Energy-Aware Server Provisioning and Load Dispatching for Connection-Intensive Internet Services

Gong Chen*, Wenbo He⋆, Jie Liu†, Suman Nath†, Leonidas Rigas‡, Lin Xiao†, Feng Zhao†
*Dept. of Statistics, University of California, Los Angeles, CA 90095
⋆Dept. of Computer Science, University of Illinois, Urbana-Champaign, IL 61801
‡Microsoft †Research, One Microsoft Way, Redmond, WA 98052
gchen@stat.ucla.edu, webohe@uiuc.edu
{liuj, sumann, leonr, lixiao, zhao}@microsoft.com

Abstract

Energy consumption in hosting Internet services is becoming a pressing issue as these services scale up. Dynamic server provisioning techniques are effective in turning off unnecessary servers to save energy. Such techniques, mostly studied for request-response services, face challenges in the context of connection servers that host a large number of long-lived TCP connections. In this paper, we characterize unique properties, performance, and power models of connection servers, based on a real data trace collected from the deployed Windows Live Messenger. Using the models, we design server provisioning and load dispatching algorithms and study subtle interactions between them. We show that our algorithms can save a significant amount of energy without sacrificing user experiences.

1 Introduction

Internet services such as search, web-mail, online chatting, and online gaming, have become part of people's everyday life. Such services are expected to scale well, to guarantee performance (e.g., small latency), and to be highly available. To achieve these goals, these services are typically deployed in clusters of massive number of servers hosted in dedicated data centers. Each data center houses a large number of heterogeneous components for computing, storage, and networking, together with an infrastructure to distribute power and provide cooling.

Viewed from the outside, a data center is a "black box" that responds to a stream of requests from the Internet, while consuming power from the electrical grid and producing waste heat. As the demand on Internet services drastically increases in recent years, the energy used by data centers, directly related to the number of hosted servers and their workload, has been skyrocketing [8]. In 2006, U.S. data centers consumed an estimated 61 billion kilowatt-hours (kWh) of energy, enough to power 5.8 million average US households.

Data center energy savings can come from a number of places: on the hardware and facility side, e.g., by designing energy-efficient servers and data center infrastructures, and on the software side, e.g., through resource management. In this paper, we take a software-based approach, consisting of two interdependent techniques: *dynamic provisioning* that dynamically turns on a minimum number of servers required to satisfy application-specific quality of service, and *load dispatching* that distributes current load among the running machines. Our approach is motivated by two observations from real data sets collected from operating Internet services. First, the total load of a typical Internet service fluctuates over a day. For example, the fluctuation for the number of users logged on to Windows Live Messenger can be about 40% of the peak load within a day. Similar patterns were found in other studies [3, 20]. Second, an active server, even when it is kept idle, consumes a non-trivial amount of power ($> 66\%$ of the peak from our measurements, similar to other studies [9]). The first observation provides us the opportunity to dynamically change the number of active servers, while the second observation implies that shutting down machines during off-peak period provides maximum power savings.

Both dynamic provisioning and load dispatching have been extensively studied in literature [20, 3, 6]. However, prior work studies them separately since the main foci were on *request-response* type of services, such as Web serving. Simple Web server transactions are typically short. There is no state to carry over when the total number of servers changes. In that context, first the minimum number of servers sufficient to handle the current request rate is determined. After the required number of servers are turned on, incoming requests are typically distributed evenly among them [3].

Many Internet services keep long-lived connections. For example, HTTP1.1 compatible web servers can optionally keep the TCP connection after returning the initial web request in order to improve future response per-

formance. An extreme case is connection-oriented services like in instant messaging, video sharing, Internet games, and virtual life applications, where users may be continuously logged in for hours or even days. For a connection server, the number of connections on a server is an integral of its net login rate (gross login rate minus logout rate) over the time it has been on. The power consumption of the server is a function of many factors such as number of active connections and login rate. Unlike request-response servers that serve short-lived transactions, long-lived connections in connection servers present unique characteristics:

1. The capacity of a connection server is usually constrained by both the rate at which it can accept new connections and the total number of active connections on the server. Moreover, the maximum tolerable connection rate is typically much smaller than the total number of active connections due to several reasons such as expensive connection setup procedure and conservative SLA (Service Level Agreement) with back-end authentication services or user profiles. This implies that when a new server is turned on, it cannot be fully utilized immediately like simple web servers. The load, in terms of how many users it hosts, can only increase gradually.
2. If the number of server is under provisioned, new login requests will be rejected and users receive "service not available" (SNA) errors. When a connection server with active users is turned off, users may experience a short period of disconnection, called "server initiated disconnections" (SID). When this happens, the client software typically tries to reconnect back, which may create an artificial surge on the number of new connections, and generate unnecessary SNAs. Both errors should be avoided to perserve user experiences.
3. As we will see in Section 3.5, the energy cost of maintaining a connection is orders of magnitude smaller than processing a new connection request. Thus, a provisioning algorithm that turns off a server with a large number of connections, which in turn creates a large number of reconnections, may defeat the purpose of energy saving.

Due to these unique properties, existing dynamic provisioning and load dispatching algorithms designed for request-response services may not work well with connection services. For example, consider a simple strategy that dynamically turns on or off servers based on current number of users and then balances load among all active servers [3]. Since a connection server takes time, say T, to be fully utilized, provisioning X new servers based on *current users* may not be sufficient. Note that newly booted servers will require T time to take the target load, and during this time the service will have fewer fully utilized servers than needed, causing poor service. Moreover, after time T of turning on X new servers, workload can significantly change, making these X new servers either insufficient or unnecessary. Consider, on the other hand, that a server with N connections is turned off abruptly when the average connected users per server is low. All those users will be disconnected. Since many disconnected clients will automatically re-login, currently available servers may not be able to handle the surge. In short, a reactive provisioning system is very likely to cause poor service or create instability. This can be avoided by a *proactive* algorithm that takes the transient behavior into account. For example, we can employ a load prediction algorithm to turn on machines gradually before they are needed and to avoid turning on unnecessary machines to cope with temporary spikes in load. At the same time, the provisioning algorithm needs to work together with load dispatching mechanism and anticipate the side effect of changing server numbers. For example, if loads on servers are intentionally skewed, instead of balanced, to create "tail" servers with fewer live connections, then turning off a tail server will not generate any big surge to affect login patterns.

In this paper, we develop power saving techniques for connection services, and evaluate the techniques using data traces from Windows Live Messenger (formerly MSN Messenger), a popular instant messaging service with millions of users. We consider server provisioning and load dispatching in a single framework, and evaluate various load skewing techniques to trade off between energy saving and quality of service. Although the problem is motivated by Messenger services, the results should apply to other connection-oriented services. The **contributions** of the paper are:

- We characterize performance, power, and user experience models for Windows Live Messenger connection servers based on real data collected over a period of 45 days.
- We design a common provisioning framework that trades off power saving and user experiences. It takes into account the server transient behavior and accommodates various load dispatching algorithms.
- We design load skewing algorithms that allow significant amount of energy saving (up to 30%) without sacrificing user experiences, i.e., maintaining very small number of SIDs.

The rest of paper is organized as follows. Section 2 discusses related work. In Section 3, we give a brief overview of the Messenger connection server behavior and characterize connection server power, performance,

and user experience models. We present the load prediction and server provisioning algorithms in Section 4, and load dispatching algorithms in Section 5. Using data traces from deployed Internet services, we evaluate various algorithms and show the results in Section 6. Finally, we conclude the paper with discussions in Section 7.

2 Related Work

In modern CPUs, P-states and clock modulation mechanisms are available to control CPU power consumption according to the load at any particular moment. Dynamic Voltage/Frequency Scaling (DVFS) has been developed as a standard technique to achieve power efficiency of processors [19, 13, 17]. DVFS is a powerful adaptation mechanism, which adjusts power provisioning according to workload in computing systems. A control-based DVFS policy combined with request batching has been proposed in [7], which trades off system responsiveness to power saving and adopts a feedback control framework to maintain a specified response time level. A DVFS policy is implemented in [21] on a stand-alone Apache web server, which manages tasks to meet soft real-time deadlines. Flautner et al. [10] adopts performance-setting algorithms for different workload characteristics transparently, and implements the DVFS policy on per-task basis.

