Dynamics and Unpredictability of Load Balancing

Gregory Robinson
Department of Physics
University of California, Davis
garobinson@ucdavis.edu

June 12, 2010

Abstract

Load balancing, the dynamic redirection or reallocation of computing resources, has become prevalent in critical services in recent years. It is used in virtually all large real-time computing environments from logistics to military intelligence to stock exchanges. Algorithms used are often contrived and implemented without any consideration of what possible behaviors they can induce, nor what unpredictability they inherently exhibit. By simulating a very simple reallocation load balancing algorithm, this project examines some types of behavior that can occur. Findings indicate that the simple model used here is entirely predictable ($h_\mu = 0$) unless time delays between instruction and allocation are stochastic. With these stochastic delays, the system may become chaotic but retains short-term periodic structure. Additionally, we find that gradual relaxation to equilibrium is favorable over periodic and chaotic behavior as measured by both aggregate latency and reallocation overhead.

1 Introduction

1.1 Motivation

The advent of distributed computing brought with it significant challenges. Not least of these is the necessity to dynamically adjust which resources are used by various components of a distributed service. Engineers implementing load balancing on their clusters often neglect to carefully analyze the consequences of the algorithm and parameters they choose. Ignoring such analysis could lead to problems of varying degree. One such problem is inability to explain why a system is behaving in some undesirable way, thereby increasing the time required to alleviate that behavior. Even worse, load balancing algorithms can go haywire, disabling the system they intend to improve.

1.2 Summary of Project and Results

My report intends to illuminate the complicated nature of even simple load balancing algorithms. I examine what kinds of strange behaviors can occur, how to make sense of real-time load data, and the tradeoff between overhead and responsiveness for chaotic systems.

This exploration begins with a toy model of a discrete-time dynamic that instantaneously moves processing jobs from overloaded clusters to underloaded clusters. This system equilibrates regardless of initial conditions and the rate at which reallocation occurs. A slightly more realistic model reallocates jobs in a fixed number of timesteps. This small complication leads to periodic behavior for some initial conditions and parameters, and equilibrates for others. Yet another refinement of the toy model is stochastic reallocation timescale, which gives rise to chaos. Performance of the algorithm is approximated by two measures: additional time incurred as a result of overloaded clusters, and overhead incurred due to reallocating resources. Both measures favor gradual relaxation to equilibrium over both chaos and limit cycles.

1.3 Zooming Out

Studying the characteristics of load balancing is both interesting and applicable, but the implications are broader than just computation. I am eager to learn more about how non-instantaneous dynamics can complicate more realistic physical systems. Propagation time very well may be crucial to understanding, for example, the complicated behavior of relativistic N-body systems and neural networks [1, 2].

2 Background

2.1 Distributed Computing

Distributed computing is an umbrella term used to describe the method of splitting independent pieces of a computational application into pieces and *distributing* them across many computers. This distributed application could be protein folding, searching for extraterrestrial life, indexing the internet, or inferring transition probabilites for a very complicated epsilon machine. (The latter case will keep me busy for a few weeks this summer to further investigate this project.) Distributing an application increases its speed by increasing available resources such as CPUs, memory, hard disk, and network capacity.

The collection of computers performing the work is called a *grid*. Very large grids are often partitioned into *clusters*. Each cluster may contain hundreds to hundreds of thousands of individual hosts [3]. There may even be clusters of clusters, but this project need not be concerned with that.

2.2 Load Balancing

Distributed applications typically require careful design to maximize their utilization the vast resources at their disposal. One method of doing so is called load balancing. This denotes any scheme of rearranging the load put on each host or, more macroscopically, on each cluster.

One method of load balancing involves redirecting requests sent to an overloaded cluster toward an underloaded cluster. Another method, more typical of applications with significant interdependence between hosts, involves dynamically allocating more of a resource as it is necessary. (I have heard contention that this abuses the term, but it is a common misuse if so.) More interesting strategies can Rube Goldberg machines implemented with tens of thousands of lines of code and hundreds of both fixed and dynamic parameters.

Perhaps the simplest method is “round-robin” load balancing, which redirects client requests to alternating nodes in a cluster [4]. This paper will focus on the latter case, which is both more realistic for complicated services and more interesting.

3 System Under Investigation

In the spirit of computer engineers' casual approach to load balancing, I designed the model completely in the dark. I started with a description of what kind of load balancing is necessary and a list of assumptions I was willing to make.

3.1 Constraints, Assumptions, and Approximations

This project specializes on local resource balancing. Network resources should not be ignored, but it is still useful to see how each kind of load balancing works individually.

This project assumes that:

- The grid comprises large clusters with many hosts and many instances of a highly parallel service.
- Because clusters are "large", the load associated with a single instance (i.e., on a single host) of a service is small compared to total utilization.
- As a consequence of each instance being small compared to the cluster, utilization of each node takes approximately real values.
- The dynamic is conservative. It neither creates nor kills any jobs, and can only move them.
- The dynamic "ignores" clusters loaded within some optimal range. (Doesn't fix what is not broken.)
- The total load on the grid is constant.
- Resources available to the service are constant.
- Available resources are uniform between clusters.

