Don't Lose Sleep Over Availability: The GreenUp Decentralized Wakeup Service

Siddhartha Sen[†], Jacob R. Lorch, Richard Hughes, Carlos Garcia Jurado Suarez,
Brian Zill, Weverton Cordeiro[‡], and Jitendra Padhye
Microsoft Research [†]*Princeton University* [‡]*Universidade Federal do Rio Grande do Sul*

Abstract

Large enterprises can save significant energy and money by putting idle desktop machines to sleep. Many systems that let desktops sleep and wake them on demand have been proposed, but enterprise IT departments refuse to deploy them because they require special hardware, disruptive virtualization technology, or dedicated per-subnet proxies, none of which are cost-effective. In response, we devised GreenUp, a minimal software-only system that allows any machine to act as a proxy for other sleeping machines in its subnet. To achieve this, GreenUp uses novel distributed techniques that spread load through randomization, efficiently synchronize state within a subnet, and maintain a minimum number of proxies despite the potential for correlated sleep times. In this paper, we present the details of GreenUp's design as well as a theoretical analysis demonstrating its correctness and efficiency, using empirically-derived models where appropriate. We also present results and lessons from a seven-month live deployment on over 100 machines; a larger deployment on ~1,100 machines is currently ongoing.

1 Introduction

A number of recent studies [2, 4, 18, 32] show that desktop computers in enterprise environments collectively waste a lot of energy by remaining on even when idle. By putting these machines to sleep, large enterprises stand to save millions of dollars [31].

The reasons machines stay awake when idle are well known [2, 23]. Most OSes put a desktop machine to sleep after some amount of user idle time, but users and IT administrators override this to enable remote access "at will." In enterprise environments, users typically access files or other resources from their machines, while IT administrators access machines for maintenance tasks. This scenario is prevalent in Microsoft and other large corporations [16]. Thus, any system for putting machines to sleep must maintain their availability.

A number of solutions have been proposed to solve this problem [2, 3, 8, 18]. However, many of the proposed solutions are difficult to deploy. For example, Somniloquy [2] requires specialized hardware, Litegreen [8] requires a fully virtualized desktop, and SleepServer [3] requires special application stubs.

The most promising approach, which requires no hardware changes and only minimal software deployment, is the "sleep proxy" approach, first proposed by Allman et al. [4] and further detailed by others [18, 23]. The key idea is that traffic meant for each sleeping machine is directed to a proxy. The proxy inspects the traffic, answers some such as ARP requests on behalf of the sleeping machine, and wakes the machine when it sees important traffic such as a TCP SYN. The proxy discards most of the traffic without any ill effects [18].

We previously deployed such an approach [23], but when we tried to start large-scale trials of it, we ran into major difficulties with our IT department that required us to completely redesign our solution. Basically, they refused to deploy and maintain a dedicated proxy on every subnet, due to the costs involved. Moreover, such a proxy constitutes a single point of failure, and would necessitate additional backup proxies, further adding to the cost. These opinions were also shared by the IT departments of some of Microsoft's large customers. We discuss this more in §2.

These considerations led us to a decentralized design for GreenUp, our system for providing availability to machines even as they sleep. The key idea behind GreenUp is that any machine can act as a sleep proxy for one or more sleeping machines on the same subnet. Whenever a machine falls asleep, another one starts acting as a sleep proxy for it. If the sleep proxy itself falls asleep, another sleep proxy rapidly takes over its duties.

Like most distributed systems, GreenUp must handle issues of coordination and availability among participants that operate independently. However, our problem setting has several distinctions that led us to an atypical design. First, machines are on the same subnet, so they can efficiently broadcast to each other and wake each other up. Second, GreenUp does not require a strongly-consistent view of the system. Third, GreenUp runs on end-user machines, making them sensitive to load and inherently unreliable. That is, their users will not tolerate noticeable performance degradation, and they may go to sleep at any time.

The techniques we have developed for this environment can be applied to any distributed system facing similar types of machine behavior and consistency requirements. *Distributed management* uses randomization to spread sleep proxy duties evenly over awake machines

without explicit coordination. *Subnet state coordination* uses the subnet broadcast channel shared by all participants to efficiently disseminate state among them. Additionally, we show how *guardians* can protect against a new type of correlated failure we observe: correlated sleep times among participants.

Our work makes the following contributions:

- We provide a complete design for a distributed system that makes sleeping machines highly available, including novel techniques for distributed management, subnet state coordination, and mitigation of correlated sleep with guardians (§4).
- We analytically justify our design techniques and prove that they achieve the properties we require. Where appropriate, we use models derived from real sleep behavior (§5).
- We demonstrate the feasibility, usability, and efficiency of our design by implementing it and deploying it on over 100 users' machines (§6), and discuss the many lessons we have learned from this deployment over the past seven months (§7).

After providing motivation in §2 and background useful for understanding our design in §3, §4–§7 detail our contributions. §8 discusses key issues and areas of ongoing work, §9 presents related work, and §10 concludes.

2 Motivation

In our earlier work [23], we discussed the engineering issues involved in deploying a sleep proxy solution. However, when we wanted to start large-scale trials of our system in collaboration with Microsoft's IT department (MSIT), we ran into difficulties that required us to completely re-engineer it.

MSIT was eager to deploy a solution that let idle desktops sleep while keeping them accessible. Currently, MSIT mandates and enforces a common sleep policy that achieves energy savings but reduces availability: user desktops automatically sleep after 30 minutes of inactivity. The goal is not just to save money, but also to contribute to Microsoft's overall environmental impact reduction initiative. If individual users wish to exclude their machines from this mandate, they have to request exceptions for each individual machine, and must provide a reason for seeking the exception. Based on an informal analysis of the reasons mentioned by the users seeking exceptions, MSIT decided that a solution that would automatically wake a sleeping desktop upon remote access would reduce exception requests.

Since deploying special hardware [2] or full virtualization [8] would be both too disruptive and too costly [23], these were not feasible, and only two options remained.

One option was to use the ARP proxy functionality offered by many modern NICs, combined with their ability to trigger wakeup when observing packets with specific bit patterns. However, this approach fails in the presence of complex encapsulation and IPSec encryption, both of which are commonly used in enterprise networks [23]. We also found that wake-on-pattern functionality in many modern NICs is unreliable.

The other possibility was to use a sleep proxy system [23], but this too had problems. One issue was deployment cost, due to its requirement of a dedicated machine on each subnet. Microsoft's corporate network consists of hundreds of subnets, so the cost of deploying and, more importantly, managing dedicated servers on each subnet would have been too high.

Worse yet, these servers would constitute a new single point of failure for each subnet, impacting the robustness and availability of the system. Our user studies showed that most users typically do not access their machines remotely, but when they do, the need is usually critical. For example, a salesperson may want to access some documents from his desktop while meeting with a client. Thus, each subnet would require additional backup servers and a system for monitoring liveness to achieve high availability, further raising the cost.

The same concerns were echoed by IT departments of some of Microsoft's largest customers [16]. As a result, a thorough re-thinking of the solution was needed. In particular, we decided to re-architect the system to make it completely distributed and serverless.

3 Background

This section provides a brief primer on Wake-on-LAN technology and on the original sleep proxy design [18, 23]. The discussion is not comprehensive; it only covers details useful in understanding our design.

3.1 Wake-on-LAN technology

Basic Wake-on-LAN (WoL) technology [5] has been available in Ethernet NICs for quite some time. To use it, a machine enters the S3 or S5 sleep state [11] while the NIC stays powered on. If the NIC receives a special *magic packet*, it wakes the machine. The magic packet is an Ethernet packet sent to the broadcast address whose payload contains 16 consecutive repetitions of the NIC's MAC address [5].

Because the packet is sent to the broadcast address, it is generally not possible to wake a machine on a different subnet. Some NICs do respond to WoL packets sent to unicast addresses, but this is not true for all NICs. Subnet directed broadcasts could wake machines in other subnets, but only with special support from routers. Thus, for robustness, our system relies only on basic WoL functionality and operates only within a single subnet.