A lot of efforts have been made to address power efficiency in data centers. In [11], Ganesh, et al. proposed a file system based solution to reduce disk array energy consumption. The connection servers we study in this paper have little disk IO load. Our focus is on provisioning entire servers. In [20], Pinheiro et al. presented a simple policy to turn cluster nodes on and off dynamically. Chase et al. [3] allocated computing resources based on an economic approach, where services "bid" for resources as a function of required performance, and the system continuously monitors load and allocates resources based on its utility. Heath et al. [14] studied Web service on-off strategies in the context of heterogeneous server types, although focusing on only short transactions. Various other work using adaptive policies to achieve power efficiency have been proposed. Abdelzaher et al. in [1] employed a Proportional-Integration (PI) controller for an Apache Web server to control the assigned processes (or threads) and to meet soft realtime latency requirements. Other work based on feedback and/or optimal control theory includes [6, 5, 23, 22, 16, 15], which attempt to dynamically optimize for energy, resources and operational costs while meeting performance-based SLAs.

In contrast to previous work, we consider and design load dispatching and dynamic provisioning schemes together since we observe that, in the context of connection servers, they have subtle interaction with each other. These two components work together to control power consumption and performance levels. Moreover, we adopt forecast-based and hysteresis-based provisioning to bear with slow rebooting of servers.

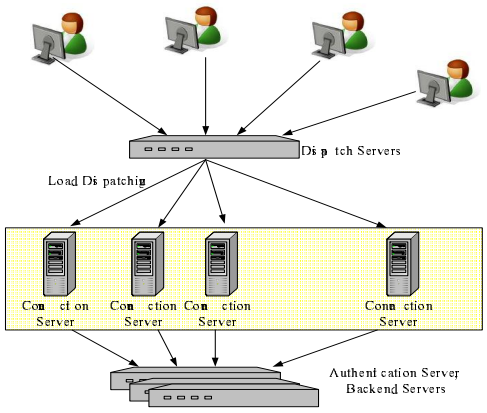


Figure 1: Connection service architecture.

3 Connection Servers Background

Connection servers are essential for many Internet services. In this paper, we consider dedicated connection servers, each of which runs only one connection service application. However, the discussion can be generalized to servers hosting multiple services as well.

3.1 Connection Service

Figure 1 shows an example of the front door architecture for connection intensive Internet applications. Users, through dedicated clients, applets, or browser plug-ins, issue login requests to the service cloud. These login requests first reach a dispatch server (DS), which picks a connection server (CS) and returns its IP address to the client. The client then directly connects to the CS. The CS authenticates the user and if succeeded, a live TCP connection is maintained between the client and the CS until the client logs off. The TCP connection is usually used to update user status (e.g. on-line, busy, off-line, etc.) and to redirect further activities such as chatting and multimedia conferencing to other back-end servers.

At the application level, each CS is subject to two major constraints: the maximum login rate and the maximum number of sockets it can host. The new user login rate L is defined as the number of new connection requests that a CS processes in a second. A limit on login rate $L_{\max}$ is set to protect CS and other back-end services. For example, Messenger uses Windows Live ID (a.k.a. Passport) service for user authentication. Since Live ID is also shared by other Microsoft and non-Microsoft applications, a service-level agreement (SLA)

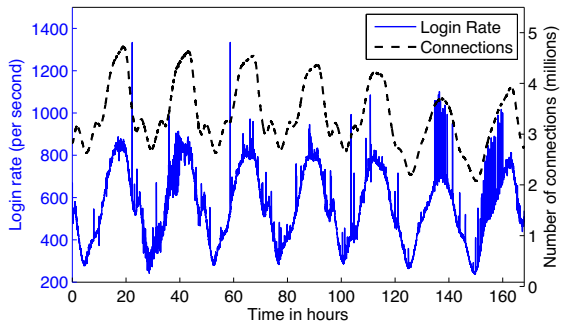


Figure 2: One week of load pattern (Monday to Sunday) from Windows Live Messenger connection servers

is set to bound the number of authentication requests that Messenger can forward. In addition, a new user login is a CPU intensive operation on the CS, as we will show in Section 3.5. Having $L_{\max}$ protects the CS from being overloaded or entering unsafe operation regions.

In terms of concurrent live TCP connections, a limit $N_{\max}$ is set on the total number of sockets for each CS for two reasons: the memory constraints and the fault tolerance concerns. Maintaining a TCP socket is relatively cheap for the CPU but requires a certain amount of memory. At the same time, if a CS crashes, all its users will be disconnected. As most users set their clients to automatically reconnect when disconnected, a large number of new login requests will hit the servers. Since there is a limit on new login rate, not all reconnect requests can be processed in a short period of time, creating undesirable user experiences.

3.2 Load Patterns

Popular connection servers exhibit periodic load patterns with large fluctuation, similar to those reported in [6].

Figure 2 shows a pattern of the login rates and total number of connections to the Messenger service over the course of a week. Only a subset of the data, scaled to 5 million connections on 60 connection servers, is reported here. It is clear that the number of connections fluctuates over day and night times. In fact, for most days, the amount of fluctuation is about 40% of the corresponding peak load. Some geo-distributed services show even larger fluctuation. The login rates for the same time period are scaled similarly. It is worth noting that the login rate is noisier than the connection count.

Since the total number of servers must be provisioned to handle peak load to guarantee service availability, it is a "overkill" when the load is low. If we can adapt server resource utilization accordingly, we should be able to save substantial energy. In addition, the smooth pattern indicates that the total number of connections is predictable. In fact, if we look at the pattern over weeks, the similarity is more significant. Since turning on servers takes time, the prediction will help us act early.

3.3 Power Model

Understanding how power are consumed by connection servers provides us insights on energy saving strategies. Connection servers are CPU, network, and memory intensive servers. There is almost no disk IO in normal operation, except occasional log writing. Since memory is typically pre-allocated to prevent run-time performance hit, the main contributor to the power consumption variations of a server is the CPU utilization.

We measured power consumption of typical servers while changing the CPU utilization by using variable workloads. Figure 3 shows the power consumption on two types of servers, where the horizontal axis indicates the average CPU utilization reported by the OS, and the vertical axis indicates the average power consumption measured at the server power plug.

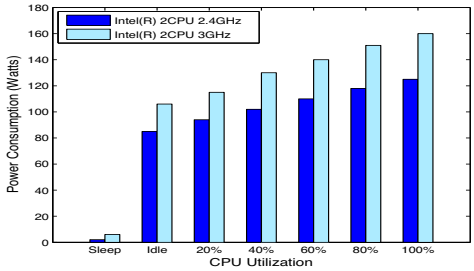


Figure 3: Power consumption v.s. CPU utilization

We observe two important facts. First, the power consumption increases almost linearly with CPU utilization, as reported in other studies [9]. Second, an idle server consumes up to 66% of the peak power, because even when a server is not loaded with user tasks, the power needed to run the OS and to maintain hardware peripherals, such as memory, disks, master board, PCI slots, and fans, is not negligible. Figure 3 implies that if we pack connections and login requests to a portion of servers, and keep the rest of servers hibernating (or shutting-down), we can achieve significant power savings. However, the consolidation of login requests results in high utilization of those servers, which may downgrade performance and user experiences. Hence, it is important to understand the user experience model before we address power saving schemes for large-scale Internet service.

3.4 User Experiences

In the context of connection servers, we consider the following factors as user experience metric.

1. Service Not Available (SNA) Error. Service unavailable is an error message returned to the client software when there is not enough resource to handle user login.

This happens when DS cannot find a CS that can take the new login request, or when a CS receives a login request, it observes that its login rate or number of existing sockets exceeds corresponding limits.

2. Server-Initiated Disconnection (SID). A disconnection happens when a live user socket is terminated before the user issues a log off request. This can be caused by network failures between the client and the server, by server crash, or by the server sending a "reconnect" request to the client. Since network failure and server crash are not controlled by the server software, we use server-initiated disconnection (SID) as the metric for short term user experience degradation. SID is usually a part of connection server protocol, like MSNP [18], so that a server can be shut down gracefully (for software update for example). SID can be handled transparently by the client software so that it is un-noticeable to users. Nevertheless, some users, if they disable automated reconnect or are in active communication with their buddies, may observe transient disconnections.

3. Transaction Latency. Research on quality of services, e.g. [21, 6], often uses transaction delays as a metric for user experiences. However, after we examined connection server transactions, including user login, authentication, messaging redirection, etc., we observe no direct correlation between transaction delays and server load (in terms of CPU, number of connection, and login rate). The reason is that connection services are not CPU bounded. The load of processing transactions is well under control when the number of connection and the login rates are bounded. So, we will not consider transaction delays as a quality of service metric in this paper.

