate the savings offered by 4 sample proxy designs. These cases reveal the tradeoffs between design complexity, available functionality and energy savings, and discuss the appropriateness of various design points in different use environments, such as home and office.

Finally, we present a general and flexible strawman proxy architecture, and we build an extensible Click-based proxy that exemplifies one way in which this architecture can be implemented.

Aknowledgments

We thank Robert Hays, Ken Christensen, Gianluca Iannaccone, Eric Mann, Rabin Patra and Kevin Fall for their suggestions, and Eve Schooler for her help collecting trace data. We also thank the anonymous reviewers and our shepherd Yuanyuan Zhou for their useful feedback.

References

[1] Alert Standard Format (ASF) Specification, v2.0, DSP0136, Distributed Management Task Force. `http://www.dmtf.org/standards/asf`.

[2] ENERGY STAR Program Requirement for Computers. Version 4.0. `http://www.eu-energystar.org`.

[3] The Wireshark Network Protocol Analyzer. `http://www.wireshark.org/`.

[4] Power and Thermal Management in the Intel Core Duo Processor. In *Intel Technology Review, Vol.10*, 2006.

[5] Advanced configuration and power interface. `http://www.acpi.org`.

[6] Y. Agarwal, S. Hodges, J. Scott, R. Chandra, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *NSDI*, 2009.

[7] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an energy-efficient future internet through selectively connected end systems. In *HotNets*, 2007.

[8] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[9] BRO IDS. `http://www.bro-ids.org`.

[10] K. J. Christensen and F. B. Gulledge. Enabling power management for network-attached computers. *International Journal of Network Management*, 1998.

[11] K. J. Christensen, C. Gunaratne, B. Nordman, and A. D. George. The next frontier for communications networks: power management. *Computer Communications*, 27(18):1758–1770, 2004.

[12] ECMA International. TC32-TG21 – Proxying Support for Sleep Modes. `http://www.ecma-international.org/memento/TC32-TG21-M.htm`.

[13] C. Gunaratne, K. Christensen, and B. Nordman. Managing Energy Consumption Costs in Desktop PCs and LAN Switches with Proxying, Split TCP Connections, and Scaling of Link Speed. *International Journal of Network Management*, October 2005.

[14] M. Gupta and S. Singh. Greening of the internet. In *ACM SIGCOMM, Karlsruhe, Germany*, August 2003.

[15] Intel remote wake technology. `http://support.intel.com/support/chipsets/rwt/`.

[16] J. Klamra, M. Olsson, K. Christensen, and B. Nordman. Design and implementation of a power management proxy for universal plug and play. *Proceedings of the Swedish National Computer Networking Workshop (SNCW)*, Sep 2005.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[18] Microsoft Window Vista Logo Program Requirements and Policies. `http://www.microsoft.com/whdc/winlogo/hwrequirements.mspx`.

[19] B. Nordman. Networks, Energy, and Energy Efficiency. *Cisco Green Research Symposium*, March 2008.

[20] C. Webber, J. Roberson, M. McWhinney, R. Brown, M. Pinckard, and J. Busch. After-hours power status of office equipment in the usa. *Energy*, 31(14):2823–2838, Nov 2006.

[21] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and practical limits of dynamic voltage scaling. In *DAC*, 2004.