3.2 Basic design of the sleep proxy

The core sleep proxy functionality of our system is similar to that described by Reich et al. [23] We summarize it via an example. Consider two machines S and P,

both on the same subnet. S will fall asleep, and P is the dedicated sleep proxy for the subnet.

When the operating system on S decides the machine should sleep, it sends a "prepare to sleep" notification to all running processes. A daemon running on S waits for this event, then broadcasts a packet announcing that S is about to sleep. The packet contains S's MAC and IP addresses, and a list of open TCP ports in the listen state.

When P receives this packet, it waits for S to fall asleep. Once pings indicate S has fallen asleep, P sends appropriately crafted ARP probes binding S's IP address to S's MAC address [23]. These trigger the underlying layer-2 spanning tree to be reprogrammed such that all packets meant for S are delivered to P instead. Notice that this is *not* address hijacking: it is *Ethernet port hijacking* [23], which is more general and robust against various DHCP issues.

P operates its NIC in promiscuous mode, and hence can observe packets meant for S. P ignores most traffic meant for S [18], with two exceptions.

First, P answers ARP requests and IPv6 neighbor-solicitation messages directed to S. This "ARP offload" functionality is needed to ensure other machines can direct packets to S's address [18].

Second, if P observes a TCP SYN directed to an open port on S, it wakes S by broadcasting a magic packet containing S's MAC address. The OS on S, upon waking up, will send its own ARP probes. These reprogram the layer-2 spanning tree so that all subsequent packets meant for S are correctly delivered to S instead of to P.

Meanwhile, the client that sent the original TCP SYN retransmits the SYN several times: 3 sec later, 6 sec after that, 12 sec after that, etc. Eventually, a retransmitted SYN reaches S, which is now awake and can complete the connection in a normal manner. The user experiences a small delay since the first SYN is not responded to [23], but otherwise the user experience is seamless.

4 GreenUp Design

The primary goal of GreenUp is to ensure that all participants remain highly available while allowing them to sleep as much as possible. A secondary goal is to rely only on participant machines, not dedicated servers, to aid availability and ease deployment. Designing such a system presents unique opportunities and challenges:

(i) Subnet domains. Each GreenUp instance runs within a subnet domain, providing an efficient broadcast channel for communicating and for waking machines up.

(ii) Availability over consistency. The availability of machines is more important than the consistency of system state or operations. In particular, strong consistency can be sacrificed in favor of simplicity or efficiency.

(iii) Load-sensitive, unreliable machines. Running on end-user machines requires us to limit the resources we use, *e.g.* CPU cycles, since users will not tolerate noticeable performance degradation. Also, these machines may go to sleep at any time, and they may exhibit correlated behavior, *e.g.* sleeping at similar times.

GreenUp's design exploits these opportunities and meets these challenges, using novel techniques that apply to a broader class of distributed systems. GreenUp is suitable for enterprises where desktop machines provide services to remote users. This is the case for Microsoft and some of its large customers, as we have discussed. In organizations where desktops are used only as terminals, or where users only use mobile laptops, GreenUp is not appropriate. In §8, we discuss possible extensions of GreenUp for certain server cluster settings.

4.1 Overview

To aid the presentation of our design, we introduce the following terminology. We call a machine with GreenUp installed a *participant*. When a participant is capable of intercepting traffic to sleeping machines, we call it a *proxy*. Generally, every participant is a proxy as long as it is awake. When a proxy is actively intercepting traffic for one or more sleeping machines, we say it is *managing* them: it is their *manager* and they are its *managees*. To manage a participant, a manager needs to know the managee's *state*: its IP address, its MAC address, and its list of open TCP ports.

The key idea for maintaining availability in GreenUp is to enable any proxy to manage any sleeping participant on the same subnet. When a participant goes to sleep, some proxy rapidly detects this and starts managing it. For this, we use a scheme called *distributed management* that spreads management load evenly over awake proxies by using randomized probing (§4.2). To detect and manage asleep machines, each proxy must know each participant's state, so we distribute it using a scheme based on subnet broadcast called *subnet state coordination* (§4.3). This scheme tolerates unreliable machines and provides eventual consistency for participants' views of state. Distributed management will stop working if all proxies go to sleep, a condition we call *apocalypse*. Using real trace data, we demonstrate that such correlated sleep behavior is plausible, and consequently show how to use *guardians* (§4.4) to always maintain a minimum number of awake proxies.

Figure 1 shows an example illustration of these techniques from our live deployment. The coming sections explain them in greater detail.

4.2 Distributed management

In this section, we discuss our scheme for *distributed management*. The techniques we develop here can be used by any distributed system to coordinate a global task using local methods, provided the system can tolerate occasional conflicts. In our scheme, proxies do not explic-

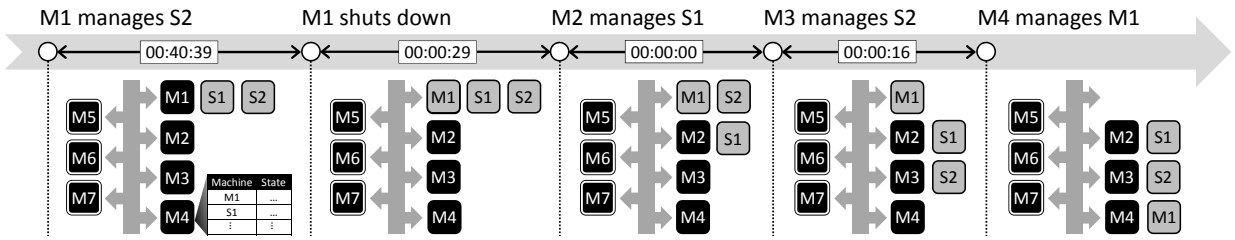

Figure 1: Live operation of GreenUp. Awake proxies are colored black, guardians have an additional white border, and asleep or shut down machines are gray. Participants use subnet state coordination to exchange state, shown in the inset of M4. M1 starts shutting down at 11:18 am on July 11; its abandoned managees S1 and S2 are managed again 29 seconds later via distributed management. M1 is managed 16 seconds after that, since it required time to shut down. Guardians M5, M6, and M7 stay awake to prevent apocalypse.

itly coordinate, yet they ensure a sleeping participant is rapidly detected and managed (§4.2.1). In the event that multiple proxies accidentally manage the same participant, they quickly detect this and all but one back off (§4.2.2). Due to our distributed setting, our managers operate differently (§4.2.3) from the sleep proxy described in §3.2.

4.2.1 Distributed probing

To maintain the availability of sleeping machines, we need to ensure that asleep participants are always managed. It might seem that we could rely on participants to broadcast a notification when they are about to sleep, but this is unreliable. The notification may not be seen if there is network loss, if the participant fails, or if the participant is already asleep and its current manager fails. Thus, proxies must continually monitor participants to discover when they become both asleep and unmanaged.

As we will see in §7.2, monitoring every participant on a large subnet can cause noticeable load. Because of the load sensitivity challenge, we distribute the monitoring work across all proxies via *distributed probing*, as follows. Each proxy periodically *probes* a random subset of participants. If a participant is awake, it responds to the probe; otherwise, its manager responds on its behalf. If the proxy receives no response, it starts managing the machine itself. Note that assigning the management task to the proxy that first detects the sleeping machine effectively distributes the management load across all proxies, as we prove in §5.2.

A proxy *probes* a participant by sending it two packets: a TCP SYN to a GreenUp-specific port, and an ICMP echo request or *ping*. If the machine is awake, its operating system will send an ICMP echo reply. If the machine is managed, its manager will be watching for TCP SYNs to it, and will respond to the probe with a UDP message indicating the machine is managed. If the proxy hears neither response, it resends the probe packets every second until a sufficiently long period passes without a response, at which point it concludes the participant is asleep and unmanaged. In our current implementation, we use a timeout of 25 seconds, which we consider a reasonable amount of time to ascertain a machine is asleep.

