Balanced Overlay Network (BON):
Decentralized Load Balancing via Self-Organized Random Networks

Jesse S. A. Bridgewater
University of California, Los Angeles
Department of Electrical Engineering
jsab@pobox.com

P. Oscar Boykin
University of Florida
Department of Electrical and Computer Engineering
boykin@ece.ufl.edu

Vwani P. Roychowdhury
University of California, Los Angeles
Department of Electrical Engineering
vwani@ee.ucla.edu

Abstract

We introduce a new load balancing paradigm, where the instantaneous in-degree distribution of a dynamic overlay network encodes the current level of free resources at each of the server/computing nodes. The allocation of a new job is executed by sampling the in-degree of nodes via a short random walk, and assigning the job to a node with the maximum in-degree (hence, the one with the most free resources) among the sampled nodes; the reduced free resources of the node is updated by randomly deleting an appropriate number of incoming edges. Similarly, when a job is completed at a node, it updates its status by adding a desired number of incoming edges, where the other end of the edges are again picked via short random walks. We show that this mapping of free resources to the degree distribution of an overlay network is well suited to any situation where each server provides similar service such as large-scale computing (cluster and grid computing) and web content mirroring. Extensive simulation results show that our random-walk based protocol leads to an almost optimal distribution of load: the steady state topology of the overlay network approaches random d-regular graphs. Thus, for applications where resources can be adequately represented by a scalar quantity, our load balancing protocol is highly efficient and effective. Implementing the protocol directly using lightweight sockets to maintain the overlay network should involve minimal overhead, however direct connections are not required since edges in the overlay network could be “virtual” edges that represent routes in the underlying transport layer.

1 Introduction

Since the advent of networked computers, protocols have been developed to make more efficient use of distributed computing resources. Numerous methods have been developed to maximize the use of networked computers for large-scale computing, databases and content servers to name a few. The use of selective polling, global random choice, and local diffusion on static underlying networks has produced tremendous gains in the field of load balancing [24, 18, 22, 23, 21]. A system where some of these ideas have been applied is the openMosix[9] cluster computing architecture. Systems like openMosix bring networked computers together to behave much like a virtual symmetric multiprocessor (SMP) computer with a relatively slow communication bus among the processors (network connection). As such, they are well suited to solving problems that are predominately non-communication bound, since jobs can migrate anywhere in the network and communication among jobs may be slow. For many tasks, such as optimization problems, where the overall problem naturally breaks down into small parts that can run independently, the slow communication links among the computing nodes is not a significant bottleneck. Good examples of such separable problems include SETI@home[10], a distributed computing project that analyzes radio telescope data for signs of extra terrestrial life, and Folding@home[4], a similar specialized distributed computing project that simulates protein folding to better understand diseases. Each of these special purpose distributed applications runs on hundreds of thousands of computers and unleashes tens to hundreds of TFLOPS from otherwise idle computers; our approach has the potential to scale to similar sizes but to provide general
purpose computing capabilities.

To state our approach as simply as possible, we consider a system of \(N\) comparable-capacity web servers all of which mirror the same set of contents. For efficient usage of these resources, one would want to distribute the download requests as evenly as possible, so that no server is significantly more loaded than others. The question is, can one achieve such a load balancing task, without a centralized server or some equivalent mechanism to monitor the global state of the network. Numerous load-balancing applications for web servers [25, 16, 13] based on global monitoring or global random selection have been proposed and several open source projects have formed to provide capacity and geography load-balancing [11, 12, 6].

Prior load balancing approaches have the common element that the connectivity of the network does not convey information about the state of the network; the network relies on the connectivity graph to communicate load state information and to migrate jobs to other nodes, but the graph structure of the links between nodes does not play a central role in the balancing algorithm itself.

