Handling Flash Crowds from your Garage

Jeremy Elson and Jon Howell
Microsoft Research
{jelson,howell}@microsoft.com

Abstract

The garage innovator creates new web applications which may rocket to popular success – or sink when the flash crowd that arrives melts the web server. In the web context, *utility computing* provides a path by which the innovator can, with minimal capital, prepare for overwhelming popularity. Many components required for web computing have recently become available as utilities.

We analyze the design space of building a load-balanced system in the context of garage innovation. We present six experiments that inform this analysis by highlighting limitations of each approach. We report our experience with three services we deployed in "garage" style, and with the flash crowds that each drew.

1 Introduction

For years, developers and researchers have refined ways of scaling Internet services to larger and larger capacities. Many well-known techniques are now available [33, 17, 4, 19, 6, 36, 15, 13].

But capacity is expensive: racks full of web servers, database replicas, and load balancing switches require a significant up-front investment. For popular sites, where load is consistently high, it is easy to justify this investment; the resources will not be idle. Less popular sites can not justify the expense of large idle capacity. But how can an unpopular site make the transition to popular—when "flash crowds" often make that transition almost instantaneous and without warning?

In this paper, we consider the question of scaling through the eyes of a character we call the *garage innovator*. The garage innovator is creative, technically savvy, and ambitious. She has a great idea for the Next Big Thing on the web and implements it using some spare servers sitting out in the garage. The service is up and running, draws new visitors from time to time, and makes some meager income from advertising and subscriptions. Someday, perhaps, her site will hit the jackpot. Maybe it will reach the front page of Slashdot or Digg; maybe Valleywag or the New York Times will mention it.

Our innovator may get only one shot at widespread publicity. If and when that happens, tens of thousands of people will visit her site. Since her idea is so novel, many will become revenue-generating customers and refer friends. But a flash crowd is notoriously fickle; the outcome won't be nearly as idyllic if the site crashes under its load. Many people won't bother to return if the site doesn't work the first time. Still, it is hard to justify paying tens of thousands of dollars for resources *just in case* the site experiences a sudden load spike. Flash crowds are both the garage innovator's bane and her goal.

One way out of this conundrum has been enabled by contemporary *utility computing*. More and more of the basic building blocks of scalability—network bandwidth, large-scale storage, and compute servers—are now available in forms analogous to traditional utilities like electricity and water. That is, a contract with a utility has very little overhead, gives you access to vast resources almost instantly, and only bills you for the resources you use.

Over the past year, our research group created three web sites that experienced sudden surges in popularity, including one that was (literally) Slashdotted. Each was implemented using publicly available computing utilities and was able to withstand its flash crowd at low cost.

1.1 Contributions

In this paper, we contribute a detailed analysis of the issues and tradeoffs a typical garage innovator will encounter when building low-cost, scalable Internet services. We explain these tradeoffs in Section 3. Our analysis draws heavily from both a series of controlled micro-benchmark experiments, which we describe in Section 4, and wisdom gleaned from our own deployment of three "garage-scalable" services, all of which were subject to real flash crowds. These case studies, found in Section 5, report our design and implementation decisions, how each service responded to a flash crowd, and lessons learned from each experience.

Before diving into our analysis, however, we first lay groundwork in Section 2, which describes the current state of utility computing and briefly reviews the most common strategies for building scalable Internet services.

2 Contemporary Utility Computing

The past two years have seen a surge of tools that are wonderfully useful for garage innovators. We describe several of them in this section. First, we offer a list of what we think are the properties essential for garage use:

Low overhead during the lean times. Cost should be proportional to use, not to capacity. During long periods of unpopularity, a garage budget can't pay for the huge capacity that might someday be needed.

Highly scalable. The garage-based service may only need one server today, but when the flash crowd comes, it might need 20 or 200. Worst-case flash crowd resources have to be available: if a service is underprovisioned, there is no point in using it.

Quickly scalable. It's not enough that large-scale resources are *available*; they have to be available *quickly*. There's no time to call customer service, ask for an upgraded account, and start configuring machines. Flash crowds are notoriously fickle. If a service can't scale in near-immediate response to a surge of interest, there is no point in using it.

Services that meet these criteria are often referred to as *utility computing*, a term coined by John McCarthy in 1961. Utility computing services rely on *statistical multiplexing*: providing service to a large number of customers whose load spikes are likely to be de-correlated.

