# Sleepless in Seattle No Longer

Joshua Reich†, Michel Goraczko, Aman Kansal, Jitendra Padhye

†*Columbia University, Microsoft Research*

**Abstract:** In enterprise networks, idle desktop machines rarely sleep, because users (and IT departments) want them to be always accessible. While a number of solutions have been proposed, few have been evaluated via real deployments. We have built and deployed a sleep proxy system at Microsoft Research. Our system has been operational for six months, and has over 50 active users. To the best of our knowledge, this paper is the first to report on lessons learned from building, deploying and running a sleep proxy system on a real network. Overall, we find that our system allowed user machines to sleep quite well (most sleeping over 50% of the time), but much potential sleep time was missed due to IT management tasks that play havoc with machine sleep. We suggest a number of ways to fix this problem. We also discover and address a number of issues overlooked by prior work, including complications caused by IPsec. We found certain popular cloud-based applications did not work well with our design, and we deployed an ad-hoc fix to the problem. We believe our experience and insights will prove useful in guiding the design of future sleep solutions for enterprise networks.

## 1 Introduction

A number of studies [29, 3, 40, 5] have noted that most office machines are left on irrespective of user activity. At Microsoft Research, we find hundreds of desktop machines awake, day or night – a significant waste of both energy and money. Indeed, potential savings can amount to millions of dollars per year for large enterprise networks [39].

As businesses become more energy conscious, more desktops may be replaced by laptops. However, currently desktops comprise the majority of enterprise machines [17] with hundreds of millions additional desktops being sold every year [21, 26, 17]. Where users make heavy use of local resources (e.g., programming, engineering, finance), desktops continue to be the platform of choice. Hence, managing desktop power consumption is an area of both active research [29, 3, 40, 5] and commercial [2, 38, 30, 33] interest.

So, why do machines stay awake when users are inactive? The most common reason is that users and IT administrators want remote access to machines at will. Users typically want to log into their machines or access files remotely [3], while IT administrators need remote access to backup, patch, and otherwise maintain machines.

A number of solutions to this problem have been proposed [33, 3, 5, 29]. The core idea behind these is to allow a machine to sleep, while a *sleep proxy* maintains that machine's network presence, waking the machine when necessary. Some of these proposals rely on specialized NIC hardware [33, 3]; others advocate use of network-based proxies [5, 29].

Unfortunately, most previous work has been evaluated either using small testbeds [3, 29, 5] or trace-based simulations [29]. We are not aware of any paper detailing the deployment of any of these proxying solutions in a real enterprise network on actual user machines. This is disconcerting: systems that work well on testbeds often encounter potentially serious challenges when deployed in real networks.

This paper aims to fill that gap. We describe the design and deployment of a network-based sleep proxy on our corporate network. To our knowledge, this is the first paper to report such results from a real network.

Our design expands on the network proxy approach proposed in [5, 29]. Our architecture has two core components: a per-subnet sleep proxy, and a sleep notifier program that runs on each client. The sleep notifier alerts the sleep proxy just before the client goes to sleep. In response, the proxy sends out ARP probes [13] to ensure that all future traffic meant for the sleeping client is delivered to the proxy instead. The proxy then monitors this traffic, responding to some (e.g., ARP Requests) on the client's behalf, waking the sleeping client for certain specified traffic, and ignoring the rest. Client wakeup is accomplished by sending Wake-on-LAN (WOL) [42] packets.

We chose this approach because it was easier to deploy than hardware [3] or virtualization-based solutions [8]. We provide in-depth discussion of the merits of this and alternative approaches in Sec. 3.

Our system has been operational for over six months, and currently has over 50 active users. Our software is deployed on user's primary workstations, not test machines. Indeed, this preliminary deployment has been so successful that our IT department has begun recommending our system to users. We have instrumented our system extensively; capturing numerous details about sleep and wakeup periods, why machines wake up and why they stay up. Instead of using generic estimates of PC power consumption, we use a sophisticated software-based, model-driven power measurement system to more accurately measure machine power consumption.

In this paper we describe a number of practical issues we encountered when deploying our sleep proxy in a corporate network. Many of these issues have been overlooked by previous work. For example, our implementation must not only deal with vanilla IPv4 and IPv6 packets, but also tunneled packets. Our corporate network uses IPsec, and we show that without the right setup, use of IPsec can severely impact the operation of a sleep proxy. We describe race conditions that arise when the sleep proxy attempts to redirect traffic from sleeping client to itself, and provide a practical solution. We show how issues such as DHCP lease expiration and proxy failure can be handled in a simple manner. We also point out how our sleep proxy approach interferes with the operation of two popular cloud-based applications, and describe a simple solution we devised.

By analyzing trace data from our system, we find that our system allowed the clients to sleep quite well. Many machines slept over 50% of the time. However, the average power savings was only 20%, which casts a pall over the optimistic predictions made in [29, 3]. We find that while users do access their machines remotely, remote accesses by IT department are the primary cause of machine wake ups. In fact, we discovered that a set of IT servers and applications were the main culprits for keeping machines awake as well. IT server connection attempts repeatedly woke sleeping machines. In one extreme case, a single machine was contacted over 400 times within a two-week period. Additionally, some of the locally running IT applications (e.g., virus scanners) kept machines up by temporarily disabling sleep functionality. We also identify bugs in common software (e.g., Adobe Flash player) that interfere with proper sleeping.

We conclude that fixing the IT setup will yield the greatest improvement in power savings in our environment. For example, IT servers could coordinate and perhaps "rate-limit" how and when they wake machines and keep them awake. In contrast, simple strategies like more aggressive sleep policies will be much less effective. Overall, we believe that the insights gleaned from our experience will be useful in guiding the design and deployment of future sleep solutions in enterprise networks.

## 2 Related work

While the basic concept of sleep proxying has been known for some time [15], it has received much renewed attention lately [5, 28, 3]. Among recent publications, the two most closely related to our work are [3] and [28].

In [3], the authors describe a hardware-based solution. They augment the NIC with a GumStix [19] device, which is essentially a small form factor, low-powered PC. Once the host machine goes to sleep, the GumStix device takes over. It handles select applications (e.g., file downloads) on behalf of the host PC, but wakes up the host PC for more complex operations. While this approach is more flexible than the sleep proxy we have built, it is far less practical for two reasons. Not only is additional hardware required on every PC, but both applications and host OS modification are required to enable state transfer between host PC and GumStix device. Both these requirements are a substantial barrier to widespread deployment of this technology. In contrast, our approach requires neither extra hardware, nor application modifications.

In [28], the authors carry out an extensive trace-based study of network traffic, arguing for a network-based sleep proxy. Their primary finding is that in an enterprise environment, broadcast and multicast traffic related to routing and service discovery cause substantial network 'chatter", most of which can be safely ignored by a sleep proxy. They also posit that most unicast traffic directed to a host after it has gone to sleep can also be ignored, so long as the host is woken when traffic meant for a set of pre-defined applications arrive (early work had focused on avoiding disrupting existing TCP connections [14, 22]). Based on these insights, they propose a number of sleep proxy designs.

While our proxy design builds upon the insights of [28], we make several additional contributions in this paper. First, unlike [28], our design includes a client-side agent, which considerably simplifies the overall architecture, making it robust, and virtually configuration-free. Second, we build and deploy our sleep proxies in a real operational network on users' primary workstations. In contrast, the prototype in [28] was tested only a small testbed without real users, and did not address challenges such as IPsec traffic and proxy failures. Third, our instrumentation measures sleep and wakeup behavior of operational machines. We document why machines do not sleep, when and why they wake, etc. Fourth, our deployment includes a model-based power measurement competent. Since machine power usage can vary by 2.5x while awake, our power estimates provide significantly greater fidelity than the "one size fits all" model used by [28].

We now turn to commercial systems. Intel offers two hardware-based solutions, Remote Wakeup Technology (RWT) [33] and Active Management Technology (AMT) [6], that can remotely wake up a sleeping machine. AMT is primarily meant for management tasks (e.g., out of band access for asset discovery, remote troubleshooting). RWT is more closely related to our work. RWT requires the NIC of the sleeping machine to maintain a persistent TCP connection to an authorized server. The NIC wakes up the host machine upon receiving a special message over this TCP connection. RWT requires modification of client applications and works only with Intel hardware. Even the wakeup service has to be digitally signed by Intel. In contrast, our solution is entirely software-based, hardware-agnostic, and requires no application modification.

Apple has recently released a sleep proxy geared toward home networks that works only with select Apple hardware [7]. For enterprise networks, systems such as 1E [30], Adaptiva [2] and Verdiem [38] are available. The primary focus of these systems is to enable the system administrator to estimate power usage, and wake up sleeping machines to perform management tasks such as patching. A number of industry participants are trying to standardize basic sleep proxy functionality [16].

Several other approaches to saving power, such as power-proportional computing [9], dynamic voltage and frequency scaling [34] and the TickLess kernel [36] have been investigated, and can be used in conjunction with our system. Researchers have also looked at networking hardware and software stacks as potential targets for power savings. Examples include [20, 29, 12, 11]. [4, 37, 25] examine data center power consumption and savings approaches appropriate for that environment.

Prior work has shown that CPU utilization and certain performance counters can be used to estimate computer energy use [32, 10, 18]. Our power estimation technology provides enhanced accuracy by considering additional factors not considered in prior work, such as processor DVFS states and monitor power.