We present a scheme fundamentally different from those proposed in the literature: instead of monitoring servers and their availability via a static network or diffusing load through a static network topology, we create a dynamic overlay network that provides both a measure of instantaneous load distribution, and dynamics for job allocation and resource update. The way we adapt our dynamical network system to the task of load balancing is as follows: First, a node’s in-degree is made to correspond to its current estimate of its free resources. Second, the edge dynamics in our system are used to perform the job allocation and resource updating tasks for the load balancing process. When a new job arrives the node receiving the job initiates a random walk and targets the highest in-degree node visited on the walk to accept the job. The target node on receiving the job removes one of its incoming edges to reflect the reduced availability of its resources. Similarly, when a node/server completes a job, and becomes less loaded, it adds an incoming edge to itself (again via a random walk, as prescribed by our edge insertion dynamic) to increase its in-degree. In steady state, the rate at which jobs arrive would equal the rate at which jobs are completed, and hence the underlying network has a fixed average number of edges.

Thus, a dynamic overlay network, connecting all the servers, emerges. The state of this network (as indexed by the in-degree distribution of the nodes) represents the instantaneous distribution of load over all the servers. The job assignment and the resource update steps, performed according to the edge deletion and insertion steps in our network dynamics, distributes the load fairly across all the servers in the network.

In particular the mapping of free resources to in-degree implies that a perfectly balanced network is a regular graph. We demonstrate in simulations that our protocol produces nearly regular graphs (figure 5) and thus provides nearly optimal load balancing performance.

This overlay architecture, as schematically represented in Fig 1 is well suited for many applications including highly scalable web server software. Every major web site uses mirrors to, among other things, balance the request load over multiple servers. This service is currently provided by companies such as Akamai which maintain proprietary overlay networks with tens of thousands of nodes and which routinely handle double-digit percentages of total Internet traffic. A system using the techniques proposed here can provide effective load balancing for Open Source software projects with hundreds of largely unbalanced mirrors and other organizations seeking alternate mirroring solutions. Good examples of such projects include the Linux
So far we have implicitly assumed that any node can directly connect to any other node in order to form the connections that represent free resources. This assumption is not necessary and it has been made for the sake of presentational clarity. In cases where not all parts of a network are directly mutually addressable, the "connections" could be stored as paths in the underlying backbone physical network. The job allocation and resource update steps will operate as usual on this cached virtual network.

The structure of this paper is as follows: in section 2 we describe the algorithm, in section 3 we show via simulations that our algorithm performs almost optimally and in section 4 we discuss implementation of this scheme for an HTTP mirror system.

2 Protocol Description

Here we describe in detail the edge dynamic protocol for creating and maintaining the balanced overlay network and the mapping between graph properties and node resources.

2.1 Protocol Details

The following are important properties of the protocol:
(i) Each node represents a server or processor providing service to a networked community.
(ii) The in-degree of a node represents the amount of free resources of the particular node, e.g., the number of extra jobs in can handle. In practice, the free resources will be a measure of what performance the next arriving job can expect to experience.
(iii) The maximum in-degree, $C$, is the maximum capacity that each node in the network can handle.
(iv) The state of the network, i.e., the in-degree distribution, represents how balanced the load distribution is.

In the steady state, jobs/requests arrive at the same rate as the jobs are completed by the suite of servers. Hence, when a job arrives, in our representation, we need to decrease the in-degree of the node to which the job is assigned. Similarly, when a job is completed the corresponding server may have more resources and it should indicate its new state by increasing its in-degree.

The outline of the steps is summarized below.

- Create a graph $G$ whose nodes have in-degree proportional to free resources.
- When new load arrives at node $v_i$, $v_i$ performs a short random-walk on $G$ and distributes the load to the node on the walk that has the largest in-degree.
- Nodes compensate for changes in load by adding or removing edges in order to keep in-degree proportional to free resources.

The random walks that are used in the above operations are taken to be at least of length $O(\log N)$. This is due to the fact that a random $d$-regular undirected graph has mixing time (i.e., the length of the random walk necessary to sample edges uniformly) that scales as $O(\log N)$, where $N$ is the number of nodes in the network $\mathbb{G}$. Although the graphs under consideration here are directed graphs, we observe that they also have diameters (in the directed sense) of $O(\log N)$.