From this architectural description, it is clear that the DS and its load dispatching algorithm play a critical role in the shape of the load in connection servers. This motivates us to focus on the interaction between CS and DS, the provisioning algorithms and load dispatching algorithms to achieve power saving while maintaining user experiences in terms of SNA and SID.

3.5 Performance Model

To characterize the effect of load dispatching on service loads, it is important to understand the relationship between application level parameters such as user login and physical parameters such as CPU utilization and power consumption. In other words, we need to identify the variables that significantly affect CPU and power. This would enable us to control CPU usage and power consumption of the servers by controlling these variables.

We use a workload trace of Messenger service to identify the key variables affecting CPU usage and power. The trace contains 32 different performance counters such as login rate, connection count, memory usage, CPU usage, connection failures, etc. of all production Messenger servers, over a period of 45 days. To keep the data collection process lightweight, aggregated values over 1 second are recorded every 30 seconds.

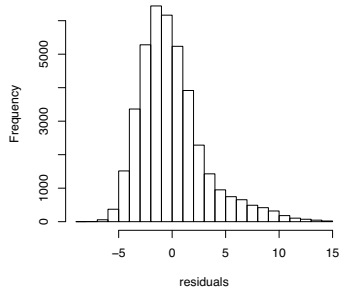


Figure 4: Residual distribution of performance model.

Given time series data of all available variables, we intend to identify important variables affecting the response variable (CPU utilization) and build a model of the response variable. Our statistical modeling methodology includes various exploratory data analysis, bi-direction model selection, diagnostic procedures and performance validation. Readers can refer to our technical report [4] for details.

The final model obtained from our methodology is:

$$\hat{U} = 2.84 \times 10^{-4} \cdot N + 0.549 \cdot L - 0.820 \quad (1)$$

where $\hat{U}$ denotes the estimate of the CPU utilization percentage—the conditional mean in linear models, N is the number of active connections, and L is login rate. For example, given 70,000 connections and 15 new logins per second, the model estimates the CPU utilization to be 27.3%. The residual (error) distribution of this model can be seen from Figure 4. It shows that for 91.4% percent of cases, the observed values are within the range from -5 to 5 of the estimates from the model. Referring to the previous example, the actual CPU utilization falls in between 22.30 and 32.30 with probability 0.914. About 7.6% cases for which the model makes underestimation have larger residuals from 5 to 15. The remaining 1% cases have residuals less than -5.

We use 2 weeks of data to train the model, and another 4 weeks for validation. The validation demonstrates that the distributions of testing errors are consistent with the distribution of training errors, concluding that CPU usage (and hence power consumption) of Messenger servers can reasonably be modeled with login rate (L) and number of active connections (N). Moreover, for a given maximum tolerable CPU utilization (defined by the Messenger operations group), the model also provides $N_{\max}$ and $L_{\max}$, maximum tolerable values for N and L. Our provisioning and load dispatching algorithms therefore consider only these two variables and try to ensure that $N < N_{\max}$ and $L < L_{\max}$, while minimizing total energy consumption.

4 Energy-Aware Server Provisioning

As we explained in the previous section, there are two aspects of the connection load, namely the login rate and number of connections. Performance modeling in Section 3.5 validates the operation practice of setting limits $L_{\max}$ and $N_{\max}$ on individual servers for purposes of server provisioning and load dispatching. In this section, we discuss server provisioning methods that take into account of these two hard limits.

Let $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$ denote the total login rate and total number of connections at any given time t. Ideally we would calculate the number of servers needed as

$$K(t) = \max\left\{\left\lceil \frac{L_{\text{tot}}(t)}{L_{\max}} \right\rceil, \left\lceil \frac{N_{\text{tot}}(t)}{N_{\max}} \right\rceil\right\}. \quad (2)$$

Here the notation $\lceil x \rceil$ denotes the smallest integer that is larger than or equal to x. The number $K(t)$ is calculated on a regular basis, for example, every half an hour.

There are two problems with this simple formula. First, $K(t)$ usually needs to be calculated ahead of time based on forecasted values of $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$, which can be inaccurate. This is especially the case if we need to turn on (cool start) servers to accommodate anticipated load increase. The lead time from starting the server to getting it ready is considered significant, with new logins arriving at a fast rate. Waking up servers in stand-by mode takes less time, but uncertainty still exists because of short-period fluctuation of the load.

Another problem of this simple calculation is less obvious but more critical. Using Equation (2) assumes that we can easily dispatch load to fill each server's capacity, say, almost instantaneously. However, this is not the case here because of the dynamic relationship between login rate and number of connections. For example, when a new server is turned on, we cannot expect it to take the full capacity of $N_{\max}$ connections in short time. The number of connections on a server can only increase gradually, constrained by the bound $L_{\max}$ on login rate. (For Messenger connection servers, it takes more than an hour to fill a server from empty.) The dynamic behavior of the system, when coupled with a load dispatching algorithm, requires additional margin in calculating $K(t)$.

A natural idea to fix both problems is to add extra margins in server provisioning:

$$K(t) = \max\left\{\left\lceil \gamma_L \frac{L_{\text{tot}}(t)}{L_{\max}} \right\rceil, \left\lceil \gamma_N \frac{N_{\text{tot}}(t)}{N_{\max}} \right\rceil\right\}, \quad (3)$$

where the multiplicative factors satisfy $\gamma_L > 1$ and $\gamma_N > 1$. It remains the problem of how to determine these two factors. If they are chosen too big, we have over provisioning, which leads to inefficiency in saving energy; if they are chosen too small, we end up with under provisioning, which compromises quality of service.

Choosing the right margin factors requires careful evaluation of the forecasting accuracy, as well as detailed analysis of the dynamic behavior of the load dispatching algorithm. We will split the factors as

$$\gamma_L = \gamma_L^{\text{frc}}\gamma_L^{\text{dyn}}, \qquad \gamma_N = \gamma_N^{\text{frc}}\gamma_N^{\text{dyn}}$$

where the superscript "frc" denotes forecasting factors that are used to compensate for forecasting errors; the superscript "dyn" denotes dynamic factors to compensate for dynamic behaviors caused by the load dispatching algorithm used. In the rest of this section, we focus on determining the forecasting factors. The dynamic factors will be discussed in detail in Section 5, along with the description of different load dispatching algorithms.

4.1 Hysteresis-based provisioning

We first consider a simple provisioning method based on hysteresis switching, without explicit forecasting.

At the beginning of each scheduling interval, say time t, we need to calculate $K(t+1)$, the number of servers that will be needed at time $t+1$, and schedule servers to be turned on or off accordingly. The information we have at time t are the observed values of $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$. Using Equation (2) (or (3), but only with the dynamic factors), we can estimate the number of servers needed at time t. Call this estimated number $\hat{K}(t)$. By comparing it with the actual number of active servers $K(t)$, we determine $K(t+1)$ as follows:

$$K(t{+}1) = \begin{cases} K(t), & \text{if } \gamma_{\text{low}}\hat{K}(t) \le K(t) \le \gamma_{\text{high}}\hat{K}(t) \\ \left\lceil \frac{1}{2}\left(\gamma_{\text{low}} + \gamma_{\text{high}}\right)\hat{K}(t) \right\rceil, & \text{otherwise.} \end{cases}$$

Here the two hysteresis margin factors γ_{low} and γ_{high} satisfy $\gamma_{\text{high}} > \gamma_{\text{low}} > 1$.

This method does not use load forecasting explicitly. However, in order to choose appropriate values for the two hysteresis parameters γ_{low} and γ_{high}, we need to have good estimate of how fast the load ramps up and down based on historical data. This method is especially useful when the load variations are hard to predict, for example, when special events happen or for holidays that we do not have enough historical data to give accurate forecast.

4.2 Forecast-based provisioning

The load of connection servers demonstrates a so-called seasonal characteristic that is common for many Internet services, electrical power networks, and many economic time series. In particular, it has periodic components in days, weeks, and even months, as well as a long-term growth trend. For the purpose of server provisioning, it

suffices to consider short-term load forecasting — forecasting over a period from half an hour to several hours. (Midterm forecasting is for days and weeks, and long-term forecasting is for months and years.)

There is extensive literature on short-term load forecasting of seasonal time series (see, e.g., [12, 2]). We derive a very simple and intuitive algorithm in Section 4.3, which works extremely well for the connection server data. To the best of our knowledge, it is a new addition to the literature of short-term load forecasting. On the other hand, however, the following discussion applies no matter which forecasting algorithm is used, as long as its associated forecast factors are determined, as we will do in Section 4.3 for our algorithm.