## 3 Design Goals & Alternatives

As discussed earlier, enterprise users often do not let their machines sleep as they may require remote access. Our goal in deploying a sleep proxy is to encourage users to allow their machines to sleep – by ensuring their machine will wake on remote access attempts. We now describe the basic functionality required from a sleep proxy, define our design goals, and describe design alternatives. Before we begin, we note that our use of the term ”sleep” refers to ACPI S3 (suspend to RAM) [1]. Our system does support ACPI S4 and S5 as well.

### 3.1 Basic sleep proxy functionality

A *sleep proxy* detects when a *sleep client* machine ($M$) has gone to sleep, typically because that machine's *idle timeout* had been reached. The idle timeout is the amount of time the user must be inactive before the machine will go to sleep. In Windows, the idle timeout is typically 30 minutes, although a machine may stay up much longer (Sec. 5.2).

The proxy then monitors network traffic destined for $M$. Based on a pre-defined *reaction* policy, the sleep proxy may (a) ignore some of the traffic (e.g., ARP requests not for $M$ , (b) respond to some of the traffic on behalf of $M$ (e.g. ARP requests for $M$), and (c) for the selected traffic (e.g. TCP SYNs for $M$) wake $M$.

Our sleep proxying system plays no role in determining *when* or *whether* a machine sleeps. That decision is left to each machine's operating system, as we believe the local OS is in the best position to determine when a machine can be safely put to sleep.

## 3.2 Design goals

Our goal is to build a practical, deployable sleep proxy for typical corporate networks, composed of desktop machines with wired connectivity. In a typical usage scenario, the user's machine goes to sleep, and wakes automatically on remote connection attempts (e.g., log in, file access).

The design of our sleep proxy was directed by four goals. $(a)$ The system had to save as much power as possible, $(b)$ while minimizing disruptions to users. It is critically important to ensure a sleeping machine is always be woken when the user desires access: otherwise no one would use the system. Furthermore, the system had to be $(c)$ easy to deploy and maintain, since we operated without the benefit of a large IT staff. We explicitly decided not to add hardware to client machines, as it makes deployment significantly harder. Finally, we required the architecture be $(d)$ scalable and extensible, since the system had to operate in a dynamic live network

It was not our goal to support laptops *per se* as they offered much less opportunity for power savings. They consume much less power when active, and are more often put to sleep by users. Thus, while some of the work we have done is applicable to laptops, we do not address laptop-specific challenges such as mobility in our work.

As all the machines in our network run Windows, some details of our implementation are Windows specific. However, our architecture is designed to be OS agnostic.

## 3.3 Design alternatives

We now consider three design alternatives, and evaluate them in light of our requirements.

### 3.3.1 NIC Pattern Matching

The first potential approach is to simply use the combination of *Wake-On-Pattern+ARP Offload*. This capability is available on most modern wired NICs.

**How it works:** The NIC effectively acts as the sleep proxy for the machine. It responds to incoming ARPs on behalf of the sleeping machine (*ARP Offload*), thereby maintaining the machine's network presence. The NIC can be programmed to detect specific patterns in incoming traffic, and wake up the host machine if a packet with specified pattern arrives (*Wake-On-Pattern*). The interface for specifying patterns [27] includes built-in support for IPv4 and IPv6 TCP-SYNs; one only need specify additional information (e.g., ports). Raw bit patterns with starting offset can also be specified.

**Pros:** These NICs are available on most modern machines, so no additional hardware needs to be deployed.

**Cons:** We found that for our purposes the capabilities offered by these NICs were not adequate. Our corporate network is quite complex: it supports IPv4, IPv6, v6-over-v4 and requires IPsec. To ensure machines were woken whenever users required access, we had to handle packets requiring flexible inspection (e.g. a TCP SYN in an ESP packet carried in an IPv6 packet, tunneled in an IPv4 packet - Sec. 4.3.2). While such packets may be detected by explicit bit-pattern matching, the complexity of doing so grows linearly with the number of open ports. This difficulty would only be exacerbated if the system administrator wanted to restrict the set of hosts allowed to wake a given machine. (see Sec. 8). Moreover, future needs may dictate deeper packet inspection beyond current NIC capabilities. Thus, this approach does not meet criteria (b) and (d) listed above. While some of these issues may be resolved by augmenting the NIC with extra hardware as done in [3], this would violate criteria (c).

### 3.3.2 Virtualization

**How it works:** Users install a hypervisor on their desktop, and then install and use a VM on top of the hypervisor. When the desktop machine needs to sleep, the VM is migrated to a hosting server. When necessary, (e.g. if the user wants to run a CPU intensive application), the desktop machine is woken, and the VM migrated back.
**Pros:** This approach is attractive because if the migration can be made seamless, the desktop does not have be woken up for transactions of even moderate complexity that can be carried out on the hosting server. As the machine can go to sleep without interrupting existing network connections, the machines an go to sleep much more often, and hence the power savings may be greater.
**Cons:** To deploy a system based on this approach, we would have had to install hypervisor on the end user systems , and boot their existing OS as a VM. Most users would not have agreed to such a drastic change to their work environment. Apart from taking a performance hit, virtualization may encounter problems with a number of common end-user devices (e.g., cameras, external drives), whose drivers do not always work well when virtualized.

An alternate approach might only virtualize certain applications (e.g. using Microsoft's Application Virtualization [8]). However, application virtualization does not address the requirements of IT department, which needs remote access to machines for management tasks such as patching and security scanning. Additionally, support for application virtualization varies by OS as do the mechanics of transferring virtualized applications to a proxy. Like a full virtualization approach, application virtualization requires significant sleep-client side configuration/modification.

### 3.3.3 Network-Based Sleep Proxies

This approach was proposed in [15], its feasibility recently given careful study by [28].
**How it works:** This approach relies on a separate machine acting as a sleep proxy for the sleeping machine. The sleep proxy detects when a client goes to sleep. It then modifies Ethernet routing (Sec. 4.3.1) to ensure that all packets destined for the sleeping machine are delivered to the sleep proxy instead. The proxy examines the packets, and wakes up the sleeping client when needed, by sending a Wake-On-LAN (WOL) [42] packet.

**Pros:** Very little hardware support is required from the client machine - the client NIC only needs to support WOL. As the sleep proxy runs on a separate, general purpose computer, it has great flexibility in handling incoming traffic for the sleeping machine. The sleep proxy can do complex, conditional packet parsing and can even wake the sleeping machine based on non-network events such as requests by system administrators, users entering the building (with support from building access systems), etc. As we shall see, this design also scales well (Sec. 7.5.2).
**Cons:** This design requires deployment of a sleep proxy on a separate machine. Also, existing network connections are broken. We argue that this is not an issue for typical corporate workloads (Sec. 4.4).

We have chosen this approach as it is both very easy to deploy and requires minimal changes to user machines. It affords great scalability and flexibility as the sleep proxy can be changed without disturbing the client machines.

## 4 Architecture

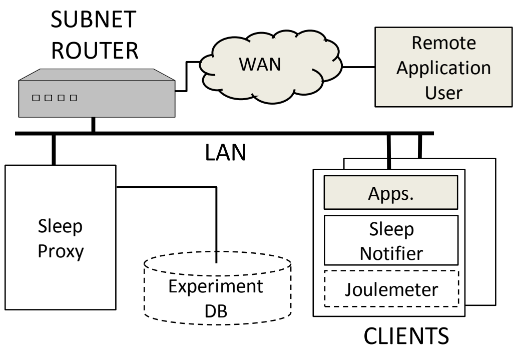


Figure 1: System block diagram. Blocks shaded gray represent existing components that are not modified in any way for the sleep proxy to work. Blocks with dashed outlines are part of our instrumentation setup.

The overall architecture of our system is shown in Figure 1. We require one sleep proxy per subnet. We also required the clients to install a small background service[1], *sleep notifier*. In this section, we will focus on the design of the sleep proxy and the sleep notifier which form the core of our solution. We discuss Joulemeter in Sec. 5.1

As discussed earlier, a sleep proxy responds to some traffic, wakes the sleep client for other traffic, and ignores the rest. Our choice of reaction policy is similar to that of the proxy scheme (*proxy3*), which [28] found provided the highest simulated power savings. This reaction policy, whose rationale is discussed in Sec. 4.4, responds mechanically to IP resolution requests (e.g., ARP) and wakes the sleep client only on TCP connection attempts to selected ports, ignoring other traffic.

Before digging into design details (Secs. 4.2 and 4.3), we provide a quick overview of how our system works.

[1]A daemon, in Unix terminology.

## 4.1 System Overview

Imagine a *sleep client* $M$ running sleep notifier. $M$'s sleep notifier registers with the OS to receive notification when the machine is about to go to sleep. At such time, the OS alerts the sleep notifier.

$M$'s sleep notifier then alerts the sleep proxy $S$ that $M$ is going to sleep, providing a list of $M$'s TCP ports in the *listening* state (actively listening for incoming connections). Assume that the SSH port, 22, is one such port.

Upon receiving the notification, $S$ adds $M$ to its list of proxied clients and sends out an ARP probe (Sec. 4.3.1), re-mapping the switched Ethernet to direct future packets for $M$ to the network port at which $S$ resides. $S$ now begins receiving traffic that was meant for $M$. $S$ responds to ARP requests and IPv6 *Neighbor Discovery* packets as if it were $M$, thereby maintaining $M$'s network presence and ensuring traffic for $M$ continues to arrive at $S$.