2.2 Network-resources mapping

The above-mentioned dynamics create and maintain the structure of the overlay network through the deletion and creation of edges which represent resource capacity. Mapping the properties of this overlay network to node resources can be handled easily by making the natural assumption that there is a scalar metric, $R$, that each node can locally calculate to determine its free resources. In the example of a web mirroring application of the protocol, the relevant metric is the bandwidth that the next request can expect to receive. It would be calculated as the peak outgoing bandwidth $B_i^{(\text{max})}$ divided by the current number of requests $D_i$ plus 1: $R_i = \frac{B_i^{(\text{max})}}{D_i + 1}$. The nodes agree on a size for the unit of capacity represented by an in-degree. Using this unit of capacity, each node maintains a targeted in-degree proportional to free resources:

$$k_i(t) = \max\left(\frac{C}{D_i}, k_i^{(\text{min})}\right)$$  \hspace{1cm} (1)

Given that the distribution of resources in a balanced overlay network maps onto the graph’s in-degree distribution, it is natural to use the in-degree distribution variance as a measure of load balancing performance. This brings up the question of what the in-degree distribution of a perfectly balanced network looks like. The first thought is that a perfectly balanced network will be a completely regular graph in the general case, however a zero variance is only possible if $N | 2E$ (if $N$ divides $2E$). This is clear since only under that condition can every node in the graph have the same in-degree. In networks that have dynamic load patterns this ideal case will not occur in practice.

Another way to view this issue is to realize that the average degree can be an integer or it can be between two integers. A perfectly balanced network whose mean in-degree is an integer will have a variance of zero; this is clear since every node has exactly the same in-degree. However another perfectly balanced network is one where the mean in-degree is exactly between two integers, $\langle k \rangle = \ldots$
Figure 2. Networks have different optimal values of variance depending on the average in-degree (as seen in equation [3]). Here we see clearly that zero variance is only possible when \( \langle k \rangle \) is an integer. The worst case optimal variance (0.25) is when \( \langle k \rangle \) is exactly between two integers. Given the dynamic nature of this type of network, the average in-degree will not be controllable so networks that achieve a variance of 0.25 or better may be optimally-balanced graphs. If the average degree fluctuates equally across the possible range of degrees, we can expect a perfectly balanced network to have an average variance of \( \langle \sigma^2 \rangle = 1/8 \).

\[
(k + (k + 1))/2.
\]

In this case half of the nodes have in-degree \( k \) and the other half \( k + 1 \). The variance in this case is 1/4. More generally for any \( \langle k \rangle \) the optimal variance is:

\[
\sigma^2 = \langle (k - \langle k \rangle)^2 \rangle.
\]  
(2)

This means that if \( \sigma^2 < 1/4 \) the graph is optimally balanced (since perfect balance can result when \( \sigma^2 \) is that high). The general form of the in-degree variance for a perfectly balanced network is the following:

\[
\sigma^2 = \sum_{i=k_{\min}}^{k_{\max}} P(k_i)(k_i - \langle k \rangle)^2.
\]  
(3)

For a perfectly balanced network this is a sawtooth curve as depicted in Fig. 2.

Since the mapping of resources to network structure has to be common among all nodes it is important to ask how the network can adapt and rescale the mapping of resources to network properties. Distributed mechanisms\[17\] to alter the mapping of resources to in-degree can be added to account for changes in capacity and network size. However this issue is independent of the balanced overlay protocol so we will not consider this detail here. Also as we mentioned earlier, the assumption that the overlay network edges are direct connections is not necessary, and it has been made for the sake of presentational clarity. In cases where not all parts of a network are directly mutually addressable, the "connections" could be stored as paths in the underlying backbone physical network. The job allocation and resource update steps will operate as usual on this cached

Figure 3. The equilibrium networks produced by the proposed protocol are nearly regular graphs (and nearly optimally balanced). The in-degree distribution depicted here is tightly peaked with a near-optimal variance of 0.304 which is very close to 0.25 which is the highest variance that an optimally balanced network can have (discussed in section 2). This network has \( N = 1024 \) nodes and an average degree \( \langle k \rangle = 29 \).
virtual network.

In the case of grid computing applications, the mapping would be quite similar. Existing distributed computing systems such as openMosix[9] represent computational resources by distilling all aspects of computer performance into a scalar metric or cost. This metric is then used to determine how loaded a node is. We will likely pursue a nearly identical approach for a distributed computing system.

3 Simulation Results

The simple protocol discussed above results in graphs that are very nearly d-regular as can be seen in figure 3. The mapping that the protocol creates between in-degree and free resources implies that every node in a regular graph has the same in-degree and thus has the same free resources.