An illustrative shift towards utility computing can be found in the way large colocation centers sell bandwidth to customers. It is common today to see it billed as a utility: a customer gets a fast (say, 100Mbps) connection to her servers. The entire 100Mbps is usable, and actual usage is metered. Using very little bandwidth costs very little; a sudden usage surge is billed accordingly. This contrasts with circuits installed to an individual customer's site, virtually all of which are billed according to peak capacity regardless of actual usage.

2.1 Building Blocks

In this section, we lay the foundation for the rest of the paper, describing some of the utility computing services that have arisen in the past few years. Then, in Section 2.2, we describe a few well-known scaling architectures and describe how a garage innovator can implement them using the utility building blocks that are available today.

2.1.1 Storage Delivery Networks

One great boon to the garage innovator has been the rise of Storage Delivery Networks (SDNs), such as Amazon's S3 [26] and the Nirvanix platform [25]. SDNs have interfaces that resemble a simple managed web server. Developers can upload static files such as web pages and images that the SDN will store and serve to clients using standard HTTP.

Unlike traditional managed web hosting, often implemented using a single machine, SDNs are large clusters of tightly coupled machines. The implementation details of data replication, distributed consensus, and load distribution are all hidden behind the simple static-content interface. A single customer's flash crowd can potentially use the full power of the entire cluster.

This strategy should sound familiar: SDNs are similar to Content Distribution Networks (CDNs) such as Akamai [1] andLimelight Networks [21]. CDNs and SDNs have a number of technical differences; for example, SDNs are typically located in a single datacenter, while CDNs minimize latency using thousands of Internet points of presence. CDNs are far less attractive to garage innovators than SDNs, however—not for technical reasons, but economic ones. The cost of entry into a CDNs is typically high, as well as frustratingly opaque [29] ("contact a sales representative for more information!"). Large start-up costs and minimum bandwidth commitments place most CDNs out of the reach of garage innovators who don't yet have the budget associated with a wide audience. S3 and Nirvanix have no start-up or recurring costs; they are strictly fee-for-service utilities. A customer who serves one gigabyte of data in a month will literally be billed 20 cents for the month. There may be no fundamental technical or economic reason why CDNs cannot adopt a similar billing model; perhaps some day they will.

SDNs thus fill a useful niche today for the garage innovator. They are more quickly scalable than typical managed hosting servers; they do not carry the significant expense of geo-distribution that is inherent to CDNs; and their pricing models allow flash crowd-capable content distribution without any significant investment until the flash crowd arrives.

2.1.2 Commodity Virtualization

We mentioned in the introduction of Section 2 that colocation centers now charge for bandwidth in a utility-computing, garage-friendly way. However, up until recently, the only way to exploit utility bandwidth was to buy or rent a server and pay a monthly fee (typically a few hundred dollars) to host it. For the garage innovator on a truly shoe-string budget, this can be prohibitive. Dozens of hosting companies now offer virtual machines for rent, typically starting at around 20 dollars a month. There are dozens of examples (search the Internet for "virtual private servers"). Setup usually takes about a day. Developers can use these virtual machines to exploit the pay-per-use bandwidth of a colo facility without the overhead of using an entire physical machine.

Garage innovators can also exploit the fact that virtual servers are so widely available around the globe—offered by hosting providers in dozens of locations around the United States, Europe, Asia, and the Pacific Rim. By renting several of them, a developer on a limited budget can achieve a level of geographic diversity that was formerly only possible for large-scale Internet services with large-scale budgets.

2.1.3 Compute Clouds

While the widespread availability of virtual servers has been a boon, it did have an important limitation. The flexibility of a virtual server was typically only in burstable bandwidth. If an application is CPU or disk intensive, a flash crowd doesn't just need more bandwidth, it needs more servers. Consequently, in the past year, companies have begun to follow the utility computing model for entire virtual machines, not just the bandwidth they consume.

Amazon's EC2 "elastic compute cloud" [2] and Flex-iScale [35] stand out in this area. They allow developers to configure and save a virtual machine image, then create multiple running instances of that image. Images can be instantiated in about one minute. Usage is billed by the hour, rather than by the month. Virtual machines can be started and stopped using a simple programmatic API. This makes it possible for a garage innovator to create an image of a machine running her Internet service, monitor the load, and almost instantly increase the number of running instances if needed. As we will see in Section 2.2, there are several ways of scalably distributing the load of an Internet service over multiple servers, each with different advantages.