Some time later a remote client $C$ attempts to connect to the sleeping machine $M$, using SSH. As the first transport-layer action taken in establishing this new connection, $C$ sends a TCP SYN on port 22 to $M$ which the switched Ethernet routes to $S$.

Upon examining the packet, $S$ determines that it is a TCP SYN meant for $M$ and destined to a port on which $M$ was listening when it when to sleep. $S$ therefore wakes $M$ up by sending it a WOL packet (Sec. 3.3.3), removes $M$ from the proxied client list, and drops the TCP SYN. As $M$ wakes up, it sends its own ARP probes, which ensure that future traffic meant for $M$ will arrive at $M$'s network port. Meanwhile, $C$ retransmits this SYN following the normal TCP timeout. The retransmitted SYN arrives at $M$, who responds as normal, thereby establishing the TCP connection without $C$ being any the wiser, except for a small delay - quantified in Sec. 7.5.1.

## 4.2 The Sleep Notifier

Installing the sleep notifier on sleep clients greatly simplifies the overall design. As the service runs on user desktops, our aim is to make the sleep notifier code robust and stateless, requiring as simple configuration as possible.

The primary purpose of the sleep notifier is to notify the sleep proxy when the machine is going to sleep. Just before a machine is put to sleep, the Windows OS sends out a 'get ready for sleep" (a `Win32_PowerManagementEvent`) event to all the processes and drivers running on a machine, allowing them to prepare for sleep. The sleep notifier registers to receive this event. Upon receiving the event, the notifier immediately broadcasts a *sleep notification* packets (encapsulated in a UDP packet to port 9999), containing a "going-to-sleep" opcode and list of the sleep client's listening TCP ports, to the subnet broadcast address For reliability it retransmits the packet three times.

As the sleep notification packets are broadcast, the sleep client does not need to know the identity of the sleep proxy. The sleep notifier requires no configuration nor stable storage, as there is no state to be kept. The sleep notification packet obviates the need for active probing sleep clients to determine sleep status (as done in [28]) or which ports should be proxied ([28] restricted proxied ports to a manually pre-configured set).

Since the sleep notifier may have less than two seconds in which to send the sleep notification packet before the machine falls asleep [2], it is possible, albeit unlikely, that the notification packets will not be sent in time. Consequently, the sleep notifier also sends out periodic heartbeats when the machine is awake. These heartbeats are identical to the sleep notification packet, save that they use a "heartbeat" opcode. In our current implementation, heartbeats are sent out every 5 minutes, with some jittering. When the sleep proxy misses two consecutive heartbeats from a client, it immediately sends a WOL packet to that client. If, after sending the WOL, neither a heartbeat nor a sleep notification is subsequently received from the client, the proxy assumes that the machine has left the network and removes it from the list of proxied sleep clients.

## 4.3 The Sleep Proxy

The sleep proxy needs to monitor incoming traffic to the sleep client and also wake that client by sending a WOL packet on the subnet broadcast address [3]. Redirecting traffic destined for a given machine to another machine outside of its local subnet requires substantial support from routers. Thus, the sleep proxy has to run either on the subnet router itself, or on some other subnet machine. Running a sleep proxy on the subnet router was not possible, so we use one dedicated machine per subnet to act as a sleep proxy for machines in those subnets. Sec. 8 discusses how a peer-based version of our architecture could be implemented.

### 4.3.1 Rerouting

Like most enterprise networks, our network is a switched Ethernet network. Thus, unicast traffic for a host is not generally visible to other hosts on the network. Thus, upon receiving the sleep notification from a client, the sleep proxy needs to ensure that the traffic destined for sleeping clients is re-routed to the sleep proxy's NIC.

While there are a few ways to affect such re-routing, we have found sending *ARP probes* [13], as shown in Fig. 2, to be the most reliable method. A machine uses these ARP probes to advertise its MAC and IP address, and to perform duplicate address detection (DAD). Also, the subnet switches refresh/remap their internal routing tables upon receiving these probes.

| | Field | Value |
|---|---|---|
| Ethernet Header | Source Addr | $M.MAC\_Addr$ |
| | Destination Addr | FF:FF:FF:FF:FF |
| ARP Request | Sender MAC Addr | $M.MAC\_Addr$ |
| | Sender IP Addr | 0.0.0.0 |
| | Target MAC Addr | 00:00:00:00:00:00 |
| | Target IP Addr | $M.IP\_Addr$ |

Figure 2: ARP probe for sleep client $M$

[2]The sleep notifier cannot force the system to remain awake once the notification is broadcast

[3]This packet must be broadcast since at the time it is sent, the subnet's routing is set to deliver all packets meant for the sleeping host to the sleep proxy.

Thus, when a sleep proxy receives a sleep notification from a client, it issues specially crafted ARP probes *pretending to be the sleep client* (refer again to Fig. 2). This ensures that subsequent network traffic meant for the sleeping machine is delivered to the sleep proxy instead[4]

When a sleeping machine wakes (either because the sleep proxy woke it, or because it was woken for some other reason), it will naturally send out a fresh set of ARP probes generated by the OS to ensure that it can re-use the same IP address that it had before it went to sleep. This has two nice side effects. First, the subnet switches now begin forwarding traffic meant for the sleeping (and now awake) machine, back to that machine, instead of the sleep proxy. Second, as these probes are broadcast, the sleep proxy can see them, allowing it to immediately recognize when clients have woken and cease proxying for them.

### 4.3.2 Dealing with Incoming Traffic

As discussed earlier our sleep proxy responds mechanically to IP address resolution traffic, examines incoming TCP connection attempts, and ignores all other traffic.

Responding to IP address resolution traffic is easy: the sleep proxy simply issue ARP responses and Neighbor Discovery advertisements as if it were the sleeping client. Handling TCP connection attempts is a bit more complicated, as these may be encapsulated in a number of ways.

To detect an incoming TCP connection attempt the sleep proxy must examine the packet's IP header confirming it was destined to a currently proxied machine, and contains a TCP SYN with a destination port on which that machine had been listening. While it is easy to parse a TCP SYN contained in a vanilla IPv4 or IPv6 packet, our network (like most corporate networks) is more complicated in both its use of IPv6 tunneling and IPsec ESP authentication[5].

Tunneling comes in three flavors, ISATAP, 6over4, and Teredo [35]. Our current implementation handles ISATAP and 6over4. ISATAP packets are already unwrapped for the sleep proxy by the ISATAP router and arrive as IPv6 packets on the sleep client's ISATAP IPv6 address. Thus these packets require no additional processing. 6over4 packets arrive as IPv4 packets whose next protocol is 6over4. The inner packet is then removed and parsed as a standard IPv6 packet. Our current implementation does not handle Teredo wrapping, since it is being phased out in favor of the first two mechanisms.

The use of IPsec [41] presents a number of challenges. Imagine a remote machine $C$ trying to connect to sleeping machine $M$ using TCP. Let $S$ be the sleep proxy. If IPsec is in use, there are two possibilities. Either $C$ has not communicated with $M$ in recent past, or it has.

If $C$ has not recently communicated with $M$ it would first try to establish a new security association by doing IPsec key exchange (IKE). The IKE packets are sent via UDP. The IKE sent by $C$ end up at $S$. Recall, however, that our sleep proxy wakes up packets only on receiving TCP SYNs. Thus, the sleep proxy would never wake up $M$. However, Windows requires $C$ to send a TCP SYN "in clear"as it begins the

[4]An alternate way of doing this would be to replace $M.MAC\_ADDR$ with the sleep proxy's MAC address, however this could cause the DAD mechanism to be triggered if the sleep client were to wake very quickly after sleep.

[5]Note that tunneling and IPsec can be (and indeed are) used together. Our sleep proxy routinely sees and handles TCP SYNs that are encapsulated in an ESP payload, which is carried in an IPv6 packet, which is tunneled inside an IPv4 packet.

key exchange [41]. This is done to speed up the connection establishment: TCP handshake can happen in parallel with IPsec handshake. This works in our favor: the sleep proxy can detect the TCP SYN transmitted by $C$, and wake up $M$, which can then finish the key exchange. Otherwise, $M$ would need to be woken for every IKE attempt. As we shall see later, in our network this would have lead to many spurious wake-ups.

Conversely, if $C$ has recently communicated with $M$, it may have cached the security association information. Since our network uses Encapsulated Security Payload (ESP) [24] protocol, $C$ would encrypt the TCP SYN it sends. While the TCP SYN would end up at $S$, there is no way for $S$ to decode the packet. This would have been an unsolvable problem, except that our network uses ESP only with integrity service: the payload itself is not encrypted. Thus, $S$ can parse the packet [6] inspect it, and wake $M$ if needed.

The details of the IPsec setup are critical to the proper functioning of the sleep proxy. A stricter IPsec policy (e.g. one requiring encryption) would have either resulted in too many spurious wakeups, or made the deployment of a network-based proxy solution nearly impossible. Thus, when deploying network-based sleep proxies in corporate networks, the system administrator must be cognizant of the IPsec configuration.