All simulations are conducted in the following way. A single constant node in the network initiates all new jobs. The number of jobs that arrive at each time step is Poisson distributed. Each new job random walks through the network and migrates to the node with the largest in-degree. Furthermore the follow steps were used to simulate job traffic on the overlay network and to discover whether or not the random graphs could form from non-random initial topologies.

1. **Graph Initialization**: First we create a directed graph with \( N \) nodes and \( E = N \langle k \rangle \) edges such that the maximum degree of any node is \( \leq C \). This graph is intentionally constructed in a very structured fashion so as to show that the proposed dynamics do indeed lead to a random regular graph independent of the initial configuration. The initial structure is created by connecting node \( i \) by incoming edges to nodes \((i+1) \mod N, (i+2) \mod N, \ldots, \) and \((i+\langle k \rangle) \mod N\), where \( \langle k \rangle \) is the average degree of the nodes. The dynamics effectively randomize the initial graph and reduce initial diameter of \( O(N) \) to \( O(\log N) \).

2. **Job Initiation (Edge deletion)**: For this set of simulations, at each time step we delete a Poisson-distributed number of edges. A random walk of length \( \log N \) is initiated from a fixed particular node, and the node on the random walk with the highest degree randomly deletes one of its incoming edges. We take the case where only one node initiates jobs since it represents the most difficult case to balance using short random walks.

3. **Job Completion (Edge Insertion)**: When a job ends at node \( i \) it initiates a random walk and a directed edge from node \( j \) (the last node on the walk) to node \( i \) is added. In this step a standard random walk is used instead of the walks which choose the highest in-degree; the simple reason for this is that the out-degree distribution is not used for the algorithm and so maintaining a nearly regular out-degree is not needed. Ideally a new edge would be chosen uniformly from all of the possible absent edges that could be incident on node \( i \). The standard random walk will select nodes with degree proportional to their respective in-degrees and since the graph is nearly regular with respect to in-degree the walk will approximate choosing randomly from the entire network. In these simulations at each time step we uniformly randomly select a Poisson-distributed number of jobs to finish. Then each node where a job ended initiates a random walk as just described so that it can obtain a new edge to account for its increase in free resources.

In this protocol we use random walks of length \( \log N \) to distribute new incoming jobs and to form new connections when jobs are completed. The variance of the in-degree distribution is pretty insensitive to the walk length. For very short walks (walk length of 4 in a network with \( N = 1024 \)) we expect that the balancing will not be very effective; in order to use random walks to balancing load the walk must be at least as large as the graph diameter. For walks with only 1, 2 or 3 steps, we observe that the network forms an overloaded region near the node that is initiating all the jobs. This region stays overloaded while the rest of the network is completely unloaded.

In figure 5 we consider a fixed size network and perform with different walk lengths. We observe that increasing the walk length does reduce the in-degree variance as expected, however we see in this example that the improvements gained from longer walks than 10 are modest. This is representative of the scaling seen for larger graph sizes as well; after the walk is longer than \( \log N \) the incremental decrease in variance is small as the walk length is increased. Of course if we perform a walk of sufficient length to cover the graph then we have polled each node for its resources. However the cover time for random regular graphs can be upper bounded by \( O(N^2) \) which is far more expensive than using a global polling system. Given that the short random walk approach gets very close to an optimal variance we focus our interest on the protocol with \( O(\log N) \) walk length.

In Fig. 6 we show that the performance of the algorithm scales well over a large range of network size (\( N=2048,4096,8192,16394,32768 \)). For this algorithm we do not have analytical evidence that the protocol should generate a nearly regular graph. However this algorithm can be seen as an optimization of the random walk algorithm in [15] that is shown analytically to produce Erdős-Rényi (ER) random graphs. The protocol that generates ER graphs uses the last node of the random walk for the migration of jobs rather that the highest degree node as proposed here. This implies that nearly regular graphs generated here should have variances smaller than or equal to the ER graphs. Thus intuitively we expect that the optimized algorithm that chooses the highest degree (least loaded) should balance at least as well as the algorithm that generates ER graphs.
Figure 4. Here we see that the performance of the algorithm is consistent for a large range of network loads. Here we have 5 networks each with $k_{(max)} = 54$ and $N = 1024$. The distribution peaked at $k = 10$ depicts a network that is loaded to about 90% of its capacity, while the distribution peaked at $k = 46$ is less than 20% loaded. However we see that the in-degree variances (shown directly below the corresponding distributions are all virtually the same. This shows that the algorithm is effective for networks with different load burdens. The number of jobs that arrive and depart each time step is Poisson distributed.