2.1.4 DNS Outsourcing

Another useful computing utility is outsourced Domain Name System (DNS) hosting. DNS traffic usually accounts for a small part of a site's resource budget, but outsourcing DNS is useful because it prevents DNS from becoming a single point of failure for garage-based services. (We will explore this further in Section 3.5.)

Typical services in this space are the highly redundant and low-cost UltraDNS [24] and Granite Canyon's free Public DNS Service [14]. Their DNS servers replicate and serve the DNS for customer domains. They automatically copy the authoritative DNS configuration every time they receive a change notification.

2.1.5 A missing piece: relational databases

Dynamic web services are often implemented as a collection of stateless front-end servers that paint a user interface over data stored in a relational database back end. Relational databases' powerful transactional model provides idiot-proof concurrency management, and their automatic indexing and query planning relieve programmers from the burden of designing efficient persistent data structures [8]. However, implementing highly-scalable databases that retain the full generality of the relational model has proven elusive. Scalable databases typically abandon full transactionality, arbitrary queries, or both. Utility access to a scalable database is therefore even further in the future.

There exist lightweight scalable utility databases, such as S3 and Amazon's SimpleDB [3]. Later in the paper (Sections 5.2 and 5.3), we describe experiences substituting the conventional relational database, sometimes with a lightweight database, and other times with alternate workarounds. Every approach incurs a higher development cost over using a powerful relational database, a cost of scalability we do not know how to eliminate today.

2.2 Scaling Architectures

Before proceeding to our analysis (Section 3), we will briefly review some of the common scaling architectures used today for Internet services. This discussion focuses purely on the Internet-facing part of the system: that is, methods for rendezvous of a large number of Internet clients to the large number of Internet-facing servers that handle their sessions. We do not consider scalability of back-end elements such as databases. (Our case studies in Section 5 will revisit the scalable back-end issue.)

This section will describe each design, and briefly touch on its main advantages and disadvantages.

2.2.1 Using the bare SDN

Most of the design alternatives we will consider assume that an innovator's web site has dynamic content, and therefore requires a compute cluster that can run garage code. However, some web sites rely heavily (or even exclusively) on static content. For example, video sharing sites typically show dynamic web pages that display the latest comments and ratings, but the contained video is a much larger static object.

In these cases, simply storing the static parts of a web site on an SDN is near ideal for the garage innovator. Using the SDN costs our innovator virtually nothing upfront beyond the small fee for storage. Because of the statistical multiplexing we described in Section 2.1.1, the SDN is likely to be highly available even during the arrival of a flash crowd.

The main disadvantage of using an SDN, of course, is that it serves only static content.

2.2.2 DNS load-balanced cluster

We now turn our attention to clusters running custom code designed by our garage innovator. One simple approach is to use the DNS protocol to balance load: A collection of servers implement the custom web service, and a DNS server maps a single name to the complete list of IP addresses of the servers [5]. Standard DNS servers will permute this list in round-robin fashion; if clients try the addresses in order, then various clients will be randomly spread across the servers. Clients should also fail over to a second address if the first one does not reply, affording fault tolerance. (In sections 4.3 and 4.4, we will show how these properties can fail.)

When a flash crowd arrives, new servers are brought online, and the DNS record is updated to include their addresses. Clients of ISPs that have cached the previous record won't see the new servers until the old record expires. Fortunately, the nature of a flash crowd means that most of the traffic is new. On the other hand, record expiration does reduce the responsiveness of DNS load balancing to server failure.

A startup called RightScale offers a DNS load-balancing management layer in front of EC2 [31].

2.2.3 HTTP Redirection

Another way to rendezvous clients with servers is to use a front-end server whose only function is HTTP redirection [11]. Microsoft's "Live Mail" (formerly Hotmail) exemplifies this strategy. Users access `mail.live.com`. If they have a login cookie, they are given an HTTP redirect to a specific host in the Live Mail farm, such as `by120w.bay120.mail.live.com`. (Users who are not logged in are redirected to a login page.) All interactions beyond the first redirection happen directly with that machine. The HTTP redirector, of course, can base its redirection decisions on instantaneous load and availability information about the servers in its farm.