Our goal is to start managing participants quickly once they or their current managers fall asleep. This means we cannot let a participant go very long without some proxy probing it, necessitating a periodic rather than exponential-backoff scheme. Thus, each proxy uses the following random probing schedule to ensure each participant is probed once every s seconds with probability p. Given $\mathcal{S}$, the set of participants it is not managing, and $\mathcal{K}$, the set of proxies, it probes a random subset of $\mathcal{S}$ of size $-\ln(1-p)\,|\mathcal{S}|/|\mathcal{K}|$ every s seconds. We prove in §5.1 that this quantity achieves our goal. In our current implementation, we use $p = 90\%$ and $s = 3$, so an asleep unmanaged machine is probed within 3 seconds with probability 90% and 6 seconds with probability 99%. These parameters limit the additional delay beyond 25 seconds that we must wait to ensure a machine is asleep.

4.2.2 Resolving multiple management

Distributing responsibility for a service is useful for simplifying coordination and spreading load, but it can lead to conflict. It is best suited for distributed systems, like ours, in which limited conflicts are acceptable as long as they are eventually resolved. We now discuss our approach to resolving conflicts arising from distributed probing.

Conflict arises when distributed probing leads to more than one proxy simultaneously managing the same sleeping machine S. This is acceptable, since multiple managers do not interfere in their fundamental duty of waking up S, and reflects our decision to err in favor of ensuring availability rather than consistency. However, it is wasteful of network bandwidth and causes churn in the switches' spanning tree: each manager will reply to ARP requests for S, causing the switch to relearn the port each time. Thus, we take the following steps to eventually detect and resolve multiple management.

We define the *priority* of a proxy P to manage a sleeping machine S as the hash of the concatenation of P and S's MAC addresses. In what follows, assume S is the

managee currently managed by M, H is a manager with higher priority, and L is a manager with lower priority.

1. If M hears a message from H indicating it is managing S, M stops managing S and sends an ARP request for S's IP address. The purpose of the ARP request is to trigger H to send an ARP response, thereby taking control of S's port away from M.
2. If M hears a message from L indicating it is managing S, it sends a message to L saying S is managed. This causes L to perform step 1.

We will see in §4.3 that each manager broadcasts a message upon managing a machine, and periodically rebroadcasts it. Therefore, it will eventually trigger one of the conditions above.

Other distributed systems may choose different approaches for conflict resolution, such as exponential backoff. We use priority assignment because during conflict it maintains multiple servicers rather than zero. Specifically, while two or more proxies are deciding which among them should manage a sleeping machine, it remains managed by at least one of them.

4.2.3 Manager duties

When a proxy P manages a sleeping participant S, it acts largely the same as the sleep proxy described in §3.2. For instance, P uses Ethernet port hijacking to monitor traffic to S, maintains S's network presence by responding to ARP requests, and wakes S if it sees a TCP SYN to an open port. However, there are a few notable differences arising from the distributed nature of GreenUp.

First, P monitors not only incoming SYNs destined for S but also *outbound* SYNs destined for S. This is because, unlike the dedicated sleep proxy [23], P is an end-user machine whose user may actually be trying to connect to a service on S. Since P has hijacked S's port, any traffic it sends to S will be dropped by the switch—not sent back to P—to avoid routing loops. Thus, the only way for P to detect a local attempt to connect to S is by monitoring its own outgoing traffic.

Second, as discussed in §4.2.1, P must respond to SYNs for S on the special probing port by saying S is managed. Otherwise, the prober will conclude S is unmanaged and start managing it.

4.3 Subnet state coordination

The distributed probing and management algorithm in §4.2 assumes that each proxy knows the state of each participant in the subnet. Thus, we need participant state to be reliably distributed to all proxies despite our use of unreliable, load-sensitive machines with correlated behavior. In this subsection, we describe our approach to this called *subnet state coordination*. This technique is applicable to any distributed system that needs to disseminate state among a set of machines on the same subnet, but does not require strong consistency for this state.

We chose not to use a fault-tolerant replicated service [6, 14, 25], since a correlated failure such as a power outage or switch failure could subvert its assumption of a fixed fault threshold, rendering it unavailable. Also, the provision of strong consistency is overkill since small differences in proxies' views of the global state do not cause problems. For our system, eventual consistency is sufficient.

Unlike gossip [9, 10], subnet state coordination exploits the fact that all participants are on the same subnet. Thus, (1) they have an efficient broadcast channel, (2) they can wake each other up using the scheme in §3.2, and (3) they are all loosely time-synchronized (*e.g.*, via NTP). Our technique consists of three elements:

1. **Periodic broadcast.** Each participant periodically broadcasts its state, using a period equal to the maximum desirable staleness. It also broadcasts its state whenever it changes.
2. **Rebroadcast by managers.** When a participant is asleep, its manager is responsible for its broadcasts.
3. **Roll call.** Periodically, there is a window during which all asleep participants are awakened. Each proxy drops from its participant list any that do not broadcast during this window.

Periodic broadcast takes advantage of the efficient broadcast channel, and would be sufficient if participants were perfectly reliable. However, since they run on machines that can sleep at any time, rebroadcast by managers is required to ensure that state about sleeping participants is also disseminated. Because a machine that has failed or is no longer participating in GreenUp is hard to distinguish from a sleeping participant, the roll call is used to identify the current set of participants. It ensures that each proxy eventually removes the state of a former participant, and thus stops trying to manage it; as we discuss in our technical report [26], it is acceptable for this to happen eventually rather than immediately. For durability, proxies store their global view of state on disk.

We call the periodic broadcasts of state *heartbeats*. When a participant sends a heartbeat about itself, we call this a *direct heartbeat*. When a participant sends a heartbeat about another participant, we call this an *indirect heartbeat*. Each direct heartbeat includes a timestamp, and each indirect heartbeat includes the timestamp of the last direct heartbeat on which it is based. This way, if a proxy knows more recent state about a participant than the participant's manager, it can inform the manager.

As a side effect of receiving heartbeats, a proxy learns which participants are awake and which are managed. It uses this to compute the sets $\mathcal{S}$ and $\mathcal{K}$ for distributed probing, and also for apocalypse prevention below.

For the roll call, we use a window of ten minutes each day, which is long enough to account for clock skew. Since one of our goals is to let machines sleep as much

as possible, we choose a roll call window when most machines are awake anyway, which in our organization is 2:00 pm–2:10 pm.

In environments where there is already a highly-available means for sharing state, like Bigtable [7] or HDFS [27], that system could be used instead of subnet state coordination. However, by using subnet state coordination, our system becomes instantly deployable in any environment with no external dependency.

4.4 Preventing apocalypse

While allowing proxies to sleep increases energy savings, we cannot allow too many to sleep, for two reasons. First, proxies may become overloaded, due to the demands of distributed probing and management. Second, we may reach a state where no proxy is awake, *i.e.*, *apocalypse*. This is a disastrous scenario because users who participate in GreenUp have no way of contacting their machines during apocalypse.

Initially, we thought we could rely on the natural behavior of proxies to prevent apocalypse, and only force them to stay awake when their number reached a dangerously low threshold. However, this approach only works if machines sleep independently, which we show is untrue (§4.4.1). Thus, we develop a technique that maintains a minimum number of awake proxies at all times (§4.4.2). This technique can be used by any distributed system to prevent system failures caused by correlated sleep behavior.

4.4.1 Non-independent sleep behavior

Since GreenUp is targeted for corporate deployment, we require apocalypse to occur with extremely low probability. Such a low probability can only be derived if machines go to sleep independently, so their probabilities multiply. That is, if each awake proxy $i \in \mathcal{K}$ goes to sleep with probability p_i, then we would like to say that the probability all $|\mathcal{K}|$ machines sleep is $\prod_{i \in \mathcal{K}} p_i$, which decreases exponentially.

Unfortunately, we were unable to confirm independent sleep behavior from our data. For this analysis, we used data provided to us by Reich et al. [23] tracing the sleep behavior of 51 distinct machines over a 45-day period ending 1/3/2010. We measured the pairwise independence of every pair of machines A, B with at least 100 samples each, by computing the distribution of: (1) the time to the next sleep event of B from a sleep event of A; and (2) the time to the next sleep event of B from a random time. Given sufficient samples, these distributions should be statistically similar if A and B sleep independently. We measured statistical distance using the two-sample Kolmogorov-Smirnov (K-S) test [13, p. 45–56]. At significance level $\alpha = 0.05$, we found that over 61% of the 523 pairs of machines tested failed. We conclude that we cannot rely on the natural behavior of machines to prevent apocalypse, and instead opt for a design that always keeps some machines awake.