## 4.4 Implementation Challenges

**When to send ARP probes:** The sleep proxy cannot simply send out ARP probes as soon as it receives the sleep notification from a client, as that client may send other packets before the network card sleeps. If ARP probes from the sleep proxy intermingle with traffic generated from the client that is about to fall asleep, the spanning tree protocol may end up in state where packets meant for the sleeping machine are not routed to the sleep proxy. In our early implementations, this problem created much heartache.

To avoid this problem, after receiving the sleep notification, the sleep proxy begins pinging that sleep client. The sleep proxy waits for five consecutive ping failures before sending out ARP probes and thereby taking over for the sleeping client.

**Daily wakeup & DHCP lease expiration:** Currently, the sleep proxy wakes all sleeping clients at 5AM. The primary reason is to allow these machines to initiate any backup or scanning activity. The wakeup also obviates the need for the sleep proxy to handle DHCP traffic on behalf of the clients. In our network, DHCP leases are valid for 30 days. When the client is awake, it renews the lease every day. Furthermore, it also renews the lease when it wakes up. As each client is guaranteed to wake up at least once a day, we did not need to implement DHCP renewal on our sleep proxy. The same mechanism also protects against address black-holing: whereby a sleep proxy keeps holding on to the address of a machine that has departed the network. If heartbeats are not seen for a sleep client after the daily wakeup, that machine is inferred to have left the network (as described earlier).

**Failure of sleep proxy:** In our current implementation, each subnet is served by a single sleep proxy. This creates a single point of failure. We have designed, but not yet implemented a primary-backup solution for ensuring additional reliability. Another possibility is to design a purely peer-to-peer solution (Sec. 8). Our design does offer protection against a sleep proxy crashing, and restarting. The sleep proxy stores the MAC addresses of all the machines that it is proxying for in a log maintained on non-volatile network storage.

[6] Although, even that requires use of heuristics.

Upon restarting, the sleep proxy checks the log, and proactively wakes up all the machines by sending them WOL packets. This ensures that the sleep proxy starts operations in a consistent state.

**Multi-homed machines:** The sleep proxy architecture can easily handle multi-homed machines as long as $(i)$ the sleep notification goes out on all interfaces and $(ii)$ a sleep proxy is available on each network that receives incoming connection attempts.

**Fate of connections that were open when machine goes to sleep:** In our architecture, TCP connections that were open when a client goes to sleep, are broken. We wake up the machine only on new connection requests. Intuitively, there are several reasons why it is safe to break existing TCP connections. First, the sleep proxy is not responsible for putting a machine to sleep. That decision is taken by the local OS. If the local OS deemed it safe to put a machine to sleep while it had TCP connections open, the sleep proxy should follow the lead. Second, most popular corporate desktop applications like email and web browser are inherently disconnection tolerant. Third, the applications that are not disconnection tolerant can always tell the OS to not sleep when they are running. Windows offers APIs to do this, and they are commonly used. For example, Windows Media Player will prevent sleep so long as it is playing content. See [28] for a more detailed rationale of this reaction policy. We do note that certain problems can arise with cloud-based applications under this policy, as will be discussed further in Sec. 8.

**Ignoring non-TCP traffic:** In our current implementation, the sleep proxy only wakes up the machine on incoming TCP SYNs, directed to open ports on the sleeping machine. This decision was taken based on analysis similar to that of [28]. In our network, most desktop applications use TCP. Users typically access their machines either via SMB (to retrieve files) or via Remote Desktop. Upon initiation, both these applications start new TCP connections, and hence send corresponding SYNs. Routine maintenance is handled via RPC calls, and this traffic also goes over TCP.

The impact of ignoring non-TCP traffic is difficult to estimate empirically ; although, the proof is in the pudding: after months of running our code, none of our users or IT staff have complained that their machines did not wake on remote access.

The TCP-only policy, would of course, be problematic for networks with very strict IPsec policies as discussed in Sec. 4.3.2 and UDP applications such as NFS. We don't use NFS in our network, so we have not investigated the particulars of supporting NFS, but our architecture is flexible enough to parse arbitrarily complex packets if necessary: including, potentially, stateful multi-packet parsing. Moreover our sleep proxy can easily replay packets causing wakeup, instead of just dropping them. TCP SYNs can be dropped since senders will retransmit.

**Manual wakeup:** Apart from the "automatic" wakeup described so far, we also provide for remote, manual wakeup of sleeping clients. This is achieved by maintaining a website outside our corporate firewall. Every sleep proxy maintains an open TCP connection to this web server. Users can type in the name of their machine on this website. The web service sends the name to every sleep proxy, and if a sleep proxy has the specified machine as a client, it wakes that machine up by sending it a magic packet. This service provides a "last resort" wakeup alternative. and also proved useful for addressing the cloud application problems mentioned above.

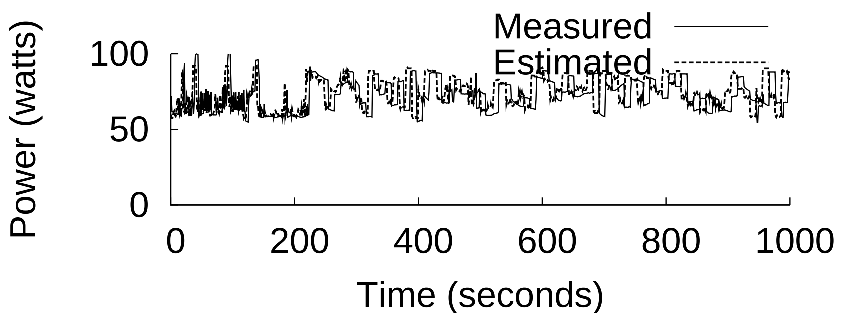


Figure 3: Measured and predicted power consumption

## 5 Instrumentation

Our sleep proxy keeps a detailed log of its interactions with clients, including when and why the clients go to sleep or wake up. On client side, we use *Joulemeter*, to estimate the power consumption of the clients, and gather information about why clients *stay* awake. Joulemeter is installed as a separate, optional service on clients.

### 5.1 Monitoring power consumption

To quantify the energy savings of our approach, we desired an accurate method of estimating our deployment's power consumption. Different machine makes and models consume power at differing rates. Further, a given machine consumes vastly different power depending on its CPU utilization level, P-state and whether its monitors are on or off. For instance, the power usage of an HP xw4300 workstation with two monitors varied from 141W to 240W with processor utilization, and changed by an additional 120W with monitor power state for a total variation of 2.5X.

However, desktop workstations do not typically have built-in instrumentation to measure power usage, and we wished to avoid attaching external power-meters to each machine for the same reasons we rejected hardware augmented sleep proxying approaches. Consequently, we used a software solution, *Joulemeter*, that produces power usage estimates based on hardware activity and pre-calibrated machine models.

The key principle behind Joulemeter's energy estimation is to use a machine specific power model. The model consists of a set of equations that relate the hardware configuration and resource utilization levels to power usage. Our current model takes into account processor P-states, processor utilization, disk I/O levels, and whether the monitor(s) are on or off. The power model for a specific hardware configuration is learned via *calibration* - controlled experiments in a laboratory settings. Once the power model is known, the machine's power consumption at run time can be estimated by monitoring CPU utilization (and P-state), disk utilization and monitor status. We omit the details of model construction due to lack of space. For a preliminary introduction see [23].

Fig. 3 shows Joulemeter estimates versus measured power consumption (using a hardware power meter) for a HP d530 workstation with 2.66GHz Pentium CPU running a workload generator that loaded the CPU and disk at random. The estimates were generated using the calibrated model produced from *a different* workstation with the same model and CPU. The results shown confirm Joulemeter's estimates track closely with the actual power consumption. In practice, no two systems are exactly alike. Still, we find that the Joulemeter predictions are accurate within 20%, for several machines for which we had built models.

In our deployment Joulemeter generated power readings were averaged over 30 second intervals and periodically uploaded to the database. We have built up a large library of power models covering most client machines in our deployment.

### 5.2 Monitoring machine insomnia

To determine why a machine is awake, Joulemeter relies on two sources. First, it periodically checks the *lastUserInput* timer provided by the OS. This timer provides the time of last user activity. We compare the value of this timer to the idle timeout (a typical Windows default value is 30 minutes). If user activity has occurred more recently than the idle timeout, we assume that the machine is being kept awake by user activity. We note that due to various technical issues this timer is not always available, so we cannot always determine whether the user is active.

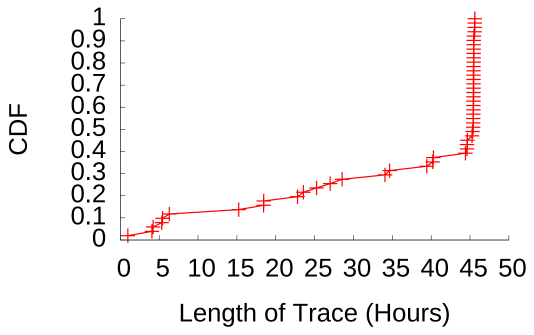
We also find that machines often stay awake even when the idle period exceeds this duration. To determine the reasons behind this, we rely on *powercfg.exe* utility that ships as part of Windows 7. The utility can often (but not always) shed light on why a machine is staying up by detailing *requests* to the OS for the machine to remain awake. For example, a remote machine may be holding a file open or a defragmenting routine may be running. Joulemeter periodically collects this information and reports it to the central database. Analysis of this information is presented in Sec. 7.