If the forecasting algorithm anticipates increased load $L_{\rm tot}(t+1)$ and $N_{\rm tot}(t+1)$ at time $t+1$, we can simply determine $K(t+1)$, the number of servers needed, using equation (3). The only thing we need to make sure is that new servers are turned on early enough to take the increased load. This usually is not a problem, for example, if we do forecasting over half an hour into the future.

More subtleties are involved in turning off servers. If the forecasted load will decrease, we will need less number of servers. Simply turning off one or more servers that are fully loaded will cause a sudden burst of SIDs. When disconnected users try to re-login at almost the same time, an artificial surge of login requests is created, which will stress the remaining active servers. When the new login requests on the remaining servers exceed $L_{\max}$, SNA errors will be generated.

A better alternative is to schedule draining before turning a server off. More specifically, the dispatcher identifies servers that have the least amount of connections, and schedules them to connect to other servers at a controlled much slower pace that will not generate any significant burden for remaining active servers.

In order to reduce the number of SIDs, we can also starve the servers (simply not feeding it any new logins) for a period of time before doing scheduled draining or shutting it down. For Messenger servers, the natural departure rate caused by normal user logoffs results in an exponential decay of the number of connections, with a time constant slightly less than an hour, meaning that the number of connections on a server decreases by half every hour. A two-hour starving time leads to number of SIDs less than a quarter of that without starving. The trade-off is that adding starving time reduces efficiency in saving energy.

4.3 Short-term load forecasting

Now we present our method for short-term load forecasting. Let $y(t)$ be the stochastic periodic time series under consideration, with a specified time unit. It can represent $L_{\rm tot}(t)$ or $N_{\rm tot}(t)$ measured at regular time intervals. Suppose the periodic component has a period of T time units. We express the value of $y(t)$ in terms of all previous measurements as

$$y(t) \;=\; \sum_{k=1}^{n} a_k y(t-kT) + \sum_{j=1}^{m} b_j \Delta y(t-j),$$

$$\Delta y(t-j) \;=\; y(t-j) - \frac{1}{n}\sum_{k=1}^{n} y(t-j-kT).$$

There are two parts in the above model. The part with parameters a_k does periodic prediction — it is an autoregression model for the value of y over a period of T. The assumption is that the values of y at every T steps are highly correlated. The part with parameters b_j gives local adjustment, meaning that we also consider correlations between $y(t)$ and the values immediately before it. The integers n and m are their orders, respectively. We call this a SPAR (Sparse Periodic Auto-Regression) model. It can be easily extended to one with multiple periodic components.

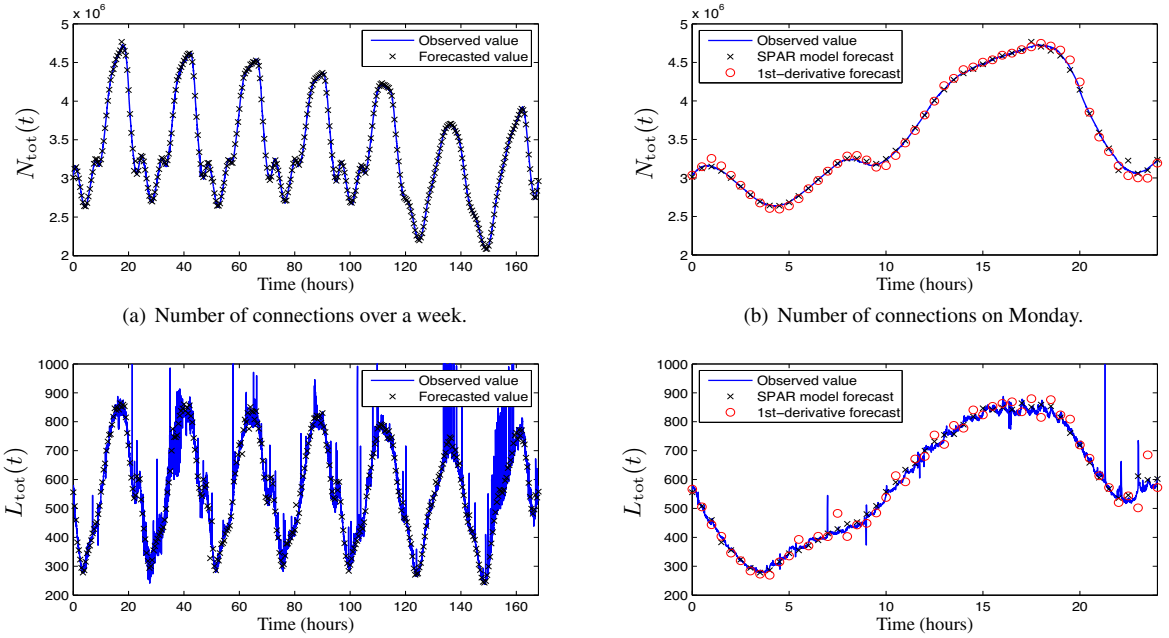
For the examples we will consider, short-term forecasting is done over half an hour, and we let the period T be a week, which leads to $T = 7 \times 24 \times 2 = 168$ samples. In this case, the autoregression part (with parameters a_k) of the SPAR model means, for example, the load at 9am this Tuesday is highly correlated and can be well predicted from the loads at 9am of previous Tuesdays. The local adjustment part (with parameters b_j) reflects the immediate trends predicted from values at 8:30am, 8:00am, and so on, on the same day.

We have tried several models with different orders n and m, but found that the coefficients a_k, b_j are very small for $k > 4$ and $j > 2$, and ignoring them does not reduce the forecasting accuracy by much. So we choose the orders $n = 4$ and $m = 2$ to do load forecasting in our examples. We used five weeks of data to estimate the parameters a_k and b_j (by solving a simple least-squares fitting problem). Then we use these data and estimated parameters to forecast loads for the following weeks.

Figure 5 shows the results of using this forecasting model. The figures show the forecasted values (every 30 minutes) plotted against actually observations (measured every 30 seconds). Note that the observed login rates (measured every second and recorded every 30 seconds) appear to be very noisy and have lots of spikes. Using this model, we can reasonably forecast the smooth trend of the curve. The spikes are hard to predict, because they are mostly caused by irregular server unavailability or crashes that result in re-login bursts.

We computed the standard deviations of the relative errors $(\hat{L}(t) - L(t))/L(t)$ and $(\hat{N}(t) - N(t))/N(t)$:

$$\sigma_L = 0.039, \qquad \sigma_N = 0.006.$$



(a) Number of connections over a week.

(b) Number of connections on Monday.

(c) Login rates over a week.

(d) Login rates on Monday.

Figure 5: Short-term load forecasting.

We also compared it with some simple strategies for short-term load forecasting without using periodic time series models. For example, one idea is to predict the load at the next observation point based on the first-order derivative of the current load and the average load at the previous observation point (e.g., Heath et al. [14]). This approach leads to the standard deviations $\sigma_L = 0.079$ and $\sigma_N = 0.012$, which are much worse than the results of our SPAR model (see Figure 5 (b) and (d)). We expect that such simple heuristics would work well for smooth trend and very short forecasting intervals. Here our time series can be very noisy (especially the login rates), and a longer forecasting interval is prefered because we do not want to turn on and off servers too frequently. In particular, the SPAR model will work reasonably well for forecasting intervals of a couple of hours (and even longer), for which derivative-based heuristics will no longer make sense.

Given the standard deviations of the forecasting errors using the SPAR model, we can assign the forecasting factors as

$$\begin{aligned} \gamma_L^{\rm frc} &= 1 + 3\sigma_L \approx 1.12, \\ \gamma_N^{\rm frc} &= 1 + 3\sigma_N \approx 1.02. \end{aligned} \tag{4}$$

These forecast factors will be substituted into Equation (3) to determine the number of servers required. To do so, we also need to specify a load dispatching algorithm and its associated dynamic factors, which we explain in the next section.

5 Load Dispatching Algorithms

In this section, we present algorithms that decide how large a share of the incoming login requests should be given to each server. We describe two different types of algorithms — load balancing and load skewing, and determine their corresponding dynamic factors $\gamma_L^{\rm dyn}$ and $\gamma_N^{\rm dyn}$. These two algorithms lead to different load distributions on the active servers, and have different implications in terms of energy saving and number of SIDs. In order to present them, we first need to establish a dynamic model of the load-dispatching system.