Figure 4 shows that networks experiencing a variety of loads benefit from nearly identical balancing performance. Whether a network is nearly idle or over 90% utilized the in-degree distributions have very similar variance and thus the performance is consistent.

Here we show the time dynamics of the in-degree distributions of the overlay network. The graph starts out in a completely ordered state described in section 2, then becomes broad and random; eventually it settles down to a nearly random regular graph. The variance starts at 0, then explodes, then comes back to near 0. The lower part of figure 7 shows the variance of the in-degree distribution vs. time. The insets show snapshots of the in-degree distribution at three important points in the graph evolution: initial graph regular graph at $t = 0$ (left inset), maximum variance graph at $t = 1600$ (middle inset) and the nearly regular random graph at $t = 6000$.

4 Implementation Plan

We discussed how to map the resources of a network onto the balanced overlay protocol in section 2. Here we describe concrete plans for an distributed web content mirroring system using the protocol; a grid computing system is also of interest but far more complicated to implement so the web mirror system will be the focus of initial implementation efforts. We will sketch briefly how the grid computing system might be approached.

4.1 Web Mirroring Architecture

The Apache web server is a high-performance web server and is the most deployed web server in the world. The architecture of Apache allows the creation of modules that can add additional functions to the server. We propose to create a module extension to this industry standard web server.

Figure 4. Here we see that the performance of the algorithm is consistent for a large range of network loads. Here we have 5 networks each with $k_{(max)} = 54$ and $N = 1024$. The distribution peaked at $k = 10$ depicts a network that is loaded to about 90% of its capacity, while the distribution peaked at $k = 46$ is less than 20% loaded. However we see that the in-degree variances (shown directly below the corresponding distributions are all virtually the same. This shows that the algorithm is effective for networks with different load burdens. The number of jobs that arrive and depart each time step is Poisson distributed.

Figure 4 shows that networks experiencing a variety of loads benefit from nearly identical balancing performance. Whether a network is nearly idle or over 90% utilized the in-degree distributions have very similar variance and thus the performance is consistent.

Here we show the time dynamics of the in-degree distributions of the overlay network. The graph starts out in a completely ordered state described in section 2, then becomes broad and random; eventually it settles down to a nearly random regular graph. The variance starts at 0, then explodes, then comes back to near 0. The lower part of figure 7 shows the variance of the in-degree distribution vs. time. The insets show snapshots of the in-degree distribution at three important points in the graph evolution: initial graph regular graph at $t = 0$ (left inset), maximum variance graph at $t = 1600$ (middle inset) and the nearly regular random graph at $t = 6000$.
server that will enable organizations to use their existing web mirror infrastructure to form an overlay network that will transparently balance the server load without the central polling mechanisms present in other systems [11, 12, 6]. Fortunately the redirection features of the HTTP protocol make this implementation very straightforward. The outline of a web mirroring application is summarized below.

- Create a graph $G$ where each node (web server) has in-degree proportional to its free resources. In this case the free resources are of a server are defined to be the amount of bandwidth the next request can expect.

- When a new request arrives at node $v_i$, $v_i$ performs a short random-walk on $G$ and targets the node on the walk that has the highest in-degree (and most free bandwidth), $v_{(max)}$. Node $v_i$ then performs an HTTP redirection so that node $v_{(max)}$ will serve the request.

- As the load experienced by each server node fluctuates, it will compensate for those changes in load by adding or removing edges according to the edge dynamic in order to keep in-degree proportional to free resources.

