OnCall: Defeating Spikes with a Dynamic Application Cluster
Keith Coleman and James Norris
October 23, 2003

1 Overview

Today’s internet application clusters are severely limited in their responses to major traffic spikes and “flash crowds,” as they run a single application on a fixed set of machines, with each machine requiring up to several hours of configuration before becoming operational. If they were to provide enough capacity to serve all spikes, clusters would become exceedingly expensive and would waste enormous resources while idling during more typical traffic workloads.

As a more efficient alternative, we propose a dynamic application cluster design based on VMs. The system is dubbed “OnCall” for the extra computing capacity that is always on-call in case of traffic spiking. Our development efforts focus on an OnCall resource manager that effectively defeats traffic spikes by re-allocating computing capacity to spiking applications, while still maintaining inter-application performability isolation.

2 Introduction

2.1 Why OnCall is the Best

Dynamic clusters handle spikes more effectively than static clusters.

Our system is better than previous attempts—both its resource allocation policy and mechanism are superior.

Resource allocation mechanism is better because:
- Accounts for numerous applications competing for scarce resources on a shared cluster (others don’t use consider clusters shared among competing apps)
- Allows application owners to specify a customized allocation policy to meet their specific price-performance needs.
- Provides functionality allowing applications to claim resources in a way that provides useful guarantees (equivalent to what one would get on a static cluster) and also provides potential to handle a spike \( \rightarrow \) more utility for lower cost than a static cluster (others don’t handle spikes in the case of shared clusters or limited resources)
- Provides functionality allowing applications to achieve stable allocation that avoids thrashing (others don’t address this)

Mechanism is superior because:
- VMs provide performability and security isolation (others aren’t secure – one application could bring down the data center)
- VMs are efficient enough and are becoming more efficient (Intel to support virtualization and Microsoft trusted computing will run on VMMs)
- With VMs can reduce server start-up time through pre-booting and pre-caching of VMs on physical boxes (others rely on copying OS images and then booting)
- VMs provide compatibility with existing applications (others do not – require writing application to a specific platform, API or structure)
OnCall can be application generic. Apps can provide custom metric monitors and policy engines, but doing so does not require rewriting applications for a specific framework. [others are not application generic]

OnCall is fault-tolerant [others don’t address this]

**2.2 Spike Response is All You Need**

Some systems and allocation policies attempt to achieve optimal allocation under the average case where there are no spikes. We concentrate on handling the exceptional case (e.g. spikes).

Handling the exceptional case is more difficult and more important than handling the average case (static clusters can already handle the average case).

By handling the exceptional case we also handle the average case. If we can meet demands under constrained cases, we will also be able to meet them under unconstrained cases.

**3 Design**

**3.1 General Overview of System**

Two key components of OnCall:

1. The Marketplace is the mechanism through which resource allocation policies operate.
2. The Platform is the actual system structure used to manage and operate the cluster

**3.2 Marketplace**

OnCall does not provide a specific resource allocation policy, but rather provides a powerful framework, the OnCall Marketplace, that allows individual applications to define their own policies. Sample policies are used to demonstrate the possibilities enabled by OnCall.

The OnCall marketplace determines the resource allocation between various competing applications on the cluster. The following discussion outlines key features of the marketplace:

*Guarantees*

OnCall provides a single, powerful guarantee that a given application will have a certain number of computers, \( G \), guaranteed to be available for its use, should it need them. Application owners pay a flat rate, proportional to the size of \( G \), for this guarantee. This represents the offline portion of the marketplace; they can sell any excess resource on these \( G \) computers in the online marketplace (see below) in order to recoup costs or even turn profits.

*Online Resource Marketplace*

Since at certain points applications may not need the full capacity provided by their \( G \) computers, they can sell this capacity to other applications though the online marketplace. Each application can provide its own policy, which is effectively a function with 2 inputs and one output:
MarketplacePolicy(\textit{PerformanceStats, Price}) returns \textit{NumComputersToBuySell}

Basically, given its current performance statistics, the application decides how many computers it would like to buy or sell at a certain price.

Application owners can determine what their policy should be by analyzing a basic feedback control loop:

There are two inputs: (1) the rate of requests, which an application owner cannot control, (2) the number of computers serving those requests, which an application owner can control. The outputs are the response times for the application’s requests, which can be aggregated using the mean, standard deviation, 90\textsuperscript{th} percentile, etc.