5.1 Dynamic system modeling

We consider a discrete-time model, where t denotes time with a specified unit. The time unit here is usually much smaller than the one used for load forecasting; for example, it is usually on the order of a few seconds. Let $K(t)$ be the number of active servers during the interval between time t and $t+1$. Let $N_i(t)$ denote the number of connections on server i at time t, and $L_i(t)$ and $D_i(t)$ be the number of logins and departures, respectively, between time t and $t+1$ (see Figure 5.1). The dynamics of the individual servers can be expressed as

$$N_i(t+1) = N_i(t) + L_i(t) - D_i(t)$$

for $i = 1, \ldots, K(t)$. This first-order difference equation captures the integration relationship between the login

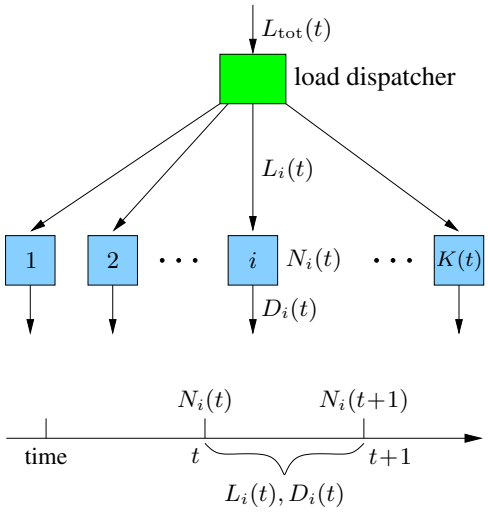


Figure 6: Load balancing of connection servers.

rates $L_i(t)$ and the number of connections $N_i(t)$. The number of departures $D_i(t)$ usually is a fraction of $N_i(t)$, which varies a lot from time to time.

The job of the dispatcher is to dispatch the total incoming login request, $L_{\text{tot}}(t)$, to the available $K(t)$ servers. In other words, it determines $L_i(t)$ for each server i. In general, a dispatching algorithm can be expressed as

$$L_i(t) = L_{\text{tot}}(t)p_i(t), \qquad i = 1, \ldots, K(t)$$

where $p_i(t)$ is the portion or fraction of the total login requests assigned to the server i (with $\sum_i p_i(t) = 1$). For a randomized algorithm, $p_i(t)$ stands for the probabilities with which the dispatcher distributes the load.

5.2 Load balancing

Load balancing algorithms try to make the numbers of connections on the servers the same, or as close as possible. The simple method of round-Robin load-balancing, i.e., always letting $p_i(t) = 1/K(t)$, apparently does not work here. By setting a uniform login rates for all the servers, regardless of the fluctuations of departure rates on individual servers (an open-loop strategy), it leaves the number of connections on the servers diverge without proper feedback control.

There are many ways to make load balancing work for such a system. For example, one effective heuristic is to apply round-Robin only to a fraction of the servers that have relatively small number of connections. In this paper we describe a proportional load-balancing algorithm, where the dispatcher assigns the following portion of total loads to server i:

$$p_i(t) = \frac{1}{K(t)} + \alpha\left(\frac{1}{K(t)} - \frac{N_i(t)}{N_{\text{tot}}(t)}\right) \qquad (5)$$

where $\alpha > 0$ is a parameter that can be tuned to influence the dynamic behavior of the system. Intuitively, this algorithm assigns larger portions to servers with relatively small number of connections, and smaller portions to servers with relatively large number of connections. Using the fact $N_{\text{tot}}(t) = \sum_{i=1}^{K(t)} N_i(t)$, we always have $\sum_{i=1}^{K(t)} p_i(t) = 1$. However, we notice that $p_i(t)$ can be negative. In this case, the dispatcher actually take load off from server i instead of assigning new load to it. This can be done by either migrating connections internally, or by going through the loop of disconnecting some users and automatic re-login onto other servers.

This algorithm leads to a very interesting property of the system: every server has the same closed-loop dynamics, only with different initial conditions. All the servers will behave exactly the same as time goes on. The only exceptions are the newly turned-on servers which have zero initial number of connections. Turning off servers does not affect others if the load are reconnected according to the same proportional rule in Equation (5). Detailed analysis of the properties of this algorithm is given in [4].

Now let's examine carefully the effect of the parameter α. For small values of α, the algorithm maintains relatively uniform login rates to all the servers, so it can be very slow in driving the number of connections to uniform. The extreme case of $\alpha = 0$ (which is disallowed here) would correspond to round-Robin. For large values of α, the algorithm tries to drive the number of connections quickly to uniform, by relying on disparate login rates across servers. In terms of determining γ_L^{dyn}, we note that the highest login rate is always assigned to newly turned-on servers with $N_i(t) = 0$. In this case, $p_i(t) = (1+\alpha)/K(t)$. By requiring $p_i(t)L_{\text{tot}}(t) \le L_{\max}$, we obtain $K(t) \ge (1+\alpha)L_{\text{tot}}(t)/L_{\max}$. Comparing with Equation (3), we have $\gamma_L^{\text{dyn}} = 1 + \alpha$.

The determination of γ_N^{dyn} is more involved. The details are omitted due to space constraints (but can be found in [4]). Here we simply list the two factors:

$$\gamma_L^{\text{dyn}} = 1 + \alpha, \qquad \gamma_N^{\text{dyn}} = \frac{1+\alpha}{r+\alpha} \qquad (6)$$

where $r = \min_t D_{\text{tot}}(t)/L_{\text{tot}}(t)$, the minimum ratio among any time t between $D_{\text{tot}}(t)$ and $L_{\text{tot}}(t)$ (the total departures and total logins between time t and $t+1$). In practice, r is estimated based on historical data.

In summary, tuning the parameter α allows us to trade-off the two terms appearing in the formula (3) for determining number of servers needed. Ideally, we shall choose an α that makes the two terms approximately equal for typical ranges of $L_{\text{tot}}(t)$ and $N_{\text{tot}}(t)$, that is, makes the constraints tight simultaneously for both maximum login rate and maximum number of connections.

5.3 Load skewing

The principle of load skewing is exactly the opposite of load balancing. Here new login requests are routed to busy servers as long as the servers can handle them. The goal is to maintain a small number of tail servers that have small number of connections . When user login requests ramp up, these servers will be used as reserve to handle login increases and surge, and give time for new servers to be turned on. When user login requests ramp down, these servers can be slowly drained and shut down. Since only tail servers are shut down, the number of SIDs can be greatly reduced, and no artificial surge of re-login requests or connection migrations will be created.

There are many possibilities to do load skewing. Here we describe a very simple scheme. In addition to the hard bound $N_{\max}$ on the number of connections a server can take, we specify a target number of connections $N_{\rm tgt}$, which is slightly smaller than $N_{\max}$. When dispatching new login requests, servers with loads that are smaller than $N_{\rm tgt}$ and closest to $N_{\rm tgt}$ are given priority. Once a server's number of connections reaches $N_{\rm tgt}$, it will not be assigned new connections for a while, until it drops again below $N_{\rm tgt}$ due to gradual user departures.

More specifically, let $0 < \rho < 1$ be a give parameter. At each time t, the dispatcher always distributes new connections evenly (round-Robin) to a fraction ρ of all the available servers. Let $K(t)$ be the number of servers available, it will choose the $\lceil \rho K(t) \rceil$ servers in the following way. First, the dispatcher partitions the set of servers $\{1, 2, \ldots, K(t)\}$ into two subsets:

$$\begin{aligned} I_{\rm low}(t) &= \{i \mid N_i(t) < N_{\rm tgt}\} \\ I_{\rm high}(t) &= \{i \mid N_i(t) \geq N_{\rm tgt}\} \end{aligned}$$

Then it chooses the top $\lceil \rho K(t) \rceil$ servers (those with the highest number of connections) in $I_{\rm low}(t)$. If the number of servers in $I_{\rm low}(t)$ is less than $\lceil \rho K(t) \rceil$, the dispatcher has two choices. It can either distribute load evenly only to servers in $I_{\rm low}(t)$, or it can include the bottom $\lceil \rho K(t) \rceil - |I_{\rm low}(t)|$ servers in $I_{\rm high}(t)$. In the second case, the number $N_{\rm tgt}$ is set further away from $N_{\max}$ to avoid number of connections exceeding $N_{\max}$ (i.e., SNA errors) within a short time.