The most obvious way to implement this system is to use the Internet directly as the underlying network for this web mirror application and we will proceed with this approach for the first release; however it will also possible to use the Brunet Hybrid P2P network that is currently under development [2] to form the overlay network. The most important advantage of this system for web mirroring is state-of-the-art firewall handling. These features will enable fire-walled nodes to serve web content and therefore participate as members of the web mirror networks; this could have the effect of drastically increasing the number of computers that are available to serve as mirrors. The other features such as support for heterogeneous underlying physical layer networks, and next-generation scalable search features [5] may also prove useful for scalable web server systems.

The first implementation of the web mirroring system will use the Internet directly. A subsequent version will provide support for using Brunet as the transport layer.
Figure 7. The variance of the in-degree distribution shows the process of reaching the equilibrium random graphs. This load balanced graph is initialized at $T = 0$ as an ordered regular graph ($\sigma^2 = 0$). During the initial rewiring the in-degree distribution spreads significantly so that by $T = 1600$ we see that $\sigma^2 \approx 25$. As the graph reaches equilibrium it become nearly regular again $\sigma^2 \approx 0.5$. This network has $N = 1024$, $< k > = 29$, and $k_{max} = 54$. The number of jobs that arrive and depart each time step is Poisson distributed.

4.2 Distributed Computing

The requirements of a distributed computing system are far more difficult that those of a web mirroring platform and so this implementation will be done after the web server application, however we can sketch the general approach to this system. The approach of openMosix and other distribute systems is often to distill all of the performance characteristics of a computer into a single scalar value which can be used to optimize the migration of processes across the network. We will likely take this mapping directly from openMosix or a similar project. To allow maximum deployment, implementation will allow for the migration of .Net CLI\textsuperscript{[8]} (Standard ECMA-335) applications that are compatible with the open source Mono\textsuperscript{[7]} implementation. Programs written in languages that target the CLI will be able to run on almost any operating system/hardware combination.

The outline of the steps is summarized below.

- Create a graph $G$ where each node (computer) has in-degree proportional to its free resources (for example the amount of CPU power available).
- When node $v_j$ runs a new program, $v_i$ performs a short random-walk on $G$ and sends the executable, command line arguments and input data to the node $v_j$, on the walk that has the largest in-degree. When the job finishes $v_j$ sends the program output back to $v_i$. Nodes compensate for changes in load by adding or removing edges in order to keep in-degree proportional to free resources.

Instead of using standard physical networks as systems like openMosix do, this system will likely use the Brunet Hybrid P2P network to form the overlay network. The advantages of this system include state-of-the-art firewall handling, support for heterogeneous underlying physical layer networks, and next-generation scalable search features that can enable distributed databases and other applications. Ini-
The performance of the load balancing algorithm increases as the walk length increases. This can be seen from the decreasing variance as the walk length is increased. However very short walks of $O(\log N)$ achieve nearly the same performance as much longer walks. This motivates the usage of short random walks for all of the protocol primitives.

Figure 5. The performance of the load balancing algorithm increases as the walk length increases. This can be seen from the decreasing variance as the walk length is increased. However very short walks of $O(\log N)$ achieve nearly the same performance as much longer walks. This motivates the usage of short random walks for all of the protocol primitives.

5 Discussion

We have proposed an algorithm that provides almost optimal load balancing performance by creating a nearly regular dynamic overlay network and mapping the in-degree of each node to its free resources. All network maintenance operations are based on the use of short random walks and rely only on local information obtained from the walks. We provide extensive simulations to show the efficacy and scalability of the protocol and present plans to use the protocol in applications.

All simulations in this paper deal with networks that have a maximum in-degree of $C = 54$ and a minimum in-degree of $k_{\text{min}} = 4$. This means that the resolution of the load state is represented by 50 possible states. As mentioned earlier, distributed mechanisms to alter the mapping of resources to in-degree can be added to account for changes in capacity and network size. The only constraint is that the number of possible load states cannot be greater than $N - 1$ (the total number of incoming edges a node can have.) Since existing polling methods work quite well for small networks, this is not really a limitation of the protocol; rather it underscores the fact that this system is designed to scale to tens of thousands of nodes and beyond.

The system can be implemented using light-weight transport protocols (UDP,RDP,etc.) to represent the large number of edges of the overlay network or the edges can represent cached routes in the underlying physical layer. Either approach should scale to huge networks.