[Armando & George – is this the right way to specify a feedback control loop problem]

Applications have a notion of dollar-value profit yield as a function of response time and a notion of cost as a function of the number of computers in use, and simply want to maximize yield minus cost.

An example of a simple but effective policy might be the following (NOTE: this will not appear in this section of the paper, but rather will be described in the Results section as the example policy we use in our tests)

- You know the number of computers you expect to need in order to handle your current traffic. You know the value of one additional user being served. You use these to calculate the price each computer is worth to you.
- You determine the number of computers you want, \(n\). If \(n\) is less than \(G\), you sell the excess computers (\(\text{excess} = G - n\)) for any price; if \(n\) is greater than \(G\), you use all \(G\) computers and you’ll be willing to buy up to \(n - G\) extra computers at some fixed price, \(P\), which is you have determined in advance to be the value of one additional computer when your application is overloaded.
- To determine the number of computers you expect to need, you take max of historic running averages and predicted future needs and use that as the expected number of computers.
- Predictions of future needs use the number of computers the application is actually using as a damper on the number of computers the application expects to need. Accomplish this by placing value on unused resources through an idle resource tax. This feature provides hysteresis to avoid thrashing.

\textit{Host Profit Generation}

The hosting provider who owns the cluster can potentially generate extra profits through two methods:

1. The host can shut down unused computers to save on utilities (e.g. air conditioning, electricity) and maintenance cost.
2. The host can provide additional capacity on the cluster (above the sum of the \(G\)-\textit{maxes}) and sell that capacity when profitable. This assumes that spikes are frequent enough for this to be profitable.

This can be done by creating a special host-owned “Shut Down” application on the cluster that is willing to buy capacity at anything less than the operational expense. The application then shuts
down any computers it is able to buy. It then restarts these computers if it can sell them at a price greater than the operational expense.

**Competitive Marketplace that’s Resistant to Tampering**

Marketplace is resistant to tampering by any one application because:

1. It can be assumed that there are enough applications so that one app’s policy could not alter the equilibrium market price (this is a safe assumption because there will only be enough capacity on the cluster to make OnCall useful if there are many different applications). In cases where there are few applications, they can assumed to be cooperative (e.g. Yahoo runs all its services on one shared cluster). These are safe assumptions because OnCall is ineffective in other scenarios.
2. When the OnCall marketplace is determining the market equilibrium price, it can select the lowest of the range of possible values, ensuring that a seller does not inflate the equilibrium price.

### 3.3 Platform

#### 3.3.1 VMs (and general infrastructure)

Each node in an OnCall cluster runs a virtual machine monitor; OnCall applications are bundled as VM capsules. This provides 4 benefits: application generality, performability isolation, security isolation, fast-boot from suspended VM. (Status: operational.)

#### 3.3.2 Pre-caching VMs

Can pre-cache suspended VMs on certain computers based on predicted needs. (Status: this has not been as thoroughly explored as the remainder of the features and may not make it in to OnCall v1. It is quite complex. We would probably create a separate market for pre-caching, but determining prices on that market is non-trivial.)

#### 3.3.3 Security

**Virtual LANs**

Each OnCall application operates on its own Virtual LAN, ensuring that applications can’t read each other’s network traffic. (Status: operational on the Collective, not yet with OnCall.)

**Virtual Machines**

VMs allow each application to control its own environment and run in complete isolation. (Status: operational.)

#### 3.3.4 Fault Tolerance

**Master and Backup Scheduler**

Two (or more) copies of OnCall’s central scheduler are running at any time—one copy is the master and the other is a backup. Use standard master election. (Status: not yet used.)

#### 3.3.5 Multi-tiered applications

In an offline application configuration file, administrators can define constraints on a given application—e.g. what components/tiers can be replicated, how many front-end servers a single database can serve, etc—which can then be used online to constrain replication. (Status: somewhat operational. Spec language and format exists, though not all features are used. May
not be part of OnCall v1. Important to account for but not necessarily a top priority feature for proof-of-concept.)

4 Results

[Want to show that that we are better than existing systems (do so by handling cases they do not handle (e.g. spikes) and that we are close to the optimal, theoretical solution.]