4.4.2 Guardians

Since proxies exhibit unreliable, correlated sleep behavior, our approach to prevent apocalypse is to maintain a minimum number of *guardians* at all times, *i.e.*, proxies that prevent themselves from automatically sleeping. They do so by making a special Windows request indicating the machine should not sleep. Maintaining guardians serves a similar purpose to deploying a static set of proxy servers, but with the advantages of zero deployment cost and resilience to failures of particular machines.

Since even guardians can become unavailable, *e.g.*, due to users hitting power buttons, we require multiple guardians to ensure high availability. The number of guardians we use is described by a function $q(n) = \max\{Q, \frac{n}{B}\}$, where n is the number of participants and Q and B are constants. The purpose of Q is to make unlikely the situation that all guardians simultaneously fail. The purpose of B is to limit the probing and management load of each proxy. We discuss our choice of $Q = 3$ in §7.3 and our choice of $B = 100$ in §7.2.

For the sake of load distribution and to avoid relying on individual machines, proxies dynamically choose the current set of $q(n)$ guardians. They use the same deterministic algorithm for this, and thus independently reach the same decision. Specifically, each proxy computes its position in an ordering of the proxies $\mathcal{K}$, by hashing the MAC address of each proxy with the current date. A proxy becomes a guardian if and only if it is in one of the $q(n)$ highest positions. If a proxy ever notices that there are only $q(n) - c$ proxies, it wakes the c asleep participants with the highest positions. If any of these participants does not wake up within a reasonable period, it wakes the next-highest asleep participant, and so on.

The above approach only fails when all guardians stop so close together in time that they cannot detect this, such as during a power outage. Fortunately, when power is restored, the proxies with Wake-on-Power enabled in their BIOS will come back on and use their durably-stored state to bring GreenUp out of apocalypse. Unfortunately, Windows currently does not support programmatically enabling Wake-on-Power, and machines that do not Wake-on-Power and are unable to Wake-on-LAN from an off state cannot be woken after a power outage. Thus, we plan to programmatically enable Wake-on-Power when it becomes available, and also plan to advise users to enable it manually for the time being.

5 Analysis

This section analyzes certain stability, correctness, and efficiency properties of GreenUp. In doing so, we analytically justify several of GreenUp's design decisions, including its key parameter choices. We guide our anal-

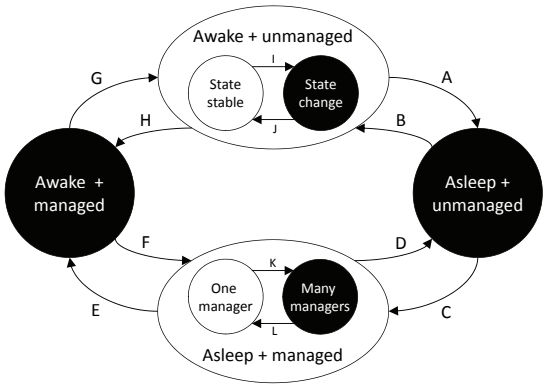

Figure 2: The lifecycle of a machine in GreenUp. Stable states are shown in white.

ysis by modeling the lifecycle of a participant as a state machine with stable and unstable states, shown in Figure 2. Our system maintains the invariant that it converges quickly and reliably to stable states.

5.1 Probe frequency and amount

Let $\mathcal{K}$ be the set of proxies. Every s seconds, each proxy $i \in \mathcal{K}$ sends b_i probes uniformly at random to $n - m_i$ participants, where m_i is the number of participants managed by i, including itself. Declare a *round* of probing to be complete when all $\sum_{i\in\mathcal{K}} b_i$ probes have been sent. We only care about a participant being probed if it is asleep and unmanaged, as otherwise it does not require probing.

Lemma 1. *If $b_i = -(n - m_i)\ln(1 - p)/|\mathcal{K}|$ for $0 < p < 1$, then the probability that an asleep, unmanaged participant remains unprobed after a round is at most $1 - p$, and the expected number of unprobed participants is concentrated about its mean.*

Proof. Let Z_j be the indicator random variable for participant j remaining unprobed after a round. By assumption, no proxy is j's manager, so all proxies probe j:

$$\begin{aligned}\Pr(Z_j = 1) &= \prod_{i\in\mathcal{K}} \left(1 - \frac{1}{n - m_i}\right)^{\frac{-(n-m_i)\ln(1-p)}{|\mathcal{K}|}} \\ &\leq \prod_{i\in\mathcal{K}} e^{\frac{\ln(1-p)}{|\mathcal{K}|}} \\ &= \left((1-p)^{\frac{1}{|\mathcal{K}|}}\right)^{|\mathcal{K}|} = 1 - p\end{aligned}$$

To show concentration about the mean $\mathrm{E}[Z] = \sum_j \mathrm{E}[Z_j] \leq n(1-p)$, we apply Azuma's inequality [17, p. 92] to an appropriate martingale sequence. □

Moreover, the probability a participant is unprobed decreases by $(1-p)$ each round, *i.e.*, exponentially. We set $p = 90\%$ in our implementation, ensuring that transition C in Figure 2 is fast and reliable.

5.2 Management load

We can view participants falling asleep as a balls-in-bins process [21]. Initially, there are n awake proxies and each has a "ball". When proxy i goes to sleep, i and its managees are eventually managed by random proxies due to distributed probing: in effect, i throws m_i balls randomly over the awake proxies. When i wakes up, it stops being managed: in effect, i retrieves its ball.

Lemma 2. *After t sleep events, the distribution of managees on the $n - t$ awake proxies is identical to that of throwing t balls into $n-t$ bins uniformly at random [21].*

Proof. Let $\mathcal{X}_k$ be the set of proxies whose balls are thrown during the k^{th} sleep event, for $1 \leq k \leq t$. We characterize the probability of proxy i's ball landing in proxy j. If $i \in \mathcal{X}_t$, then the probability is $P_t = 1/(n-t)$ since this is the last sleep event. For $1 \leq k \leq t-1$, observing that each proxy's ball is (re)thrown independently of any other ball, the probability is:

$$\begin{aligned}P_k &= \Pr[i \in \mathcal{X}_k \text{ throws ball in } j] \\ &= \left(\frac{n-t}{n-k}\right)\left(\frac{1}{n-t}\right) + \left(\frac{t-k}{n-k}\right)\sum_{j=k+1}^{t}\frac{P_j}{t-k}\end{aligned}$$

since either i lands in an awake proxy (one of which is j), or i lands in one of $t-k$ proxies that subsequently sleep. Since $P_t = 1/(n-t)$, this recurrence gives $P_k = 1/(n-t)$ for all $1 \leq k \leq t$.

□

Lemma 2 implies that transitions C and D preserve load balance when a proxy goes to sleep. We now describe two optimizations to these transitions that have not yet been implemented. First, if proxies awaken at random, we can preserve load balance by giving an awakened proxy half of its previous manager's managees. Second, when a proxy is about to sleep, we can avoid the period during which its managees are unmanaged by explicitly handing them off to another proxy chosen at random. The resulting process is closely related to the "coalescent" process in population genetics [12] used to trace the origins of a gene's alleles. We have studied the load properties of the process and have proved the following result, to appear in a forthcoming report:

Theorem 1. *After t random sleep events using the above hand-off policy, the expected maximum number of managees on an awake proxy is less than H_{n-t} times the optimal $n/(n-t)$, where H_i are the harmonic numbers.*

5.3 Other transitions and cycles

Our apocalypse prevention scheme in §4.4.2 ensures that transition C is reliable. The remaining transitions in Figure 2 are easy to argue. Transition J is fast because participants broadcast their state whenever it changes.