This solution is attractive for two reasons. First, it introduces very little overhead: the redirector is not involved in the session other than providing the initial redirect. Second, redirection doesn't take much time; a single redirection server can supply redirections to a large number of clients very quickly. (URL forwarding services such as `tinyurl.com` and `snipurl.com` demonstrate of this: individual redirections take very little time, so they can easily provide redirection service at Internet scales.)

2.2.4 L4 or L7 Load Balancing

In both L4 and L7 load balancing, a machine answers all requests on a single IP address, and spreads the request streams out to back-end servers to balance load. The client appears to communicate with a single logical host, so there is no dependency on client behavior. Faults can be quickly mitigated because the load-balancing machine can route new requests away from a failed server as soon as the failure is known.

L4 (layer 4) load balancing is also known as "reverse network address translation (NAT)". An L4 balancer inspects only the source IP address and TCP port of each incoming packet, forwarding each TCP stream to one of the back-end servers. L4 balancing can run at "router speeds" since the computational requirements are so modest.

L7 (layer 7) load balancing is also known as "reverse proxying." An HTTP L7 load balancer acts as a TCP endpoint, collects an entire HTTP request, parses and examines the headers, and then forwards the entire request to one of the back-end servers. L7 balancing requires deeper processing than L4, but provides the balancer the opportunity to make mapping decisions based on HTTP-level variables (such as a session cookie), or even application-specific variables.

One important disadvantage of load balancers is that high-performance load balancing switches can be very expensive (tens to hundreds of thousands of dollars), difficult to fit into a garage budget. However, there are lower-cost options. First, there is free software, such as Linux Virtual Server [22], and commodity software, such as Microsoft Internet Security and Acceleration Server [32], that implement L4 and L7 load balancing, though they are less performant than dedicated hardware. The second option is a service introduced in October of 2007 by FlexiScale [35]. They combine on-demand virtual machines with fractional (utility) access to a high-performance L4/L7 load balancing switch. To our knowledge, this is the only current offering of a load balancing switch billed as a utility.

2.2.5 Hybrid Approaches

The techniques described above can be combined to offset their various limitations.

One example above (Section 2.2.1) splits a service, such as a video sharing site, into a low-bandwidth active component managed by a load-balanced cluster, and a high-bandwidth static component served out of the SDN.

Alternatively, consider a DNS cluster of L4/L7 load balancers: Each L4/L7 cluster is fault-tolerant mitigating DNS' sluggishness to recover from back-end faults; and the entire configuration can scale beyond the limits of a single L4/L7 cluster.

3 Analysis of the Design Space

In this section, we analyze the tradeoffs a garage innovator is likely to encounter when building a scalable service, using one of the design templates we reviewed in Section 2.2, and implemented on top of the building blocks we reviewed in Section 2.1. Our analysis is drawn from both a series of micro-benchmark experiments, fleshed out in Section 4, and lessons learned from our own implementations of garage-style services that were subjected to real flash crowds, described in Section 5.

The important design criteria are:

Criterion	Design			
	Bare SDN	HTTP Redir.	L4/L7 Load Bal.	DNS Load Bal.
§3.1: Application Scope	Static HTTP	HTTP	All	All
§3.2: Scale Limitation	Very large	Client arrival rate	Total traffic rate	Unlimited
§3.3: Client affinity	N/A	Consistent	Consistent	Inconsistent
§3.4: Scale-Up Time	Immediate	VM Startup Time (about a minute)	VM Startup Time (about a minute)	VM Startup + DNS TTL (5-10 minutes)
§3.4: Scale-Down Time	Immediate	Session Length	Session Length	Days
§3.5: Front-End Node Failure: Effect on New Sessions	N/A	Total Failure	Total Failure	Major Failure
§3.5: Front-End Node Failure: Effect on Estab. Sessions	N/A	No effect	Total Failure	Rare effect
§3.5: Front-End Node Failure: Effect on New Sessions (m redundant front-ends)	Unlikely	long delay for $1/m$th sessions?	long delay for $1/m$th sessions?	Short delay (§4.2)
§3.5: Front-End Node Failure: Effect on Estab. Sessions (m redundant front-ends)	Unlikely	No effect	$1/m$th sessions fail	A few sessions see short delay
§3.6: Back-End Node Failure: Effect on New Sessions	Unlikely	No effect	No effect	long delay for $1/n$th of sessions
§3.6: Back-End Node Failure: Effect on Estab. Sessions	Unlikely	User-recoverable failure	Transient failure	long delay for $1/n$th of sessions

Table 1: A summary of the tradeoffs involved in different scaling architectures. Section 2.2 describes the four designs. In this table, "Front-End" refers to the machine that dispatches clients to the server that will handle their request. (In the case of DNS load balancing, this refers to the DNS server.) "Back-End" refers to one of the n instances (for example, of a web server) that can handle the client's request. A full discussion of the criteria is in Section 3.