## 6 Implementation and Deployment

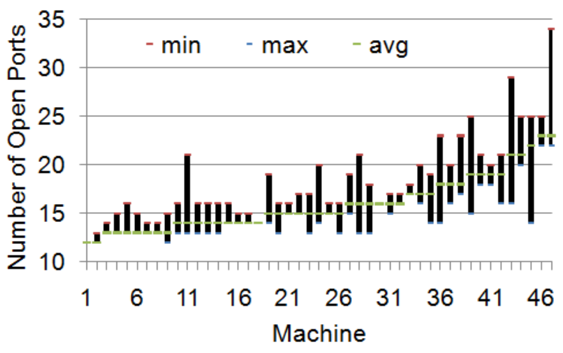
Our deployment consisted of 6 proxies (one for each of our network's 6 wired subnets), 51 clients, an SQL database, and the manual wakeup webservice mentioned earlier (standard IIS webserver with code written using ASP.NET). Most of the code is written in C# (5000  lines).

Only the sleep proxy contains any significant amount of unmanaged code. The sleep proxy relies on PCAP to capture and examine incoming packets. A small custom driver allows the sleep proxy to craft and inject ARP probes while bypassing the network stack. The primary data structure in the sleep proxy is a hashtable used to keep track of clients and their status. We first used ordinary desktop machines as proxies and have begun migrating to the low-powered, small-form-factor machines drawing less than 25 watts of power.

On client side, apart from the required sleep notifier service, the clients install three optional applications: Joulemeter, a GUI program displaying sleep statistics and estimated energy savings, and an auto-updater service that keeps client-side code up-to-date. During client installation, we ensured that Wake-On-LAN was enabled and ARP offload (which is enabled by default for certain cards in Windows 7) was disabled on the client's NIC. We also set the idle timeout to 30 minutes.



(a) CDF of Machine Trace Length



(b) Number of Listening Ports

Figure 4: Trace Length and Listening Port Distribution

## 7 Results

This section is guided by several overarching questions. What is the sleep and wake behavior of machines in our system? How much power did our solution save? What might be done to obtain additional power savings? What impact did our setup have on user experience? Was the sleep proxy architecture scalable? We begin by describing the details of our dataset.

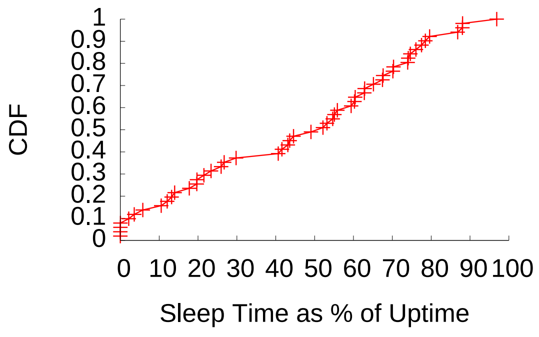
### 7.1 Dataset Overview

While our deployment has been active for half a year in various stages, for the rest of this section we focus on the 45 day period from November 19th, 2009 through January 3, 2010. During this time, we gathered data from 51 distinct machines belonging to 50 distinct users. As users installed our software at differing times, not all machines provided data for the entire period (although most did). Fig. 4(a) shows the cumulative distribution of trace lengths of individual machines. Our users were a self selecting group, so their behavior may not be representative of all user populations.

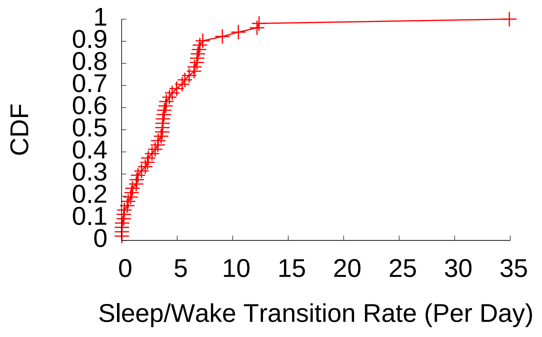
#### 7.1.1 Machines in our study

As we noted in Sec. 5.1, machine power consumption depends on the particulars of that machine's hardware configuration. The hardware configuration of machines in our deployment was varied, but not overly so. Of the 51 machines, 43 are HP and 6 were Dell. Only one of the machines has an AMD processor, the rest having Intel CPUs. Most of the machines are dual or quad cored. The CPU frequencies vary from 2-3.4GHz. Twenty seven machines had one monitor, 20 had two, and five had three. Five machines ran Windows Vista, all the rest ran Windows 7.

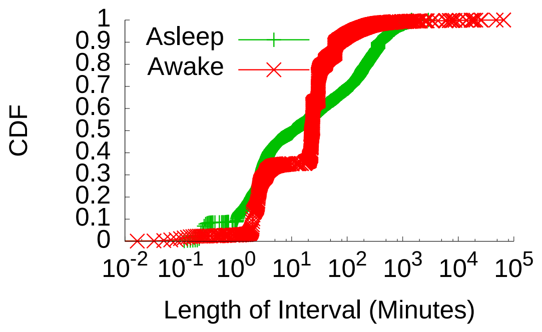
As we wake up machines for incoming TCP SYNs only on listening ports, it is worth examining the number of listening TCP ports on each machine. This number, of course, varies over time, as active processes and settings change. Fig. 4(b) shows the min, max, and average number of listening ports by machine. One machine had 35 ports open simultaneously!



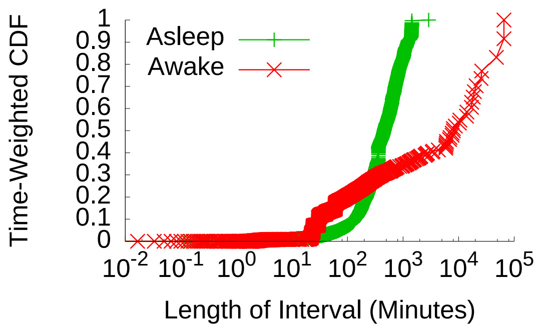
(a) Time Sleeping(%)



(b) Wake Transitions per Day



(c) CDF of Sleep/Awake Intervals



(d) Time-Weighted CDF

Figure 5: Aggregate Sleep/Wake statistics

#### 7.1.2 Traffic

Since all traffic destined for sleeping clients arrives at their sleep proxies, we can examine this traffic in centralized manner, without installing sniffers on individual machines. While we have deployed a sleep proxy on each of our six subnets, 59% of our machines are connected to the largest subnet. We have seen as many as 800 active machines on this subnet. We examined in detail a trace of all (23 million) packets arriving at the sleep proxy serving this subnet during a typical work week (5.5 days).

Of this traffic, 96% were multicast and broadcast packets. Of the multicast packets, 12.31% were ARP requests, which the sleep proxy examined and replied to as needed. The vast majority of the multicast traffic was safely ignorable [28]. The remaining 4% traffic was unicast: destined either to the proxy itself, or to the sleeping clients. 75% of these packets were wrapped by ESP and 8.4% were tunneled v6-over-v4 packets - underscoring the importance of parsing such packets. 7% of the total unicast packets were UDP (mostly IPsec related) and 3% were ICMP, which the sleep proxy ignores. Most of the remaining traffic was TCP, and the proxy was able to ignore the vast majority of it. During this time, we woke sleeping clients for just 747 TCP SYNs ($\leq$0.1% of UDP traffic). Our analysis of the traffic data confirmed the importance of filtering TCP SYNs based on port. More than half of incoming TCP connection attempts were destined to ports on which the sleep client was not listening. If we had woken clients without filtering by port, we would have had more spurious wake-ups than valid ones!

## 7.2 Sleep/Wake Behavior

We note that five of our 51 clients did not sleep at all, as their their users manually disabled sleep functionality.

### 7.2.1 Aggregate sleep/wake behavior

Fig. 5(a) shows the percentage of time each machine spent sleeping, as a CDF across all machines. The uniform slope of the CDF demonstrates that the average sleep time was quite variable, with 50% of the clients sleeping more than half the time. Fig. 5(b) plots the CDF of the average number of sleep-to-wake transitions per day for the machines. Most machines average fewer than seven daily wake-ups. Later, we will see that most of these wake-ups were caused by IT management traffic (e.g., updates) arriving for a sleeping machine.

We now examine the duration of sleep and awake intervals. Note that no sleep interval is longer than 1440 minutes because of the daily 5AM wakeup. The CDF of length of sleep and wake intervals is shown in Fig. 5(c), while Fig. 5(d) shows the time-weighted CDF (i.e., contribution of intervals at or below a given length to the total sleep or wake time). By comparing these two figures, we see that while most sleep and awake intervals are under one hour, the majority of both sleep and awake time comprises intervals over one hour. This implies that insomnia should be our first focus in attempting to reduce power usage (Sec. 7.3.2).

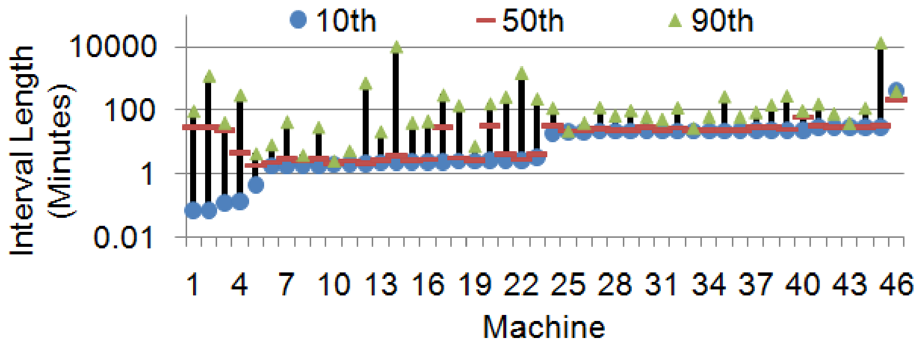
The awake interval CDF in Fig. 5(c) has abrupt changes in its slope at around two minutes, and at 30 minutes. This indicates that awake periods of two minutes and 30 minutes are prevalent in our trace, which we now investigate.