3.2 The Toy Model

A good place to start is partitioning each cluster's load $l_i \in \mathcal{L}$ into the discrete alphabet imposed by the three regions: underloaded, optimal, and overloaded. For brevity, the letters A, B, and C will denote these regions, respectively. Two parameters mark off these regions: Υ, the threshold below which a cluster is considered underloaded, and Ω, the threshold above which a cluster is considered overloaded. These regions form a parition [5]:

$$\mathcal{P} = \{P_i\} = \{A, B, C\} \tag{1}$$

$$M = \cup P_i \tag{2}$$

$$P_i \cap P_i = \emptyset, i \neq j \tag{3}$$

The partition of each cluster's load is roughly what might be displayed on a service dashboard, so these are good candidates for the measurement symbols.

By constraint, optimally loaded clusters play no role in load balancing; all balancing takes place between overloaded and underloaded clusters. This is not really a good approximation for a sensible balancing method, but roughly emulates the equivalent "feature" often found in load balancing algorithms without undue complication.

I chose to relieve overloaded clusters proportionally to the difference between their load and the midpoint of the optimal region. In a similar way, the dynamic adds load to underloaded clusters proportionally to the difference between their load and the same midpoint. The rate at which laod is removed from overloaded clusters is parameterized by $\gamma \in (0, 1]$. The resulting equations for reallocation, after being careful about making the dynamic conservative, are

$$\omega_i = \gamma \left(l_i - \frac{1}{2} (\Upsilon + \Omega) \right) \mathcal{H} (l_i - \Omega) \tag{4}$$

$$\upsilon_i = \frac{\frac{1}{2} (\Upsilon + \Omega) - l_i}{\Omega} \mathcal{H} (\Upsilon - l_i) \sum_j \omega_j \tag{5}$$

where $\mathcal{H}$ is the unit step function, ω is the amount to remove from an overloaded cluster, and υ is the amount to add to an underloaded cluster.

Initially, I intended to study this system without any delay between the time at which the grid master decides to move instances and when those instances allocate on the target cluster. This proved uninteresting, as the system thermalizes rapidly for all parameter values. I therefore refined the system by adding fixed time delays, and still later stochastic time delays. It would be wise to investigate further refinements of the stochastic time-delay model in the future. For example, the probability distribution of time delays should have some nonuniform shape that is skewed according to present load and size of the new allocation.

To make time delays feasible, a locking mechanism is necessary. Any instruction to reallocate jobs locks the relevant clusters until the instruction takes place. In the simulation used here, locking is implemented as a mask array of zeros and ones that effectively removes all locked clusters from any consideration in the balancing dynamic.

4 Methods

The entreé of this research is a simple Python application that implements the simplified algorithm above.

It is not easy to judge how favorable a load balancing algorithm is working without some meaningful measure of its performance. Two important measures are additional latency incurred due to clusters being overloaded and the cost of load balancing. Each of these measure very different aspects of the system, so there is no way to compare them in general. For this reason, I examined results of each separately.

Additional latency is a result of scheduling overhead and similar superlinear functions of load. This can be very crudely approximated as the square of the difference between an overloaded cluster's load and the optimal load of that cluster. The simple justification for this is that if one process takes 1 cpu second to complete, and there are two instances running on the same processor, the average time to complete would be two seconds. In general, the average time to complete each of N identical tasks that require T cpu seconds on a single processor is given by $<\tilde{T}>= TN$. Since there are N processes, the total latency these tasks incur on a single processor compared to each running on a dedicated processor is $TN^2 - TN$. The approximate overtime is then given by

$$\text{overtime per iteration} = \sum_{i\in\text{clusters}} \left(l_i - \frac{1}{2}\left(\Upsilon + \Omega\right)\right)^2 \mathcal{H}\left(l_i - \Omega\right) \qquad (6)$$

Overhead is much easier to measure. Here we will simply assume that overhead is a linear function of the sum of the load moved at each step. This allows us to write:

$$\begin{aligned}\text{cost per iteration} &= \sum_{i\in\text{clusters}} \omega_i && (7)\\ &= \sum_{i\in\text{clusters}} \gamma\left(l_i - \frac{1}{2}(\Upsilon+\Omega)\right)\mathcal{H}(l_i-\Omega) && (8)\end{aligned}$$

To sample the performance for various parameters, a Monte Carlo sampler randomly generates initial states with each cluster loaded up to five times optimal capacity for various values of γ. This sampler tries 100 equidistant values of $\gamma \in (0,1]$ and picks 10,000 random initial states for each. Each simulation runs for 10,000 iterations, after which the performance measures are averaged over all initial states and iterations thereof.