The multiple management protocol ensures transition L occurs and prevents transition D by using a deterministic hash on proxy priorities. Cycles (A,B) and (E,F) are unlikely: data from our deployment shows that less than 0.88% of sleep intervals are shorter than 30 seconds (enough time to determine a machine is asleep), and less than 0.68% of awake intervals are less than 10 seconds (plenty of time for a broadcast). Cycles (G,H) and (K,L) are indicative of connectivity issues, which we address in §6. Cycle (A,C,E,G) is rare because views are kept consistent by subnet state coordination and managers only awaken managees for legitimate requests.

6 Implementation

We implemented GreenUp in C#, using four main modules: (1) a state collector, which implements subnet state coordination; (2) a prober, which implements distributed probing; (3) a proxy, which implements traffic interception by interfacing with a packet sniffer; and (4) a logger, which collects debugging information and periodically uploads the logs to a central location. The current implementation of GreenUp is 9,578 lines of code according to SLOCCount [33]. It is packaged with a client UI, not discussed in this paper, that presents information to the user like how often her machine sleeps.

TCP SYNs in probes. Our initial implementation used UDP packets instead of TCP SYNs when probing. However, UDP packets are often encrypted with IPSec using key material shared between the prober and probe target. If the probe target is managed, the manager intercepting its traffic would find these packets undecipherable and would not be able to respond, leading the prober to incorrectly conclude that the target is unmanaged.

To fix this problem, we switched to TCP SYNs, which are not encrypted with IPSec in our network [23]. This solution has an important advantage: When a manager responds to a probe, it demonstrates not only its ability to successfully intercept traffic, but specifically its ability to intercept and recognize TCP SYNs, which is required to wake up a managee on actual connection attempts. This makes our protocol highly robust: If a proxy is not doing a good job of managing a sleeping node, another proxy will detect this and take its place.

Fail-safes. Since GreenUp runs on end-user machines, we implemented some fail-safes for added protection. First, we monitor the moving average of the GreenUp process's CPU utilization and, if it reaches a certain threshold (*e.g.*, 20%), we disable the proxy functionality of that machine temporarily. Second, proxies monitor their suitability for management by making sure they receive regular probes for managees and ensuring connectivity to the subnet's default gateway. If a proxy decides it is unsuitable to manage machines, it stops doing so; as with any type of proxy failure, other proxies will discover the abandoned managees and start managing them.

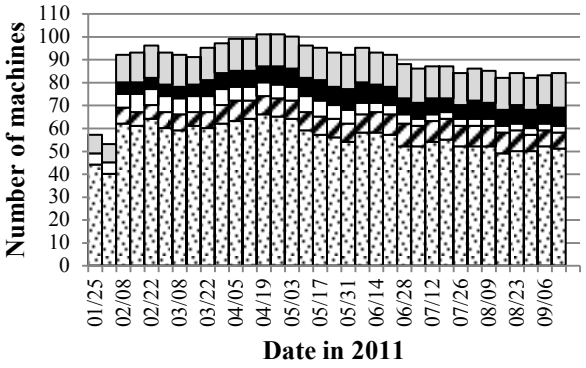

Figure 3: Number of participants each day, broken down by subnet. Each pattern depicts a different subnet.

7 Experiences and Lessons Learned

In this section, we evaluate GreenUp based on our deployment within Microsoft Research. Figure 3 shows the number of users per subnet. At peak, we had 101 participating user machines, with 66 on a single subnet. The jump on February 2 is due to an advertisement e-mail we sent. We see some attrition as people replace their computers and our advertising wanes, but as of September 13 there were still 84 participants. The participants were a mixture of 32-bit and 64-bit Windows 7 machines.

This section has two main goals: to demonstrate that GreenUp is effective and efficient, and to describe lessons we have learned that will help future practitioners. To do so, we answer the following questions. §7.1: Does GreenUp consistently wake machines when their services are accessed? When it does not, why not? §7.2: Is GreenUp scalable to large subnets? §7.3: Does GreenUp succeed at maintaining enough awake machines for the proxy service to be perpetually available? §7.4: How much energy does GreenUp potentially save? §7.5: How well does GreenUp do at waking a machine before the user attempting to connect gives up? For answers to questions unrelated to our distributed approach, such as the frequency of different wakeup causes, see our earlier work [23].

7.1 Machine availability

In this section, we evaluate how effective GreenUp is at meeting its main goal, waking machines when there are attempts to connect to them. First, we describe an availability experiment we conducted on GreenUp machines (§7.1.1). Then, since we find WoL failures to be a common cause of unavailability, we analyze GreenUp history to determine how often they occur (§7.1.2).

7.1.1 Availability experiment

To test the availability of machines running GreenUp, we used the following experiment. For each participating machine, we do the following from a single client: First, the client pings the machine, and records whether any response is received. Next, the client attempts to establish

Experiment start	Already awake	Woken	Unwakeable
8pm Sat, Mar 12	47	31	1
5pm Sun, Mar 13	56	24	1
9pm Mon, Mar 14	56	26	0
9pm Tue, Mar 15	55	26	0
10pm Wed, Sep 21	35	28	0
10pm Thu, Sep 22	44	19	0
6pm Sat, Sep 24	37	27	1
5pm Sun, Sep 25	38	27	0
11pm Mon, Sep 26	42	24	1
9pm Tue, Sep 27	41	25	1
10pm Wed, Sep 28	45	21	0

Table 1: Results of trying to wake GreenUp machines by connecting to them. WoL issues on two machines cause four failures; a temporary network problem causes one.

an SMB connection (TCP, port 139) to the machine. Barring a few exceptions, all machines in Microsoft have this port open. Finally, the client pings the machine again. If the machine responds to the first set of pings, we consider it *already awake*; otherwise, if it responds to the second set, we consider it *woken*; otherwise, we consider it *unwakeable*. In cases where the machine is unwakeable, we investigate whether it was in a state from which its owner would not expect GreenUp to wake it, and in that case do not count it. For instance, we do not count instances in which a machine was shut down, hibernated, removed from the network, or broken.

We exclude from consideration machines that appear to be laptops, since we cannot distinguish unwakeability from lack of physical connectivity. GreenUp is not intended for laptops unless they are tethered like desktops, but some users installed it on mobile laptops anyway.

Results of repeated runs of this experiment are shown in Table 1. We find that in 5 cases, 0.6% of the total, GreenUp failed to wake the machine.

Four failures occurred because two machines, one time each, entered a state in which WoL packets sent by GreenUp did not wake them. In each case, the state lasted long enough for two consecutive wake attempts to fail. For the machine unwakeable on March 12 and 13, the unwakeability seems to have been caused by GreenUp sending it a WoL packet too soon after it had become unavailable: 5 sec and 20 sec, respectively.

Our logs indicate that the fifth failure occurred because of an aberrant network condition that lasted for 36 minutes. During this time, the machine's manager was not receiving TCP SYNs from machines outside the subnet, but was receiving them from machines inside the subnet.

Our logs show that the data from this run is representative of the entire seven month deployment period. Thus, we conclude that GreenUp is extremely reliable. When it does fail to wake up a machine, it is typically because a WoL packet sent by the manager fails to wake up the managee. While these failures are rare, they merit more investigation, which we do next.

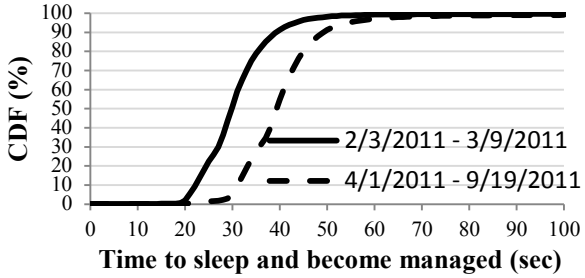

Figure 4: CDF of the time between when a machine begins to sleep and when it becomes managed, before and after deployment of a delay to mitigate WoL bugs.

7.1.2 Wake-on-LAN failures

We now investigate WoL failures in more detail. In many cases, we were able to trace the problem to out-of-date driver, firmware, or NIC hardware. However, we also discovered a rare WoL failure mode affecting a small subset of machines in our deployment. These machines respond correctly to WoL under normal conditions, but if we attempt to wake them up while they are in the process of sleeping, they became permanently unwakeable via WoL until the next reboot! Further tests revealed that the issue can generally be avoided if we wait at least 30 seconds after a machine becomes unresponsive before attempting a wakeup.