This algorithm will lead to a skewed load distribution across the available servers. Most of the active servers should have number of connections close to $N_{\rm tgt}$, except a small number of tail servers. Let the desired number of tail servers be $K_{\rm tail}$. The dynamic factors for this algorithm can be easily determined as

$$\gamma_L^{\rm dyn} = \frac{1}{\rho}, \qquad \gamma_N^{\rm dyn} = 1 + \frac{K_{\rm tail}}{\min_t N_{\rm tot}(t)/N_{\rm tgt}}. \quad (7)$$

These factors can be substituted into equation (3) to calculate the number of servers needed $K(t)$, where it can be combined with either hysteresis-based or forecast-based server provisioning.

5.3.1 Reactive load skewing

The load skewing algorithm is especially suitable to reduce the number of SIDs when turning off servers. To best utilize load skewing, we also develop a heuristic called reactive load skewing (RLS). In particular, it is a hysteresis rule to control the number of tail servers. For this purpose, we need to specify another number $N_{\rm tail}$. Servers with number of connections less than $N_{\rm tail}$ are called tail servers. Let $K_{\rm tail}(t)$ be the number of tail servers, and $K_{\rm low} < K_{\rm high}$ be two thresholds. If $K_{\rm tail}(t) < K_{\rm low}$, then $\lceil (K_{\rm high} - K_{\rm low})/2 \rceil - K_{\rm tail}(t)$ servers are turned on. If $K_{\rm tail}(t) > K_{\rm high}$, then $K_{\rm tail}(t) - K_{\rm high}$ servers are turned off. The tail servers have very low active connections, so turning off one or even several of them will not create artificial reconnection spike. This on-off policy is executed at the server provisioning time scale, for example, every half an hour.

6 Evaluations

In this section, we compare the performance of different provisioning and load dispatching algorithms through simulations based on real traffic traces.

6.1 Experimental setup

We simulate a cluster of 60 connection servers with real data traces of total connected users and login rates obtained from production Messenger connection servers (as described in Section 3.5). The data traces are scaled accordingly to fit on 60 servers; see Figure 2. These 60 servers are treated as one cluster with a single dispatcher.

The server power model is measured on an HP server with two dual-core 2.88GHz Xeon processors and 4G memory running Windows Server 2003. We approximate server power consumption (in Watts) as

$$P = \begin{cases} 150 + 0.75 \times U, & \text{if active} \\ 3, & \text{if stand-by} \end{cases} \quad (8)$$

where U is the CPU utilization percentage from 0 to 100 (see Figure 3). The CPU utilization is modeled using the relationship derived in Section 3.5, in particular equation (1). CPU utilization is bounded within 5 to 100 percent when the server is active.

The limits on connected users and login rate are set by Messenger stress testing:

$$N_{\max} = 100,000, \qquad L_{\max} = 70/\text{sec}.$$

Also through testing, the server's wake-up delay, defined as the duration from a message is sent to wake up the server till the server successfully joins the cluster, is 2

minutes. The draining speed, defined as the number of users disconnected once a server decide to shut down, is 100 connections per second. This implies that it takes about 15 minutes to drain a fully loaded server.

6.2 Forecast vs. hysteresis provisioning

We first compare the performance of no provisioning, forecast-based and hysteresis-based provisioning, with a common load balancing algorithm.

- **No provisioning with load balancing (NB).** NB uses all the 60 servers. It implements the load balancing algorithm in Section 5.2 with $\alpha = 1$.
- **Forecast provisioning with load balancing (FB).** FB uses the load balancing algorithm in Section 5.2 with $\alpha = 1$ and the load forecasting algorithm in Section 4.3. The number of servers $K(t)$ are calculated using equation (3) with the factors

  $$\gamma_L^{\text{frc}} = 1.12, \quad \gamma_N^{\text{frc}} = 1.02 \quad \text{from equation (4)},$$
  $$\gamma_L^{\text{dyn}} = 2, \quad \gamma_N^{\text{dyn}} = 1.05 \quad \text{from equation (6)}.$$

  In calculating γ_N^{dyn}, we used the estimation $r = 0.9$ obtained from historical data.
- **Hysteresis provisioning with load balancing (HB).** HB uses the same load balancing algorithm, but with the hysteresis-based provisioning method in Section 4.1. In calculating $K(t)$, it uses the same dynamic factors as FB. There is no forecast factors for HB. Instead, we tried three pairs of hysteresis margins $(\gamma_{\text{low}}, \gamma_{\text{high}})$:

  $$(1.05,\ 1.10), \quad (1.04,\ 1.08), \quad (1.04, 1.06).$$

  denoted as HB(5/10), HB(4/8) and HB(4/6).

The simulation results based on two days of real data (Monday and Tuesday in Figure 2) are listed in Table 1. The number of servers used are shown in Figure 7. These results show that forecast-based provisioning leads to slightly more energy savings than hysteresis-based provisioning. With the hysteresis margins getting tight, the difference in energy savings becomes even smaller. However, this comes with a cost of service quality degradation, as shown by the increased numbers of SNA errors caused by smaller hysteresis margins.

Algorithm	Energy (kWh)	Saving	SNA
NB ($\alpha = 1$)	478	—	0
FB ($\alpha = 1$)	331	30.8%	0
HB(5/10)	344	28.0%	0
HB(4/8)	341	28.6%	7,602
HB(4/6)	338	29.2%	512,531

Table 1: Comparison of provisioning methods. Energy savings are reduced percentages with respect to NB.

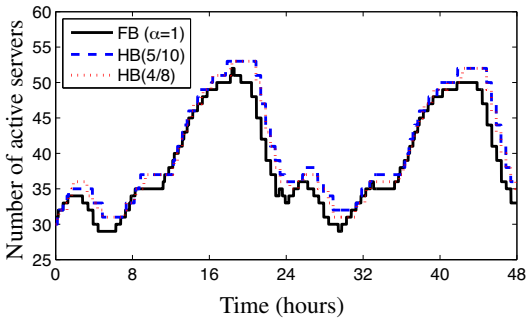


Figure 7: Number of servers used by FB and HB.

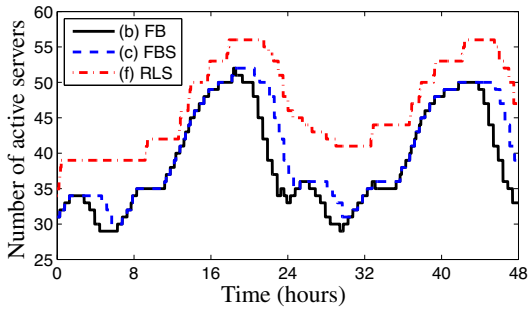
6.3 Load balancing vs. load skewing

In addition to energy saving, the number of SIDs is a another important performance metric. We evaluate both aspects on FB and the following algorithms:

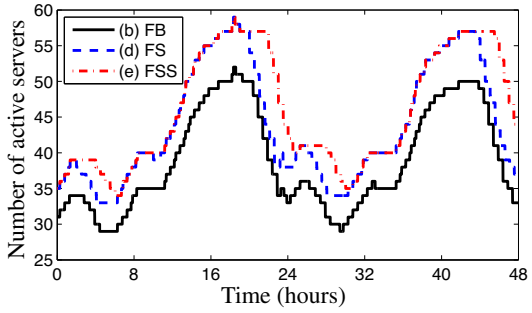
- **Forecast provisioning with load balancing and Starving (FBS).** FBS uses the same parameters as FB, with the addition of a period of starving time before turning off servers. We denote the starving time by S. For example, $S = 2$ means starving for two hours before turning off.
- **Forecast provisioning with load skewing (FS).** FS uses the same forecast provisioning method as before, combined with the load skewing algorithm in Section 5.3 with parameters $\rho = 1/2$ and $N_{\text{tgt}} = 98,000$. Its associated dynamic factors are obtained from Equation (7): $\gamma_L^{\text{dyn}} = 2$ and $\gamma_N^{\text{dyn}} = 1.2$, where $K_{\text{tail}} = 6$ is used in calculating γ_N^{dyn}.
- **Forecast provisioning with load skewing and starving (FSS).** FSS uses the same algorithms and parameters as FS, with the addition of a starving time S hours before turning off servers.
- **Reactive load skewing (RLS).** RLS uses the load skewing algorithm with the same ρ and N_{tgt} as FS. Instead of load forecasting, it uses the hysteresis on-off scheme in Section 5.3.1 with parameters $N_{\text{tail}} = N_{\max}/10 = 10,000$, $K_{\text{low}} = 2$ and $K_{\text{high}} = 6$.