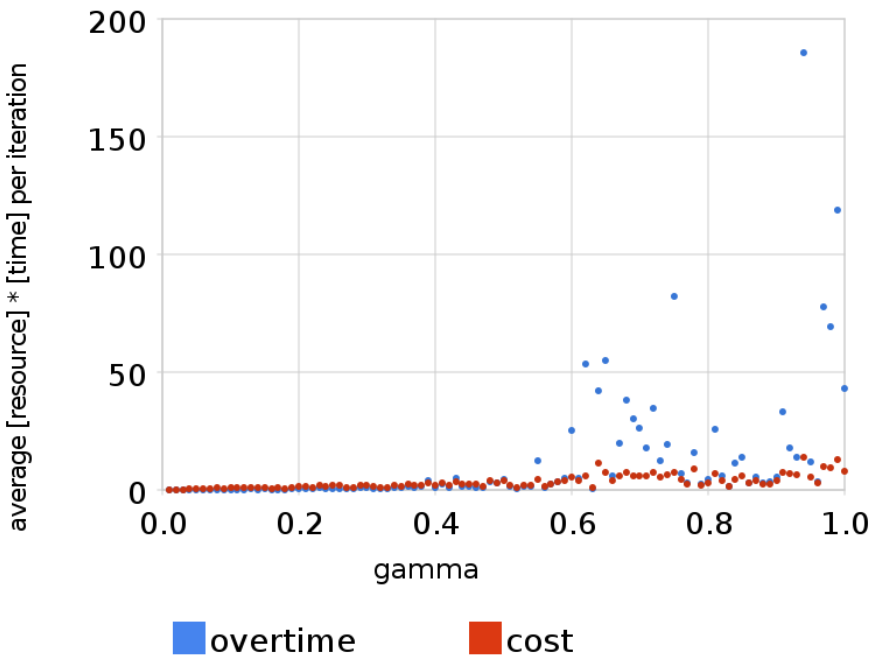
I attempted to use both state splitting and tree merging inference to this system in an attempt to find the mutual information between past and future measurement symbols. Regrettably, this has not been successful. It is computationally difficult to do this by brute force, and I have not yet been able to implement a more cunning method. In the coming months I will attempt to write a search algorithm that will sample the space of L-cylinder induced by the partition. That is, the algorithm will sample the underlying state space for each possible initial measurement symbol (all 3^N of them!) to estimate the probability distribution of future words.

Part of the reason it is so tricky to understand this system analytically is the inherent cross-dependence on each component of the load vector. There might be a way to write the equations in a more elegant manner, but that is also proving to be difficult.

5 Results

- Instantaneous dynamic equilibriates system quickly.
- Fixed-interval delay leads to either equilibrium or periodic orbits.
- Stochastic delays sometimes results in chaos.
- Slow migration rate (i.e. small λ) performs better than periodic and chaotic regions.

Figure 1: Performance Measures vs. Migration Rate



- When $N \geq 4$, uncoupled periodic orbits may exist due to staggered locking.

The most useful result I have found is that there is zero unpredictability in this system if time delays are fixed. The reason for its utility is mosty that sources of unpredictability of load balanced clusters is mostly unknown. This result narrows down the possibilities, and might be useful in designing more predictable algorithms.

Another interesting result is the trend of performance versus migration rate. As exemplified in Figure 1, it is counterintuitively harmful for the load balancer to be overzealous in moving jobs to underloaded clusters. This plot shows the average of 10,000 iterations of 10,000 initial data points for each of 100 values of γ with a grid of 10 clusters. It shows that the algorithm is fairly well-behaved until performance starts to severely degrade for γ larger than approximately 0.5. No evidence has been found in support of instability being advantageous in any way, supporting hypothesis.

The way this system becomes chaotic is rather curious. Unpredictability is a result of hopping between multiple periodic orbits of different length. In this way, there is very much a local structure in the symbol sequences — and even the hidden state inside the black box — but the long-term behavior is largely unknown.

6 Conclusion

Even a simple load balancing scheme can impose uncertainty and undesirable behavior onto a distributed application. The most significant takeaway is that even algorithms that are stable if their dynamic takes effect instantaneously may behave erratically in practice. This should serve as warning that these algorithms should be thoroughly investigated before deploying them. We have seen that there is uncertainty in future states given the present state. The chaotic behavior causing this uncertainty desrves more attention: it arises from the interleaving of periodic orbits.

This project has given me far more questions than answers. Despite constantly hearing the mantra that research is never complete, this feels especially incomplete. I still do not know much about this system, and I haven't even been able to calculate mutual information between past and future, which was one of my primary goals. Future research will include a selective sampling (as described in 4) to find this mutual information, better description of what conditions lead to chaos and why.

References

[1] Moser, Jurgen. Stable and random motions in dynamical systems. Princeton: Princeton University Press, 2001.

[2] Abbot, L.F. Decoding Neuronal Firing And Modeling Neural Networks. Quart. Rev. Biophys. 27:291-331. 1994.

[3] Top 500 Supercomputer Sites. Available from http://www.top500.org/. Internet; accessed 10 June 2010.

[4] IBM. "Understanding IBM HTTP Server plug-in Load Balancing in a clustered environment." Available from http://www-

01.ibm.com/support/docview.wss?rs=180&uid=swg21219567. Internet; accessed 09 June 2010.

[5] Crutchfield, James. Natural Computation and Self Organization Lecture 15. 4 March 2010.