Thus, in mid-March we deployed an update that spends 25 seconds probing a machine before deciding to manage it instead of our original timeout of 15 seconds. This has the side benefit of reducing the likelihood that a manager mistakenly manages an awake machine, but it also increases the time to wake a machine in the unlikely event that an access occurs just as the machine falls asleep or its manager becomes unavailable. Fortunately, this event should be rare: Figure 4 shows that a machine is nearly always managed within a minute of beginning to sleep, and a minute is a small fraction of 3.6 hours, the average period a machine spends asleep.

We measured the overall frequency of WoL failures by searching GreenUp logs for events where a machine issues a WoL but the target does not wake up within 180 seconds—enough time for even the slowest-waking machines. However, we must be careful to exclude false positives, such as attempts to wake up a machine that has been shut down. We find that the WoL failure rate is not statistically different before and after the update, mainly because it is tiny to begin with: in the entire period from Feb 3 to Sep 19, 0.3% of WoL attempts are failures. We conclude that WoL is a generally reliable mechanism.

7.2 Scalability

We now evaluate the overhead of running GreenUp, including CPU utilization and network bandwidth, and the implications for scalability to large subnets.

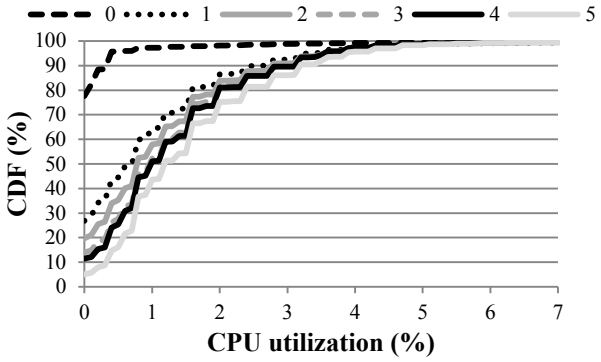

Figure 5: CDF of CPU utilization of GreenUp. Each line represents a different number of managed machines.

# of managees	CPU utilization
100	12%
200	21%
300	29%

Table 2: CPU utilization on a testbed machine as a result of simulating different management loads.

7.2.1 Observed CPU utilization

One concern about GreenUp is the load it places on each participating machine. Qualitatively, we note that our users have never complained about excessive load.

To evaluate the load quantitatively, we had GreenUp record, every ten minutes, the CPU utilization of its process. Figure 5 shows the results in CDF form: for a given number of managees, it shows the CDF of all CPU utilization observations made while the observer was managing that number of managees.

We see that CPU utilization jumps notably when the number of managees goes from zero to one. The median goes from 0.0% to 0.8%, the mean goes from 0.2% to 1.1%, and the 95th percentile utilization goes from 0.4% to 3.5%. This is because most of the CPU overhead is due to parsing sniffed packets, and this is only necessary when managing machines. We see that the overall overhead from parsing these packets is minor, increasing CPU utilization by less than 1%.

We also observe that managing additional machines increases overhead only slightly: going from one managed machine to five managed machines increases mean CPU utilization from 1.1% to 1.3%, and increases 95th percentile utilization from 3.5% to 3.9%. It is difficult to extrapolate from this data since it occupies such a small range, but it suggests that managing 100 machines would yield average CPU utilization of about 13%. In the following subsection, we present another way of estimating the overhead of managing multiple machines.

7.2.2 Scalability of CPU utilization

For this experiment, we simulate the load of managing a large number of machines in a small testbed. We use three machines, each a 32-bit Windows 7 machine with a 2.4-GHz Intel Core 2 6600, 4 GB of memory, and a 1 Gb/s Ethernet card. One goes to sleep, one manages the sleeping machine, and one just sends probes. We increase the probe rate so that the manager sends and receives as many probes as it would if there were n sleeping participants per proxy. We disable the load fail-safe discussed in §6 so we can observe CPU utilization no matter how high it gets.

Note that there are other, minor sources of overhead from managing other machines that we do not simulate. We do not simulate sending heartbeats for each managee, since sending one heartbeat per managee every five minutes is dwarfed by the overhead of sending ~0.8 probes per managee every second. We also do not simulate replying to ARP and ND requests, since we have found such requests to be rare: fewer than 0.05/sec per managee. Finally, we do not simulate parsing SYNs destined for managees since these are even rarer.

Table 2 shows the load on the manager, which we note is somewhat underpowered compared to the typical desktop in our organization. We see that even for this machine, its average CPU utilization when managing 100 machines is 12%, well under our target 20% cap. Recall that we ensure that at least one proxy is always awake for every 100 participants in the subnet, so this shows that load is kept at a reasonable rate. The table also shows that as the number of managees rises to 200 and above, load becomes less reasonable, arguing for our requirement of at least one proxy per 100 participants.

Note that we could reduce this load substantially by increasing the probing period. For instance, changing s from 3 sec to 6 sec would reduce the load by roughly half. The cost is that it would take slightly longer for an unmanaged sleeping machine to become managed. This trade-off hardly seems worthwhile given that we rarely expect a manager to have even close to 100 managees. After all, as we show in §7.3, a far greater fraction of machines than 1/100 tend to be awake at any given time.

7.2.3 Network utilization

Another factor impacting scalability is network utilization. Each GreenUp participant sends and receives probes and heartbeats, and in some cases this traffic scales with the number of participants on the subnet. Qualitatively, this overhead is tiny; we have not received any complaints about GreenUp's network utilization.

To evaluate network utilization analytically, we begin by noting the sizes of various packets we use. The probe SYNs, UDP responses, and ICMP pings are all less than 100 bytes including all headers. The heartbeat packet size depends on the number of ports reported; on our main subnet, the maximum is 554 bytes and the average is 255 bytes. A simple calculation, omitted due to lack of space, shows that even if a machine were to manage

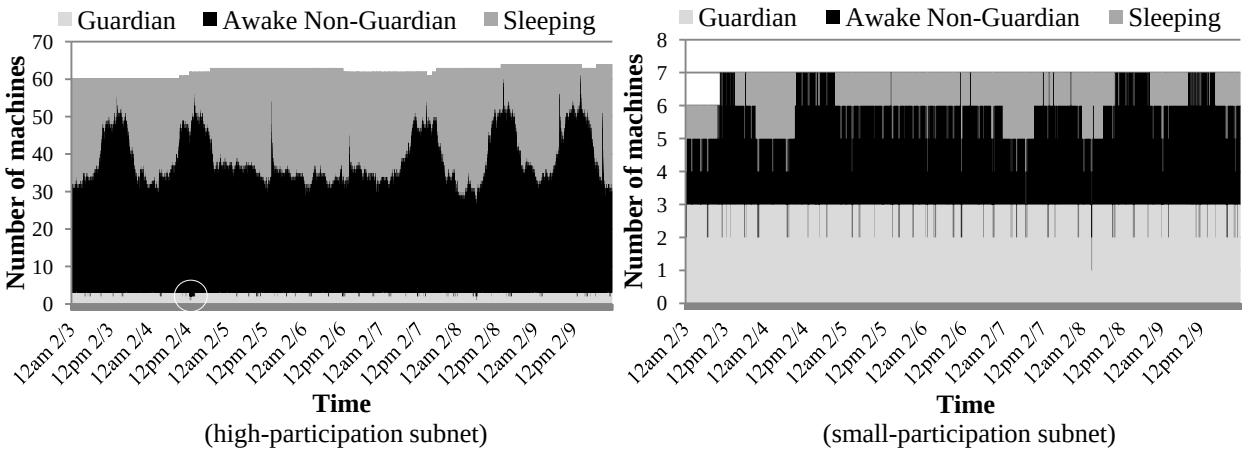

Figure 6: For two different subnets, the number of machines in various roles over time. A circle overlayed on the left graph shows an anomalously long period of time with fewer than three guardians, due to a bug in the implementation.

100 participants in a 1,000-node subnet, it would consume less than 90 Kb/s of upload bandwidth and 60 Kb/s of download bandwidth. Since typical enterprise LANs have 1 Gb/s capacity, this load is negligible.