4.1 Optimal vs Actual
Graph the optimal number of computers assigned to a given application vs the actual number of computers assigned in a given case. Here we define “optimal” as: (a) using the minimum resources needed to satisfy performance desires, and (b) providing an economically-efficient allocation of excess capacity.

This demonstrates two things: (1) that it is possible for application owners to define policies that allow them to effectively control the performance of their applications (in order to meet whatever performance desires they have), and (2) that given the outputs of all applications’ policy engines, OnCall correctly allocates machines among them.

Clearly, to test this we will have to fabricate a set of applications, each with specific performance goals. We will probably have these goals be something simple like “use the minimum number of resources necessary to achieve some desired response time”—then we can employ the simple example policy described earlier in the paper. This also provides an easy way to measure how close our policy gets us to the “optimal” allocation.

The test scenario will be under “normal” usage shifts, not spike conditions that would unnecessarily complicate the graph.

We will demonstrate optimality by showing the small area between the curve of actual response times and a desired response-time threshold—can show that in OnCall your response times never go over that threshold.

4.2 Handling a Spike
Create a spike that the cluster can accommodate and demonstrate that we can meet its needs and provide reasonable response times. Multiple graphs: # of requests, response times, number of allocated computers, aggregate cpu usage, predicted cpu usage, predicted number of servers.

Again, will fabricate of set of applications and policies.

4.3 Cluster Overload
Create a spike that we can’t handle and show performability isolation between applications.

4.4 Resistance to Thrashing
Demonstrate that given a specific policy and workload OnCall provides allows an application to avoid thrashing.
Thrashing, here, is time that servers spend starting up and shutting down (e.g. not handling requests).

How to demonstrate resistance to thrashing? Create a workload. Determine offline what the optimal allocation of resources would be assuming applications on the cluster were using identical, pre-defined marketplace policies. Then measure how the actual allocation of resources compares to the version calculated offline—measure difference between curves.

5 Related Work

5.1 Policy-related work

QoS-Driven Server Migration for Internet Data Centers (Ranjan et al – Rice, HP Labs 2002)
A simple algorithm for computing the number of servers to allocate given a certain usage on the current active machines. Computes number of servers to attain target CPU usage. OnCall is more sophisticated.

Integrated Resource Management for Cluster-based Internet Services (Shen et al. OSDI 2002.)
Every application runs on every node. Tries to maximize yield for different service levels, not for different services—this is a fundamentally different problem.

Adaptive Overload Control for Busy Internet Servers (Welsh and Culler. OSDI 2002)
Depends on SEDA architecture – admits connections based on bottlenecks within the SEDA stages. Controls performance and avoids overload within an application.

Model-Based Resource Provisioning in aWeb Service Utility (Doyle and Chase. USITS 2002)
Focused on serving static data. Models are for static content and don’t directly apply to dynamic content.

Dynamic Surge Protection: An Approach to Handling Unexpected Workload Surges With Resource Actions That Have Lead Times (IBM)
Uses regression analysis—we may too. Not based on shared environment—e.g. determines how many computers to allocate for a given application but does not consider competing applications where it would need to decide which, among several applications, will receive the available capacity.

Utility-Directed Allocation (Kelly)
Says maximization of utility is an integer linear programming problem, but doesn’t look at competing interests (e.g. different applications competing for the same resource) and doesn’t formulate a solution.

Clockwork (IBM)
Based on long-term cyclical predictions, which isn’t designed to handle spikes.

Resource Overbooking and Application Profiling in Shared Hosting Platforms (Roscoe et al)
Designed to provide probabilistic guarantees that fail during spikes.

5.2 Mechanism-related work
vMatrix (Amr Awadallah)
Outlines possible use of VMs but doesn’t address spikes or realistic allocation policies

Collective
Mechanism for managing server updates and replication.

Denali: A Scalable Isolation Kernel (Whitaker, Shaw, Gribble – UW 2002)
Uses lightweight VMs that don’t allow backwards compatibility with existing applications.

Doesn’t use VMs (so must copy OS images) and depends on enough resources being available, so it won’t really handle big spikes.

Various P2P Systems (Stading; Stavrou; Padmanabhan)
All designed for static content. P2P probably not what you’re looking for with large-scale professional hosting systems. More efficient to have things in single large-scale clusters that can be managed easily.

6 Conclusion
We have shown that…[repeat goals from introduction].