### 7.2.2 Individual sleep/wake behavior

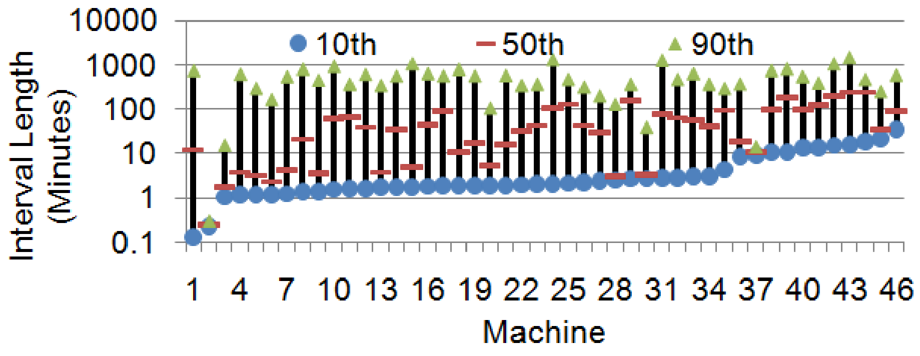
Figs. 6(b) and 6(a) show the 10th, 50th, and 90th percentile of wake and sleep intervals for each machine. The machines are sorted in order of 10th percentile. Notably, for around half of the machines the 10th percentile lies around two minutes, while for other half it lies around 30 minutes, corresponding to the jumps seen in Fig. 5(c).

We closely inspected a number of these awake periods. The prevalence of 30 minute awake periods is easily understood: this is the length of idle timeout. When a machine is woken for a quick, non-interactive task; the machine performs the task, and sleeps when the timeout expires. Yet, in such situations many machines fall asleep after just two or three minutes, despite the idle timeout being 30 minutes. We believe that this is due to some other internal timeout, set to more aggressively sleep machines woken for remote tasks. We do not know why this *remote access timeout* is active on only some machines and are investigating. When looking at our special 5AM wakeup (which we know was not user-initiated - Sec. 4.4) we saw a much greater than normal proportion of two minute wakes (on machines capable of such), adding additional weight to the hypothesis that there is a separate remote access timeout.

Fig. 6(b) shows that for about a quarter of the machines, the median sleep interval is under 10 minutes. For one machine all sleep intervals were under a minute. This machine appears to have some driver config-

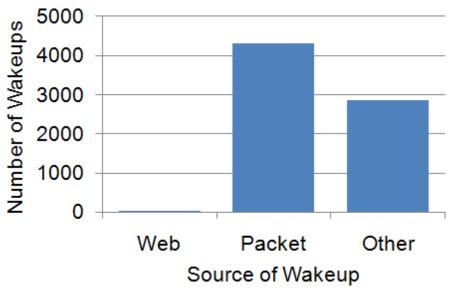


(a) Awake Interval

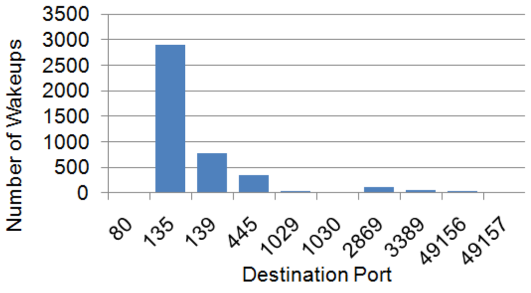


(b) Sleep Interval

Figure 6: Per-machine Sleep/Awake Intervals



(a) Cause of Wake



(b) Wakeup Source Port

Figure 7: Cause of wake-ups

uration issue that causes almost immediate wake upon sleep and was unique in our data set. Such intervals add very little to overall sleep duration and indicate potential sleep problems which will be examined further in Sec. 7.3.

#### 7.2.3 Why do machines wake up?

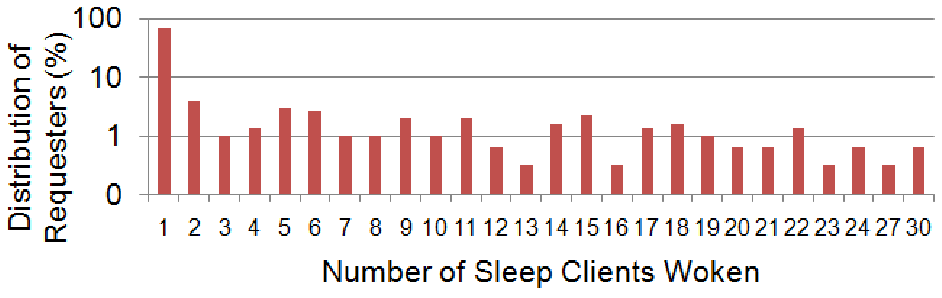
Fig. 7(a) shows the causes of wake-ups. We divide these into three categories: manual wake-up using our web site, wake-up by proxy due to incoming traffic, and other. The last bucket includes wake-ups caused by users walking up to the machine, any timer-based wake-ups caused by the BIOS, as well as occasional WOL packets sent by a commercial wakeup solution being tested by our IT department. We were able to confirm for 33% of these that the user did in fact initiate wakeup (by checking *lastUserInput* - Sec. 5.2) and for 50% of these the user definitively did not wake the machine. The remaining 27% could not be determined as *lastUserInput* was unavailable.

We see that while the web site was used in a few cases, it is not statistically significant. The majority of wake-ups caused by the sleep proxy are due to incoming TCP SYNs. The ports to which these SYNs were destined to are shown in Fig. 7(b).

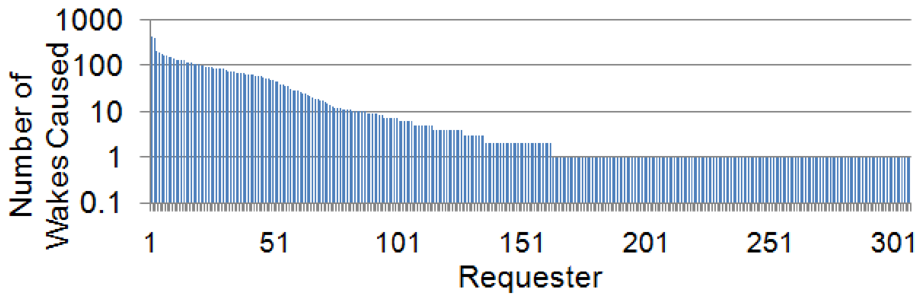
Remote Procedure Calls (port 135) were the overwhelmingly largest source of wakeup triggers, followed by NETBIOS (139) and SMB (445). SMB is the main mechanism used for remote file system access in our network. The two other notable ports are UPnP (2869) and Remote Desktop (3389). In our network, Remote Desktop is the primary mechanism for interactive remote machine access. We can see Remote Desktop is not a major wakeup source. In fact, only 39% of the machines were ever woken up due to Remote Desktop requests. Therefore, it would seem that while users leave their machines on for potential remote access, interactive remote access is used relatively rarely. This fact argues strongly for a sleep proxying approach such as ours.

#### 7.2.4 Who wakes up machines?

There were slightly over 300 IP addresses *requesters* whose incoming connection attempts caused wake-ups. Most of these only attempted to connect to a single sleep client. However, a sizable minority attempted



(a) Distribution of Requesters by Num. Clients Woken



(b) Number of Wake-ups Caused By Requester

Figure 8: Who causes wake-ups?

to connect to multiple clients as seen in Fig. 8(a). We were able to verify that all the requesters who woke 20 or more sleep clients were machines belonging to our IT department. These machines perform a variety of management actions such as verifying patch status and checking security policies. We will see later that our IT configuration is sleep-unfriendly in other ways as well (Sec. 7.3.2).

Fig. 8(b) shows the number of wakeup events caused by requester. Just as most requesters only connect to a single machine, many only cause only one wakeup and most cause only a handful. However, again a large minority of requesters cause many wake-ups each. IT-owned machines again make a large portion of this group. Interestingly, several of the most active requesters actually connect to only one or a handful of machines. In fact, the most active requester with over 400 requests connected to only two machines, and that too in in a span of just two weeks! We are currently investigating the role of this requester further.

## 7.3 Why Machines Don't Sleep Better

While we have seen that our solution is fairly successful at enabling machines to sleep (Fig. 5(a)), we wanted to investigate whether more idle time could be harvested. We begin by noting that most machines are not being woken overly often (Fig. 5(b)). However, a small subset of machines endure a large number of wake-ups, as discussed in Sec. 7.2.4. These machines are being bombarded by frequent connection attempts that interrupt their sleep often. If a machine with a standard 30 minute idle timeout wakes 12 times a day, one quarter of the day will have been spent awake due to wake-ups alone.