7.3 GreenUp availability

We now evaluate the effectiveness of GreenUp's guardianship mechanism. Recall that it aims to keep at least three machines awake on each subnet.

The methodology for our evaluation is as follows. Every minute, from a single server, we ping every participating machine. If it does not respond, we consider it asleep. Otherwise, we consider it a guardian if its logs indicate it was a guardian then.

Figure 6 shows the results for the subnet with the largest participation and for a representative subnet with low participation. We see that our guardianship protocol works as expected: generally, there are three guardians, and when occasionally a guardian becomes unavailable, it either returns to availability or is replaced. In one instance, a guardian failed in an undetected manner and was not replaced. This is highlighted in the figure with a white circle: February 4 between 12:46 pm and 2:10 pm. We traced this to a bug in the implementation, which we promptly fixed: if GreenUp is uninstalled from a guardian machine but the machine remains awake, others incorrectly believe it remains a guardian.

Another important element we observe is that at all times a large number of machines are awake even though they are not guardians. This observation suggests that we could save energy by preferentially selecting machines as guardians if they would be awake anyway; this is something we plan to do in future work. By prioritizing already-awake machines to be guardians, we should be able to fully eliminate the energy cost of keeping guardians awake. On the other hand, this observation also shows that choosing random machines to keep awake has a lower cost than expected. Since at least half of non-guardian machines are typically awake, this means that at least half the time a machine is a guardian it would have been awake anyway.

7.4 Energy savings

To evaluate how much energy GreenUp can save, we measure the amount of time GreenUp participants spend asleep. This is readily deduced from logs that indicate when machines went to sleep and woke up. Figure 7 shows the results for Feb 3 through Sep 19; overall, the average participant spends 31% of its time asleep.

An alternative approach to GreenUp's goal of high availability is to not let machines sleep. Thus, by using GreenUp rather than keeping machines awake, we increase sleep time by approximately 31% per day. If the average idle power consumed by desktops is 65 W, then GreenUp saves approximately 175 kWh per machine per year; at 10¢/kWh, this is $17.50 per machine per year.

It would be useful to evaluate whether GreenUp induces users to let their computers sleep more. However, as discussed in §2, the IT department in our organization mandates and enforces a common sleep policy, and users must go to great lengths to change machine sleep behavior. Thus, for our users the benefit of GreenUp is availability, not energy savings. In future work, we would like to evaluate whether, in an organization without such mandates, GreenUp induces users to choose more aggressive sleep policies and thereby save energy.

7.5 User patience

GreenUp's design is predicated on the assumption that users trying to connect to a sleeping machine will continue trying long enough for GreenUp to detect this and wake the machine. To validate this, we now characterize such user patience, and compare it to how long GreenUp takes to wake machines.

To evaluate user patience, we exploit a serendipitous side effect of the occasional unreliability of WoL. When

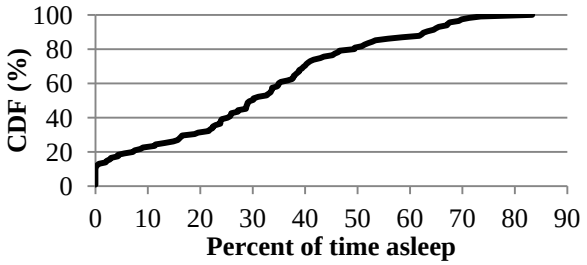

Figure 7: CDF showing distribution, among GreenUp participants, of fraction of time spent asleep.

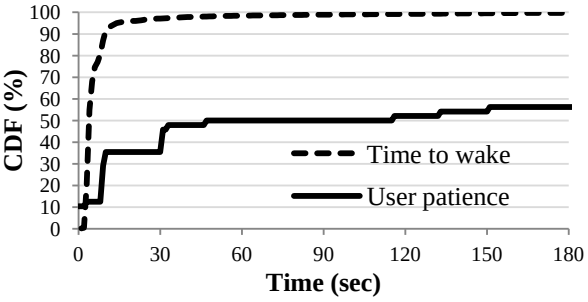

Figure 8: Comparison of how long GreenUp takes to wake machines with how long it has to wake machines, based on evaluation of user patience

a manager detects a TCP connection attempt, but the WoL packet it sends has no effect, the manager's logs show just how long the user continued trying, in vain, to connect to the machine.

In our trace data from Feb 3 through Sep 19, we found 48 events in which there was a connection attempt that allowed us to gauge user patience. Figure 8 shows the results. We find the CDF to be a step function, which is unsurprising given the behavior of TCP: after the first SYN, it sends the next four SYNs roughly 3 sec, 9 sec, 21 sec, and 35 sec later. It appears that 10% of connection attempts send only the first SYN and thus provide GreenUp no time at all to wake the machine. Most likely, many of these are port scans from the IT department's security system, but since we cannot definitively tell this we conservatively estimate them to be highly impatient users. A further 3% send only two SYNs, giving GreenUp 3 sec to wake the machine. A further 22% send only three SYNs, giving GreenUp 9 sec. The remaining 65% send at least five SYNs, giving GreenUp more than 30 sec. Indeed, the most patient 44% seem to try connecting for over five minutes.

Figure 8 also shows how long it takes for GreenUp to wake machines via WoL, using the methodology from §7.1.2. We see that 87% of wakeups take 9 sec or less, so even the 22% of users who are so impatient as to wait only this long can often be satisfied. A full 97% of wakeups take 30 sec or less, so the 65% of users who wait at least this long can nearly always be satisfied. Naturally, the 44% of users who wait at least five minutes are satisfied as long as GreenUp eventually wakes the machine, which as we showed earlier happens over 99% of the time.

Convolving the distribution of user patience with wake time, we find that GreenUp will wake the machine quickly enough to satisfy user patience about 85% of the time. This is about as good as can be expected given that 13% of the time users, or possibly port scanners appearing to be users, are so impatient as to stop trying after only 3 sec. It may be useful, in future deployments, to convey to users the value of waiting longer than this for a connection.

8 Discussion and Future Work

The feedback on GreenUp has been positive, and a larger deployment on ~1,100 machines is currently ongoing. We also continue to improve GreenUp and our users' experience. This section discusses some key issues and areas of ongoing work.

Dynamic layer 2 routing. GreenUp uses spoofed packets to make switches send packets meant for a sleeping machine to its manager. One may view this as meddling with the layer 2 routing protocol, but this is a very common technique, and has been used for many years. Indeed, exactly the same process takes place when a machine is unplugged from one Ethernet port and plugged into another. Some organizations disallow such port changes, but this is not common because it prevents employees from moving machines around without approval from IT staff. In our experience, no switches have exhibited adverse behavior due to such routing changes.

Handling encrypted traffic. GreenUp does not work if all traffic on the network is encrypted. Indeed, we had to implement special exceptions [26] when our organization began using DirectAccess [15], an alternative to VPN that encrypts all traffic with IPSec. We note that this shortcoming is shared by centralized schemes [18, 23] as well as NIC-based schemes [2]. We are currently devising techniques to improve our handling of encrypted traffic. For example, a manager can identify IPSec key exchange traffic to a sleeping machine and determine when a new security association is being set up or renewed; in certain scenarios, it may be appropriate to wake the machine. However, the only foolproof way to deal with encrypted traffic is to use a system such as LiteGreen [8], or to modify the client to send a specially crafted packet before it attempts to establish a connection to a sleeping machine.

Increasing availability. GreenUp achieves over 99% availability; we want to do better. Since most failures are due to WoL problems, we have built a mechanism to automatically test each participant for persistent WoL issues and inform users. However, we want to get to the

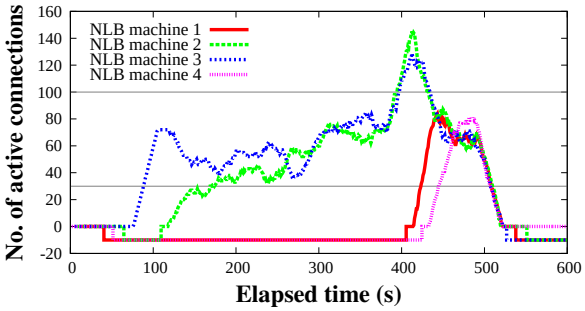

Figure 9: Load profile of a four-machine energy-efficient NLB cluster. A negative load indicates the machine is asleep. Traffic is gradually increased between 75 sec and 475 sec and then stopped.

root of all WoL issues and either directly fix them or inform hardware and software developers how to fix them.