Simulation results of some typical scenarios, labeled as (a),(b),(c),(d),(e),(f), are shown in Table 2 and Figure 8. Comparing FBS with FB and FSS with FS, we see that adding a two-hour starving time before turning off servers leads to significant reduction in the number of SIDs, and mild increase in energy consumption.

The results in Table 2 show a clear tradeoff between energy consumption and number of SIDs. With the same amount of starving time, load balancing uses less energy but generates more SIDs, and load skewing uses more energy but generates less SIDs. In particular, load skewing without starving has less number of SIDs than load



(a) Number of servers used by FB, FBS and RLS.



(b) Number of servers used by FB, FS and FSS.

Figure 8: Number of servers by different algorithms.

balancing with starving for two hours. RLS with a relatively small N_{tail} (say around $10,000$) has less number of SIDs even without starving. To give a better perspective of the SID numbers, we note that the total number of logins during these two days is over 100 millions (again, this is the number scaled to 60 servers).

6.3.1 Load profiles

To give more insight into different algorithms, we show their load profiles in Figure 9. Each vertical cut through the figures represents the load distribution (sorted number of connections) across the 60 servers at a particular time. For NB, all the loads are evenly distributed on the 60 servers, so each vertical cut has uniform load distribution, and they vary together to follow the total load pattern. Other algorithms use server provisioning to save energy, so each vertical cut has a drop to zero at the number of servers they use. The contours of different number of connections are plotted. The highest contour curves show the numbers of servers used (those with nonzero connections), cf. Figure 8.

For FB, the contour lines are very dense (sudden drop to zero), especially when the total load is ramping down and servers need to be turned off. This means servers with almost full loads have to be turned off, which causes lots of SIDs. Adding starving time makes the contour lines of FBS sparser, which corresponds to reduced number of SIDs. Load skewing algorithms (FS, FSS and RLS) intentionally create tail servers, which makes the contour lines much sparser. While they consume a bit more energy, the number of SIDs are dramatically reduced by turning off only tail servers.

	Algorithms and Parameters	Energy (kWh)	Saving	Number of SIDs
(a)	NB	478	—	0
(b)	FB ($\alpha=1$)	331	30.8%	3,711,680
(c)	FBS ($\alpha=1, S=2$)	343	28.2%	799,120
(d)	FS ($\rho=0.5$)	367	23.3%	597,520
(e)	FSS ($\rho=0.5, S=2$)	381	20.2%	115,360
(f)	RLS ($\rho=0.5$, $N_{\text{tail}}=10,000$)	375	21.5%	48,160

Table 2: Comparison of load dispatching algorithms. All algorithms in this table have zero SNA errors.

6.3.2 Energy-SID tradeoffs

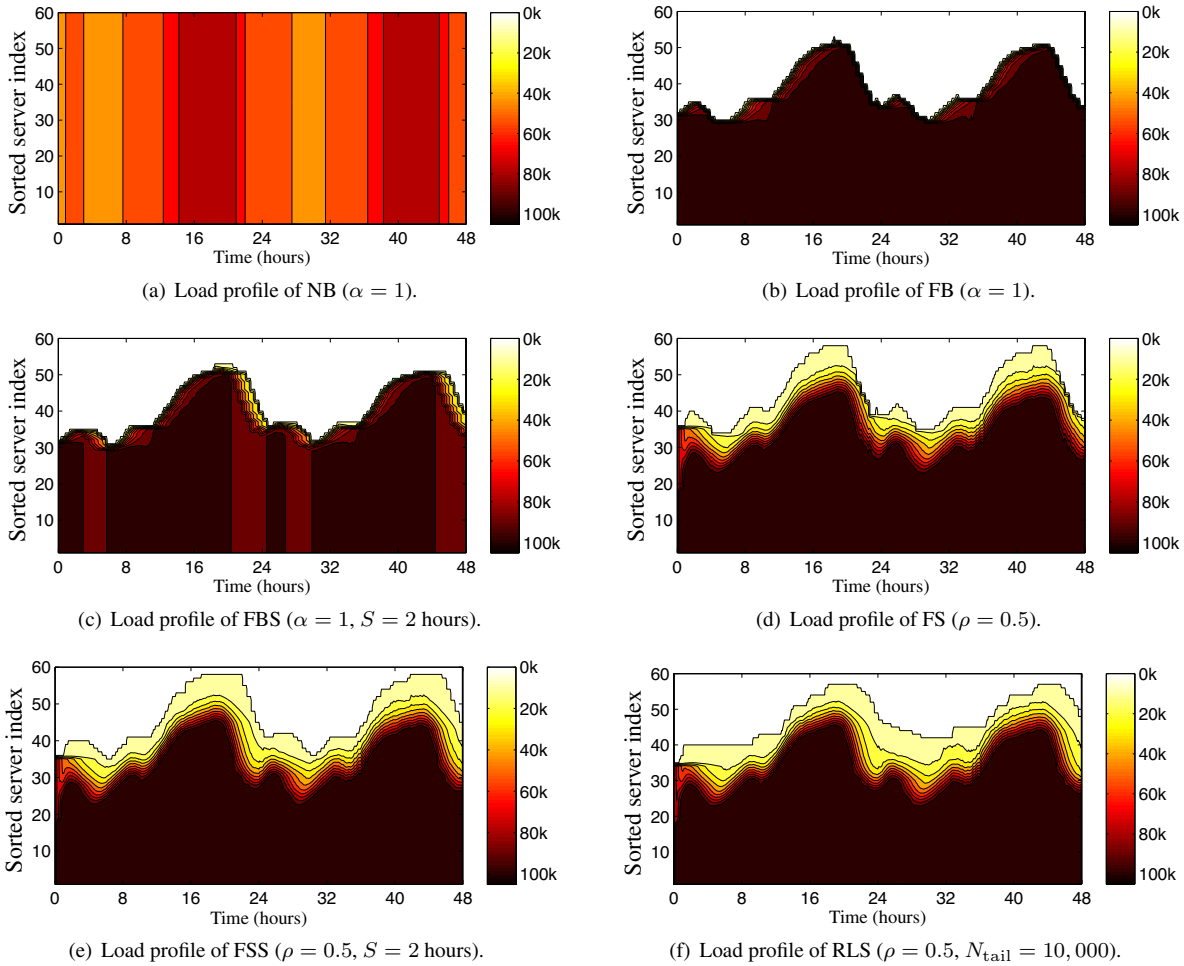
To unveil the complete picture of the energy-SID tradeoffs, we did extensive simulation of different algorithms by varying their key parameters.

Not all possible parameter variations give meaningful results. For example, if we choose ρ too small for FSS or RLS (e.g., $\rho \leq 0.4$ for this particular data trace), significant amount of SNA errors will occur because small ρ limits the cluster's capability of taking high login rates. The number of SNA errors will also increase significantly if we set $N_{\text{tail}} \geq 40,000$ in RLS. For fair comparison, all scenarios shown in Figure 10 give less than 1000 SNA errors (due to rare spikes in login rates).

For FBS, the three curves correspond to the parameters $\alpha = 0.2$, 1.0, and 3.0. Each curve is generated by varying the starving time S from 0 to 8 hours, evaluated at every half-an-hour increment (labeled as crosses). Within the figure, it only shows parts of the curves to allow better visualization of other curves. The right-most crosses on each curve correspond to $S = 2$ hours, and the number of SIDs decreases as S increases.

For FSS, the three curves correspond to the parameters $\rho = 0.4, 0.5, 0.9$. Each curve is generated by varying the starving time S from 0 to 5 hours, evaluated at every half an hour (labeled as triangles). For example, scenario (d) in Table 2 is the right-most symbol on the curve $\rho = 0.5$. The plots for FBS and FSS show that the number of SIDs roughly decreases by half for every hour of starving time (every two symbols on the curves).

For RLS, the three curves correspond to the parameters $\rho = 0.4$, 0.5, and 0.6. Each curve is generated by varying the threshold for tail servers N_{tail} from 1000 to $40,000$ (labeled as circles). The number of SIDs increases as N_{tail} increases (the right-most circles are for



(a) Load profile of NB ($\alpha = 1$).

(b) Load profile of FB ($\alpha = 1$).

(c) Load profile of FBS ($\alpha = 1$, $S = 2$ hours).