While this issue should be addressed, most sleep clients are being kept awake for other reasons the

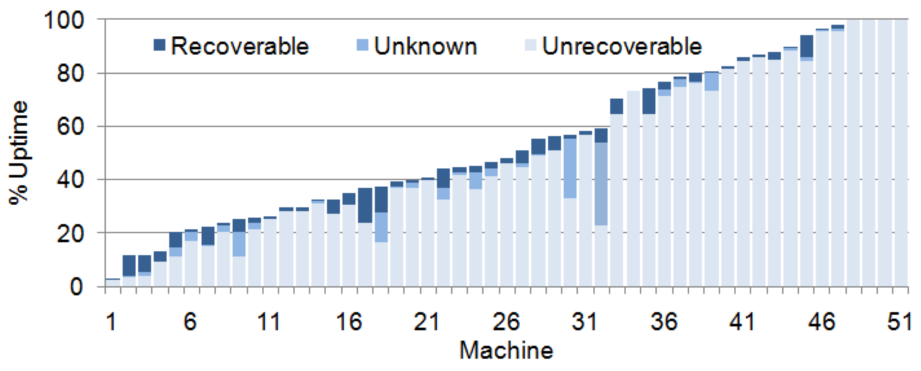


Figure 9: Awake Time as Percentage of Uptime. Broken Into Components Unknown, Recoverable, and Unrecoverable using Aggressive Idle Timeout

majority of the time. In fact, when not being kept awake, these machines manage to sleep well, sustaining few wakeup events per day. We now consider whether these machines would have benefited from a more aggressive idle timeout, and then look at the problem of insomniac machines.

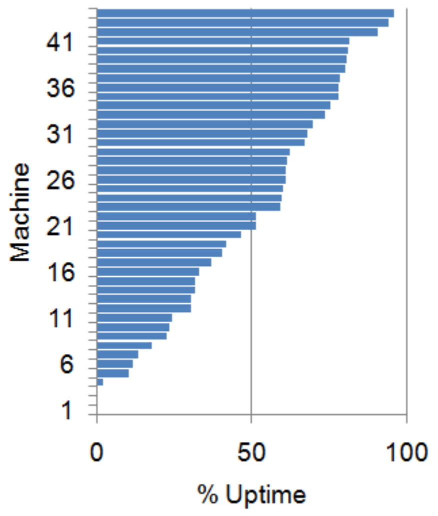
### 7.3.1 Aggressive idle timeout

As mentioned in Sec. 7.2.2, it appears that setting the idle timeout more aggressively could result in some power savings. We now consider how much could be saved with a 5-minute idle timeout (this is 1/3rd the EnergyStar guidelines recommendation [16]).

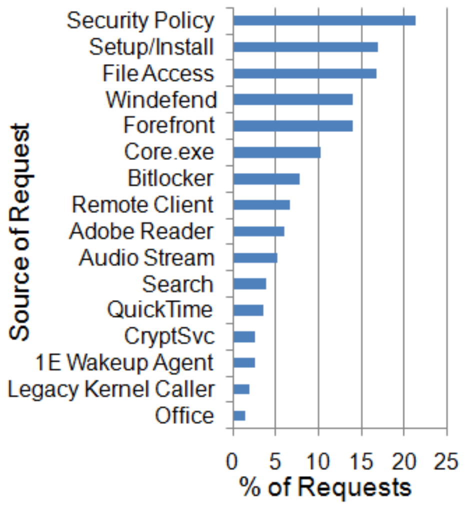
To do so, we examined each wake interval to see why the machine was being kept up. Recall from Sec. 5.2 that a machine may be kept awake because the user is active, the machine has woken up recently, or a *stay-awake* request placed by a local application with the OS.

We divided the total awake time into three components, *recoverable*, *unknown*, and *unrecoverable*. *Recoverable* time was time in which the machine could have slept if the idle timeout had been set more aggressively. This time was the sum of periods in which the user had been active within the past five minutes or the machine had been woken within the past five minutes. The *unknown* time was the time for which insufficient data was available to diagnose cause of wakefulness. The *unrecoverable* time consisted of all other time (i.e., an application had placed a *stay-awake* request with the OS).

Thus the recoverable time is a lower bound on the awake time that could have been saved by setting a more aggressive idle timeout. The sum of recoverable and unknown time provides the upper bound. Fig. 9 breaks the total wake time as percentage of uptime into these three components on a per-machine basis. We see that on most machines the impact would be relatively small. These machines are being kept up by local application stay-awake requests, to which we now turn.



(a) Percentage of Awake Time Caused by Requests



(b) Source of Requests

Figure 10: Stay-Awake Request Data

### 7.3.2 Insomnia

We now look more closely at which local applications keep machines awake. We label this phenomenon *insomnia*. Fig. 10(a) plots the fraction of awake time a given machine was kept awake by local applications requesting OS to prevent sleep. We see that the majority of awake time is in fact due to such stay-awake requests. So which applications cause these stay-awake requests? Fig. 10(b) shows the percentage of requests initiated by various applications. The news here is heartening. Four of the top sources (Security Policy Agent, Windefend, Forefront, and Bitlocker) are all applications mandated by our IT department. It may be possible to reconfigure or even re-write these applications to minimize and coordinate the duration of time they prevent sleep. At least three more (Flash, Quicktime and Audio Stream) are the result of code or driver bugs. For example, certain older versions of Flash player may keep a machine awake by playing silence even after the audio clip has finished (Windows prevents sleep when audio streams are active). The third-highest request source is SMB. SMB's default behavior prevents a machine whose files are being accessed from sleeping. Careful changes to this behavior may allow for greater sleep opportunities.

## 7.4 Power savings

Fig. 11(a) illustrates the lower bound on power savings on a per machine basis. This lower bound is calculated with the assumption that had the machine stayed up instead of sleeping, it would have consumed power at the lowest rate seen in the entire non-sleeping portion of the trace. The average across all machines is about 20%, although the variation is considerable.

Fig. 11(b), shows aggregate power consumption for a both a representative one-week period beginning

| Step | Time | From→To | Packet Type |
|---|---|---|---|
| 1 | 0 | M1→M2 | TCP SYN |
| 2 | 0.04 | S1→Broadcast | Magic Packet |
| 3 | 2.48 | M1→M2 | TCP SYN |
| 4 | 5.6 | M2→Broadcast | ARP Probe |
| 5 | 8.48 | M1→M2 | TCP SYN |
| 6 | 8.49 | M2→M1 | TCP SYN-ACK |

Table 1: Time line of a wakeup

12/3/09 and the winter break (beginning 12/24/09). During the representative week, weekend power consumption is low, spiking only at the 5AM wakeup. During the work-week, power use peaks during the work day before declining into an overnight trough and bottoms out early on Friday. In contrast we can see a markedly different pattern for the Mid-Winter week with almost no increase in activity during the day from the day preceding Christmas (which fell on Friday) through the following Monday. By the Tuesday following the holiday, we begin to see a similar level of activity to that of the representative week, albeit at a lower amplitude, as employees begin returning from the holiday. Interestingly, the power consumption over the Christmas weekend (12/26-12/27) weekend was slightly higher than during a normal weekend (12/5-12/6).

## 7.5 Micro-Benchmarks

We now validate our architectural approach by examining wakeup delay time and sleep proxy scalability.

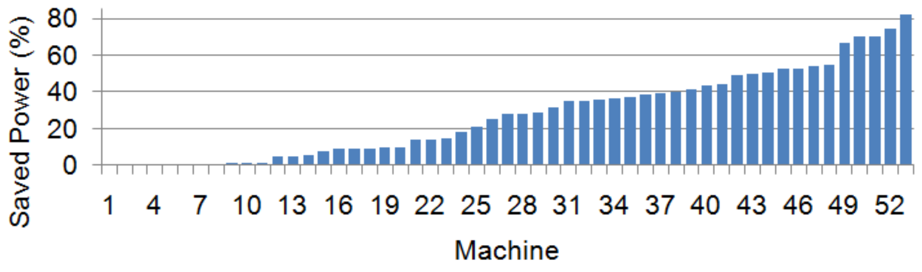
### 7.5.1 Wakeup delay

The energy saved by our system comes at a cost: the user experiences additional *startup* latency *the first time* a connection (e.g., ssh login or samba file access) to a sleep client is attempted since that client fell asleep. This happens because sleep client takes time to both wake and begin responding to an incoming TCP connection attempt. To make the system usable, we need to minimize the startup latency encountered by interactive transactions.

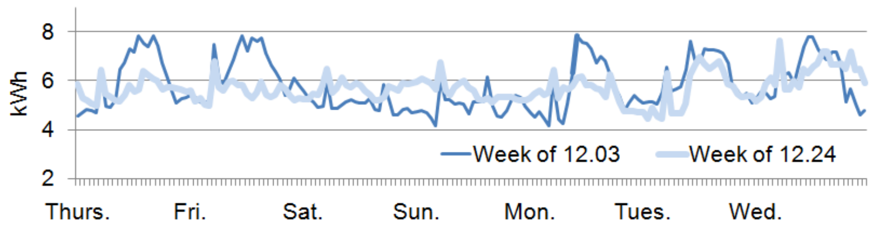
The user-perceived startup latency consists of several components: the delay involved in sending the WOL magic packet, the time required to wake up the machine, and the time required to perform any application-specific actions. To quantify these component latencies, we present a simple, but representative example.