A brief window of unavailability occurs after a sleeping machine's manager becomes unavailable and before a new manager picks it up. We described a strategy in §5.2 that eliminates this time via an explicit hand-off message. Analyzing the load properties of this strategy is our main theoretical pursuit.

Choosing better guardians. Instead of choosing random guardians, it would be more efficient to choose machines that are going to be awake anyway. Thus, we are devising a way to predict when a machine is likely to stay awake and use this information to choose guardians. Since the views of different participants may be stale or divergent, a more complicated protocol is required for allowing a machine to stop being a guardian.

Other applications of GreenUp. Although we have focused on the enterprise desktop setting, the techniques used in GreenUp also apply to certain server cluster settings. In particular, we have used GreenUp to build an energy-efficient version of Windows Network Load Balancing (NLB), a distributed software load balancer used by web sites like `microsoft.com`. NLB runs all machines of a cluster all the time; we used GreenUp to adapt the NLB cluster size to the current load. Specifically, we added a load manager module to GreenUp, using ~300 lines of C# code, to put a machine to sleep or wake another machine if the load is too low or too high, respectively. The load manager interacts with NLB via NLB's management interface—no changes are made to the NLB code. We use a priority-based scheme similar to the one for apocalypse prevention to ensure that only one machine goes to sleep or wakes up at at time. This ensures a minimum number of awake machines, and prevents too many from waking up during high load, to save energy.

We tested our prototype on a four-machine NLB cluster with an IIS web server serving static files. A client program requests a 1MB file with increasing frequency in blocks of 100 seconds for 400 seconds. We use the number of active connections as a measure of load and set the low and high load thresholds to 30 and 100, respectively. Figure 9 shows that our simple scheme achieves dynamic energy efficiency. Initially, there is no load so only machine 3 is awake. At 75 sec, it starts handling requests but quickly detects a steep load slope and wakes up machine 2. The two machines handle the load until 395 sec, when both cross the high threshold and hence wake up machines 1 and 4. The load is quickly redistributed until 475 sec, when the experiment stops and all but one machine soon sleep.

9 Related work

We briefly describe recent work and contrast it with ours. This work falls in two categories: handling sleeping machines, and coordination in distributed systems.

9.1 Sleep and wakeup

We have already described the work most closely related to ours [19, 23]. A somewhat different approach is taken by LiteGreen [8], which requires each user's desktop environment to be in a VM. This VM is live-migrated to a central server when the desktop is idle, and brought back to the user's desktop when the user returns. The main drawback of this system is the need to deploy VM technology on all desktops, which can be quite disruptive. In contrast, our system is easier to deploy and requires no central server.

The SleepServer system [3] uses application stubs to run specially-modified applications on a sleep server while the host machine sleeps. While this allows applications such as BitTorrent to keep sessions alive, it requires modifying code and developing stubs for each application. This is a significant barrier to deployment.

Apple offers a sleep proxy for home users, but it works only with Apple hardware. Adaptiva [1] and Verdiem [30] enable system administrators to wake machines up for management tasks, albeit not seamlessly.

We do not discuss data center power management, as that environment is very different from the enterprise.

9.2 Coordination in distributed systems

One way to coordinate actions in a distributed system is with a replicated state machine [25] that tolerates crash failures [14] or Byzantine failures [6]. These protocols are appropriate when strong consistency of distributed state is required and the set of machines is relatively stable. This is because during conditions such as network partitions, they pause operation until a quorum can be attained. Even systems like Zeno [28] that are designed for higher availability require weak quorums to make progress. In our system, availability is the highest priority, and we can tolerate temporary state inconsistency. The set of awake machines is also highly dynamic. Thus, these approaches are inappropriate.

Other techniques for coordination include distributed hash tables [22, 24, 29, 34] and gossip [9, 10]. These

solve the problem of disseminating information among a changing set of machines while also providing high availability. However, they are designed for wide-area networks where no efficient broadcast medium is available. Thus, we propose subnet state coordination as a new alternative when the system runs within a subnet.

Like GreenUp, there are other distributed systems that use randomization to spread load. For example, in RAMCloud [20], masters randomly select backup servers to store their replicas. Also, each RAMCloud server periodically pings one other server at random, reporting any failures to the cluster coordinator. Our distributed probing technique shows how to tune the number of probes to guarantee any desired probability of detecting a failure.

10 Conclusions

This paper presented the design of GreenUp, a decentralized system for providing high availability to sleeping machines. GreenUp operates in a unique setting where broadcast is efficient, machines are highly unreliable, and users must be satisfied with the system to use it. Our design addressed these challenges using new techniques for coordinating subnet state and managing asleep machines. Our analysis of the design showed that it achieves desired availability and efficiency properties. We also implemented GreenUp and deployed it on ~100 users' machines, providing many useful lessons and demonstrating GreenUp's practicality and efficacy. Ultimately, we were able to meet the stringent demands of IT departments for a low-cost, low-administration, and highly-reliable system for keeping sleeping machines available.

Acknowledgments

We thank system administration staff from MSIT and MSR for their invaluable help in making this project possible. We also thank our shepherd, Brian Noble, and the anonymous reviewers for their helpful feedback. Finally, we thank all the users who installed GreenUp, allowing us to collect data and refine our code.

References

[1] Adaptiva. Optimize Config Mgr with Adaptiva Green Planet, Client Health, One Site. http://www.adaptiva.com/, 2009.

[2] Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting network interfaces to reduce PC energy usage. In *NSDI*, 2009.

[3] Y. Agarwal, S. Savage, and R. Gupta. SleepServer: A software-only approach for reducing the energy consumption of PCs within enterprise environments. In *USENIX*, 2010.

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

[5] AMD. Magic packet technology. Technical report, AMD, 1995.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26, 2008.

[8] T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. G. Shin. LiteGreen: Saving energy in networked desktops using virtualization. In *USENIX*, 2010.

[9] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *PODC*, 1987.

[10] P. Eugster, R. Guerraoui, S. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4), 2003.

[11] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. Advanced Configuration and Power Interface, revison 4.0, 2009.

[12] J. F. C. Kingman. The coalescent. *Stoch. Process. Appl.*, 13, 1982.

[13] D. E. Knuth. *The Art of Computer Programming*. Vol. 2: *Seminumerical Algorithms*. Addison-Wesley, 1988. 2nd ed.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[15] Microsoft Corporation. DirectAccess. http://technet.microsoft.com/en-us/network/dd420463.aspx, 2011.

[16] Microsoft Corporation customer. Private communication, 2010.

[17] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing network energy consumption via sleeping and rate-adaptation. In *NSDI*, 2008.

[19] 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*, 2009.

[20] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *SOSP*, 2011.

[21] M. Raab and A. Steger. "Balls into bins" - A simple and tight analysis. In *RANDOM*, 1998.

[22] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[23] J. Reich, M. Goraczko, A. Kansal, and J. Padhye. Sleepless in Seattle no longer. In *USENIX*, 2010.

[24] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[26] S. Sen, J. R. Lorch, R. Hughes, C. Garcia, B. Zill, W. Cordeiro, and J. Padhye. GreenUp: A decentralized system for making sleeping machines available. Technical Report MSR-TR-2012-21, Microsoft Research, 2012.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *MSST*, 2010.

[28] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent Byzantine fault tolerance. In *NSDI*, 2009.

[29] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for Internet applications. In *SIGCOMM*, 2001.

[30] Verdiem Corporation. http://www.verdiem.com/, 2010.

[31] D. Washburn and O. King. How much money are your idle PCs wasting? Forrester Research Reports, 2008.

[32] 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*, 2006.

[33] D. A. Wheeler. Linux kernel 2.6: It's worth more! http://www.dwheeler.com/essays/linux-kernel-cost.html, 2004.

[34] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J. Sel. Areas Commun.*, 22(1), 2004.