For this algorithm we do not derive an expression for the steady-state degree distribution; we do show in extensive simulations that the protocol generates nearly regular random graphs. One argument to indicate that this algorithm scales is that this protocol is an optimization of the random walk algorithm in [15] that is shown analytically to produce Erdős-Rényi (ER) random graphs. While the protocol that generates ER graphs picks the last node of the random walk to receive migrated jobs, this algorithm chooses the highest degree node. This implies that nearly regular graphs generated here should have variances smaller than or equal to the ER graphs produced by the other protocol. Thus intuitively we expect that the optimized algorithm that chooses the highest degree (least loaded) should balance at least as well as the algorithm that generates ER graphs. Attempts to analyze this “ceiling random walk” which select the highest degree node is ongoing using statistical mechanical techniques[14].

In parallel with the initial implementation efforts described above, we will also embark on low-level simulations that mimic bursty load patterns and simulate the running of processes in a detailed fashion that mimics processes running on real machines.

In general load balancing is not restricted to the use of resources but also has a geographical component. There are a number of ways to incorporate locality into the random walk algorithm, including more general merit functions than pure in-degree, conditional sampling, and biased walks. An example of a general merit function would be to prefer geographically closer nodes on the walk even if they are not the highest degree; this effectively weights the sampled in-degrees by distance. An example of conditional sampling would be to select the highest degree node conditioned on other metrics such as geographical distance, available memory, etc. Lastly the random walk sampling itself could be biased in various ways to prefer the sampling certain kinds of nodes. Examining how these considerations affect the emergent overlay graphs may be a topic for future work.
References

[1] Apache Web Server. http://httpd.apache.org/.
[2] BruNet Hybrid P2P Network. http://cantor.ee.ucla.edu/ networks/brunet/brunet.html.
[3] Debian GNU/Linux. http://www.debian.org.
[4] Folding@home. http://folding.stanford.edu/.
[5] Linux Kernel. http://kernel.org.
[6] Linux Virtual Server. http://www.linuxvirtualserver.org/.
[7] Mono .Net. http://www.mono-project.com/about/index.html.
[8] .NET CLI. http://msdn.microsoft.com/net/ecma/.
[9] openMosaic. http://openmosix.sourceforge.net.
[10] SETI@home. http://setiathome.ssl.berkeley.edu/.
[11] Super Sparrow. http://www.supersparrow.org/.
[12] Ultra Monkey. http://www.ultrammonkey.org/.
[13] M. Andreolini, M. Colajanni, and R. Morselli. Performance study of dispatching algorithms in multi-tier web architectures. SIGMETRICS Perform. Eval. Rev., 30(2):10–20, 2002.
[14] J. Berg and M. Lssig. Correlated random networks. Phys. Rev. Lett., 89(22), 2002.
[15] J. S. A. Bridgewater, P. O. Boykin, and V. P. Roychowdhury. A statistical mechanical load balancer for the web. 2004.
[16] V. Cardellini, E. Casalicchio, M. Colajanni, and P. S. Yu. The state of the art in locally distributed web-server systems. ACM Comput. Surv., 34(2):263–311, 2002.
[17] A. D. David Kempe and J. Gehrke. Computing aggregate information using gossip. In Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science, 2003.
[18] R. Els and B. Monien. Load balancing of unit size tokens and expansion properties of graphs. In Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures, pages 266–273. ACM Press, 2003.
[19] U. Feige. A tight upper bound on the cover time for random walks on graphs. Random Struct. Algorithms, 6(1):51–54, 1995.
[20] N. Kahale. Eigenvalues and expansion of regular graphs. J. ACM, 42(5):1091–1106, 1995.
[21] R. Lling, B. Monien, and F. Ramme. A study of dynamic load balancing algorithms, 1991.
[22] D. S. Milojicic. Process migration. ACM Comput. Surv., 32(3):241–299, 2000.
[23] L. P. Peixoto. Load distribution: A survey, 1996.
[24] R. Subramanian and I. D. Scherson. An analysis of diffusive load-balancing. In Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures, pages 220–225. ACM Press, 1994.
[25] J. L. Wolf and P. S. Yu. On balancing the load in a clustered web farm. ACM Trans. Inter. Tech., 1(2):231–261, 2001.