(d) Load profile of FS ($\rho = 0.5$).

(e) Load profile of FSS ($\rho = 0.5$, $S = 2$ hours).

(f) Load profile of RLS ($\rho = 0.5$, $N_{\text{tail}} = 10,000$).

Figure 9: Load profile of different server scheduling and load dispatching algorithms shown in Table 2.

N_{tail}=40, 000). Decreasing the threshold N_{tail} for RLS is roughly equivalent to increasing starving time S for FBS and FSS.

In Figure 10, points near the bottom-left corner have the desired property of both low energy and low number of SIDs. In this perspective, FSS can be completely dominated by both FBS and RLS *if their parameters are tuned appropriately*. For FBS, it takes a much longer starving time than FSS to reach the same number of SIDs, but the energy consumption can be much less if the value of α is chosen around 1.0. Between FBS and RLS, they have their own sweet spots of operation. For this particular data trace, FBS with $\alpha = 1$ and 4 to 5 hours of starving time give excellent energy-SID tradeoff.

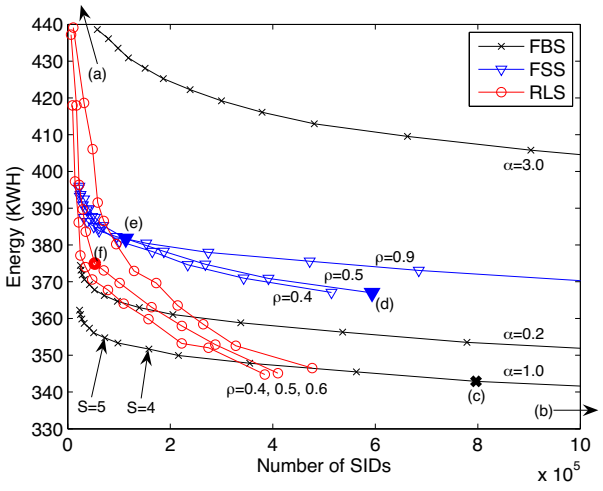


Figure 10: Energy-SID tradeoff of different algorithms with key arameters varied. The particular scenarios (c),(d),(e),(f) listed in Table 2 are labeled with bold symbols. The scenarios (a) and (b) are out of scope of this figure, but their relative locations are pointed by arrows.

7 Discussions

From the simulation results in Section 6, we see that while load skewing algorithms generate small number of SIDs when turning off servers, they also maintain unnec-

essary tail servers when the load ramps up. So a hybrid (switching) algorithm that employs load balancing when load increases and load-skewing when load decreases seems to be able to get the best energy-SID tradeoff. This is what the FBS algorithm tries to do with a long starving time, and its effectiveness is clearly seen in Figure 10. Of course, there are regions in the energy-SID tradeoff plane that favor RLS more than FBS. This indicates that there are still room to improve by explicitly switching between load balancing and load skewing, for example, when the total number of connections reaches the peak and starts to go down.

As mentioned in Section 3.4, SIDs can be handled transparently by the client software so that they are unnoticeable to users, and the servers can be scheduled to drain slowly in order to avoid creating a surge of reconnection requests. The transitions can also be done through "controlled connection migration" (CCM) — to migrate the TCP connection endpoint state without breaking it. Depending on the implementation details, CCM may also be a CPU- or networking-intensive activity. The number of CCMs could be a performance metric similar to the number of SIDs, which we trade off with energy saving. Depending on the cost of CCMs, they might change the sweet spots on the energy-SID/CCM tradeoff plane. If the cost is low, due to a perfect connection migration scheme, then we can be more aggressive on energy saving. On the other hand, if the cost is high, we have to be more conservative, as in dealing with SIDs. We believe the analysis and algorithmic framework we developed would still apply.

We end the discussions with some practical considerations of implementing the dynamic provisioning strategy at full scale. Implementing the core algorithms is relatively straightforward. However, we need to add some safeguarding outer loops to ensure they work in their comfort zone. For example, if the load-forecasting algorithm starts giving large prediction errors (e.g., when there is not enough historical data to produce accurate model for special periods such as holidays), we need to switch to the more reactive hysteresis-based provisioning algorithm. To take the best advantage of both load balancing and load skewing, we need a simple yet robust mechanism to detect the up and down trends in the total load pattern. On the hardware side, frequently turning on and off servers may raise reliability concern. We want to avoid always turning on and off the same machines. This is where load prediction can help by looking ahead and avoiding short term decisions. A better solution is to rotate servers in and out of the active clusters deliberately.

Acknowledgments

We thank the Messenger team for their support, and Jeremy Elson and Jon Howell for helpful discussions.

References

[1] ABDELZAHER, T. F., SHIN, K. G., AND BHATTI, N. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Trans. Parallel Distrib. Syst.*, 1 (2002), 80 – 96.

[2] BROCKWELL, P. J., AND DAVIS, R. A. *Introduction to Time Series and Forecasting*, second ed. Springer, 2002.

[3] CHASE, J., ANDERSON, D., THAKAR, P., VAHDAT, A., AND DOYLE, R. Managing Energy and Server Resources in Hosting Centers. In *SOSP* (2001).

[4] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. Tech. rep., Microsoft Research, 2007.

[5] CHEN, Y., DAS, A., QIN, W., SIVASUBRAMANIAM, A., WANG, Q., AND GAUTAM, N. Managing server energy and operational costs in hosting centers. In *In Proceedings of the International Conference on Measurement and Modeling of Computer Systems* (2005).

[6] DOYLE, R., CHASE, J., ASAD, O., JIN, W., AND VAHDAT, A. Model-Based Resource Provisioning in a Web Service Utility. In *In Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems* (2003).

[7] ELNOZAHY, M., KISTLER, M., AND RAJAMONY, R. Energy conservation policies for web servers. In *USITS* (2003).

[8] EPA Report on Server and Data Center Energy Efficiency. U.S. Environmental Protection Agency, ENERGY STAR Program, 2007.

[9] FAN, X., WEBER, W.-D., AND BARROSO, L. A. Power Provisioning for a Warehouse-sized Computer. In *ISCA* (2007).

[10] FLAUTNER, K., AND MUDGE, T. Vertigo: Automatic Performance-Setting for Linux. In *OSDI* (2002).

[11] GANESH, L., WEATHERSPOON, H., BALAKRISHNAN, M., AND BIRMAN, K. Optimizing power consumption in large scale storage systems. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS '07)* (2007).

[12] GROSS, G., AND GALIANA, F. D. Short-term load forecasting. *Proceedings of The IEEE 75*, 12 (1987), 1558–1573.

[13] GRUNWALD, D., LEVIS, P., FARKAS, K. I., III, C. B. M., AND NEUFELD, M. Policies for Dynamic Clock Scheduling. In *OSDI* (2000).

[14] HEATH, T., DINIZ, B., CARRERA, E. V., JR., W. M., AND BIANCHINI, R. Energy conservation in heterogeneous server clusters. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* (2005).

[15] KANDASAMY, N., ABDELWAHED, S., AND HAYES, J. P. Self-optimization in computer systems via online control: Application to power management. In *IEEE International Conference on Autonomic Computing ICAC* (2004).

[16] KUSIC, D., AND KANDASAMY, N. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In *IEEE International Conference on Autonomic Computing ICAC* (2006).

[17] LORCH, J. R., AND SMITH, A. J. Improving Dynamic Voltage Scaling Algorithms with PACE. In *SIGMETRICS/Performance* (2001).

[18] MSN protocol documentation. http://msnpiki.msnfanatic.com/.

[19] M.WEISER, B.WELCH, DEMERS, A. J., AND SHENKER, S. Scheduling for Reduced CPU Energy. In *OSDI* (1994).

[20] PINHEIRO, E., BIANCHINI, R., CARRERA, E. V., AND HEATH, T. Dynamic Cluster Reconfiguration for Power and Performance. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power* (2001).

[21] SHARMA, V., THOMAS, A., ABDELZAHER, T., SKADRON, K., AND LU, Z. Power aware QoS management in web servers. In *IEEE RTSS* (2003).

[22] WANG, M., KANDASAMY, N., GUEZ, A., AND KAM, M. Distributed cooperative control for adaptive performance management. *IEEE Internet Computing 11*, 1 (2007), 31–39.

[23] ZHU, Q., CHEN, Z., TAN, L., ZHOU, Y., KEETON, K., AND WILKES, J. Hibernator: Helping Disk Arrays Sleep Through the Winter. In *SOSP* (2005).