Two machines, M1 and M2 were connected to the same subnet. M1 was ran a simple TCP sink, and was put to sleep. Thereafter, sleep proxy S1 started proxying for M1. From M2, we attempted to establish a TCP connection to the the sink on M1. The packet trace of the connection establishment is summarized in Table 1.

The total latency is about 8.5 seconds, but the sleep proxy itself consumes only 40 milliseconds, even though it is on a busy subnet and proxying for several other machines. The largest component is the *wake-up*



(a) Lower Bound on Per-Machine Power Savings



(b) Aggregate Power Draw for Normal vs. Mid-Winter Weeks

Figure 11: Power Draw and Savings

delay (i.e., time required for M2 to wake up and become active). This is roughly the delay between steps 2 and 4 (about 5.5 seconds). The remaining *TCP-retransmit* delay occurs between steps 4 and 5 (about 3 seconds). This delay is incurred while M1 waits to retransmit the TCP SYN the second time, following regular TCP timeout algorithm [31].

Specific applications will usually encounter slightly higher latencies, as the machine needs to perform additional, application-specific actions. For example, when M1 tried to list a directory on M2 via SMB, the transaction took 13.37 seconds when M2 was asleep. The additional delay was incurred while M2 reconnected with the domain controller, and obtained security credentials to determine whether to allow M1 access.

We stress that this delay is incurred only for the first transaction that wakes up the machine. Any subsequent transactions experience normal latencies. For example, subsequent file access from M1 to M2 saw no additional delays. While our experience is that users do not mind this one-off penalty, both the wake-up and retransmit delays can be addressed. A number of research and engineering efforts are underway to address the former. The latter can be shortened either by having M1 retransmit TCP SYN more aggressively, or having S1 “replay” the TCP SYN. We are currently investigating this second alternative.

### 7.5.2 Scalability

Our current deployment uses one sleep proxy per subnet. The load on these sleep proxies is a potential concern. We find that the CPU load on a sleep server rarely exceeds 5%. The total traffic (broadcast inclusive) seen by the sleep server is also quite low (90th percentile is 250Kbps). We conclude sleep proxy operations do not require substantial resources, and a single sleep proxy could easily handle very large subnets if necessary. Conversely for reasonably sized subnets, the sleep proxy could be located on a client machine without noticeably degrading the user experience (Sec. 8).

### 7.6 Summary

We have found our architecture works quite well - allowing users and IT remote access to sleep machines, while providing significant opportunity for sleep, and associated power savings. Our measurements indicate that remote access is used with moderate frequency by IT and relatively infrequently by end users. To glean more savings, we need a *sleep-aware* IT setup in which remote wake-ups, as well as local IT applications are scheduled more carefully to maximize sleep opportunities.

## 8 Conclusion & Future Work

We have designed and deployed a network-based sleep proxy in an enterprise network on over 50 user workstations. To our knowledge, this is the first such deployment of a sleep proxying system. During our work, we uncovered and addressed several practical issues that sleep proxying systems in the corporate environment must address. Our system has functioned both to user satisfaction and our own specification for the past several months. While our system has provided significant power savings, we find that significantly more power savings could be achieved by altering the IT setup. We conclude with a brief discussion of future possibilities and concerns.

**P2P Sleep Proxy:** Our current setup requires the use of a dedicated (albeit low-power) sleep proxy machine on each subnet. We are working on a p2p architecture in which machines fall asleep one after the other, while the "last man standing" keeps watch for the entire subnet.

**Security Concerns:** While the sleep proxying system does not pose a traditional security concern, we do note that many machines waking simultaneously could cause significant power spikes. To reduce the risk of this being exploited by an attacker, proxies can rate limit the number of WOL packets sent.

**Cloud-based applications:** Some of our users complained that our system interfered with the operation of two popular cloud-based applications LiveMesh and LiveSynch. Both these applications allow a user to keep designated directories on multiple machines in sync. The metadata for syncing is maintained on a cloud server. The machines are supposed to initiate a TCP connection to the cloud server, which the cloud server uses to inform them of any pending updates. The connection can either be periodic, or long-lived, but it must be initiated by the machine. Obviously, the machine has to be awake to do so. Our manual wakeup web-portal was designed in response to this. Whenever the user finishes making changes to a shared folder, she can wake up all machines that share this directory using the manual portal, and hope that the machines synchronize. This is very clearly a kludge: Incorporation of sleep proxy functionality into cloud applications would clearly be the ideal solution and warrants further exploration.

**Open Networks:** In a 24 hour study, we examined incoming traffic to a Columbia University machine with a dynamic IP address. We found no less than 10 spurious TCP connection attempts from as many scanning machines. This evidence indicates that in an open network, it may be wasteful to wake up a machine on every incoming TCP SYN to a listening port. Our sleep proxy can easily handle any black/whitelisting policy the administrator may specify to regulate who can wake up a machine.

## References

[1] Advanced Configuration and Power Interfae, revison 4.0. http://www.acpi.info/.

[2] Adaptive Technologies. http://www.adaptiva.com/.

[3] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: augmenting network interfaces to reduce pc energy usage. In *NSDI'09*, Berkeley, CA, USA, 2009.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008.

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

[6] Intel Active Management Technology (AMT). http://www.intel.com/technology/platform-technology/intel-amt/.

[7] Apple Wake On Lan. http://www.macworld.com/article/142468/2009/08/wake_on_demand.html.

[8] Microsoft application virtualization 4.5. http://www.microsoft.com/systemcenter/appv/default.mspx.

[9] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007.

[10] W. L. Bircher and L. K. John. Complete system power estimation: A trickle-down approach based on performance events. In *ISPASS*, 2007.

[11] J. Blackburn and K. Christensen. A simulation study of a new green bittorrent. In *Workshop on Green Communications*. IEEE Internation Conference on Communications, June 2009.

[12] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright. Power awareness in network design and routing. In *INFOCOM 2008*, 2008.

[13] S. Cheshire. IPv4 Address conflict detection. RFC 4227.

[14] K. Christensen, P. Gunaratne, B. Nordman, and A. George. The next frontier for communications networks: Power management. *Computer Communications*, 27(18):1758–1770, Dec. 2004.

[15] K. J. Christensen and F. B. Gulledge. Enabling power management for network-attached computers. *Int. J. Netw. Manag.*, 8(2):120–130, 1998.

[16] T.-T. Committee. www.ecma-international.org/publications/files/drafts/tc32-tg21-2009-150.doc. Technical report, ECMA, Nov. 2009.

[17] Worldwide pc market. http://www.tgdaily.com/slideshows/index.php?s=200903062p=1.

[18] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2007.

[19] Gumstix. www.gumstix.com.

[20] M. Gupta and S. Singh. Greening of the internet. In *SIGCOMM*, New York, NY, USA, 2003.

[21] Idc netbook and pc sales projections. http://www.tgdaily.com/slideshows/index.php?s=200903062p=1.

[22] M. Jimeno, K. Christensen, and B. Nordman. A network connection proxy to enable hosts to sleep and save energy. In *Performance Computing and Communications Conference*, pages 101–110. IEEE, Dec. 2008.

[23] A. Kansal and F. Zhao. Fine-grained energy profiling for power-aware application design. In *HotMetrics08*, June 2008.

[24] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC4301.

[25] P. Mahadevan, P. Sharma, S. Banerjee, and P. Ranganathan. Energy aware network operations. In *Global Internet Symposium*. IEEE, April 2009.

[26] M. Maisto. Global pc market suffering first decline since dot com crash, September 2009.

[27] Adding and Deleting Wake on LAN Patterns. http://msdn.microsoft.com/en-us/library/dd568089.aspx.

[28] S. Nedevschi, J. Chandrashekar, J. Liu, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: Reducing energy waste in networked systems. In *NSDI*, April 2009.

[29] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI'08*, Berkeley, CA, USA, 2008.

[30] 1E Technologies. http://www.1e.com.

[31] J. Postel. Tranmission control protocol. RFC 793.

[32] S. Rivoire, P. Ranganathan, and C. Kozyrakis. A comparison of high-level full-system power models. In *HotPower'08*, 2008.

[33] Intel Remote Wakeup Technology (RWT). http://www.intel.com/technology/chipset/remotewake-qa.htm?iid=Tech_remotewake+qa.

[34] A. Sinha and A. P. Chandrakasan. Energy efficient real-time scheduling. *Computer-Aided Design, International Conference on*, 0:458, 2001.

[35] Teredo tunneling. http://en.wikipedia.org/wiki/Teredo_tunneling.

[36] Tickless kernel. http://www.lesswatts.org/projects/tickless/.

[37] V. Valancius, N. Laoutaris, L. Massoulie, C. Diot, and P. Rodriguez. Greening the internet w/ nano data centers. In *CoNEXT*. ACM, 2009.

[38] Verdiem Technologies. http://www.verdiem.com/.

[39] D. Washburn. How much money are your idle pc wasting. Forrester Researech Reports, December 2008.

[40] C. A. Webber, J. A. Roberson, M. C. McWhinney, R. E. Brown, M. J. Pinckard, and J. F. Busch. After-hours power status of office equipment in the usa. *Energy*, Nov. 2006.

[41] How IPSec Works. http://technet.microsoft.com/en-us/library/cc759130.aspx.

[42] Wake-on-LAN. http://en.wikipedia.org/wiki/Wake-on-LAN.