Application scope. Does this design work only for the web, or for every kind of Internet service?

Scale limitations. What is the crucial scale-limiting factor of the design?

Client affinity. Different load distribution strategies have different effects on how consistently a client binds to a particular server. What behavior must the garage innovator expect?

Scale-up and Scale-down time. How long does it take to expand and contract the server farm?

Response to failures. How many users do typical failures affect? What's the worst-case effect of a single failure?

Table 1 has a concise summary of the discussion in this section. Roughly speaking, the rows of Table 1 correspond to paper sections §3.1–§3.6; the columns correspond to §2.2.1–§2.2.4.

3.1 Application Scope

The first and most basic question of any scalability strategy is: *will it work with my application?*

The Bare SDN has the most restrictive application model. Services like S3 have a specific, narrow interface: they serve static content via HTTP. Though many sites contain large repositories of static content, most are not *exclusively* static, so the bare SDN is rarely the complete story.

The HTTP Redirector is slightly wider in scope. Redirection only works with HTTP (and, perhaps, a very small number of other protocols with a redirection primitive). However, unlike with an SDN, clients can be redirected to servers that can run user code, facilitating dynamic web services. However, this technique does not work for protocols that have no redirection primitive, such as FTP, SSH, SMTP and IRC.

L7 load balancers understand a specific application-layer protocols such as HTTP and FTP, and thus are constrained by their vocabulary.

DNS load balancing and **L4 load balancers** work with all applications. DNS works broadly because most applications use a common resolver library that iterates through DNS A-records until it finds one that works. L4 load balancers work broadly because they operate at the IP layer, thus working with any application protocol running over IP without cooperation from the client.

3.2 Scale Limitation

A crucial consideration is scaling limits: what bottleneck will we first encounter as the load increases?

SDNs have implementation bottlenecks that are, to put it simply, not our problem. The two main SDNs today have service level agreements that make scaling *their* responsibility. A garage innovator can pretend the capacity is infinite.

HTTP redirection is involved only at the beginning of each client's session, and thus its scaling limit depends on the typical duration of client sessions. Longer sessions amortize the cost of doing the redirection during session setup. Our experience is that redirection is so cheap that, for typical applications, it scales to thousands or more clients. To evaluate this hypothesis, we built and measured a load-balancing, server-allocating HTTP redirector, described in Section 4.1.

L4/L7 load balancing is limited by the forwarder's ability to process the entire volume of client traffic. How this limit affects a web service depends on the service's ratio of bandwidth to computation. Sites that do very little computation per unit of communication, such as video sharing sites, are likely to be quickly bottlenecked at the load balancer—especially a load balancer that is built from commodity hardware. Conversely, sites that compute-intensive and communication-light, such as a search engine, will be able to use an L4 load balancer to support far more users and back-end servers.

DNS load balancing has virtually no scaling limit for garage innovators. Our experiment described in Section 4.5 suggests that thousands of back-end servers can be named in a DNS record. A service that requires more than several thousand servers is safely out of the garage regime. One tangle is that the success of DNS-based load balancing depends on sane client behavior. Most clients behave well, selecting a random server from a DNS record with multiple IP addresses. Unfortunately, as we will see in Section 4.4, some client resolvers defeat load balancing by apparently *sorting* the list of IP addresses returned, and using the first!

3.3 Client Affinity

Consider a single client that issues a long series of related requests to a web service. For example, a user might log into a web-based email service and send dozens of separate requests as he reads and writes mail. The implementation of many such applications is easier and more efficient if related requests are handled by the same server. Some load balancing techniques can enforce this property; others do not.

SDNs provide a simple contract: the client requests an object by URI, and the SDN delivers the entire object in a single transaction. The SDN is responsible for fulfilling the request, regardless of where it arrives.

HTTP redirection provides strong client affinity because a user is sent to a specific machine at the beginning of each session. The client will continue using that server until the user explicitly navigates to a new URL.

L4 balancers, in principle, could map each client by its source IP address. In practice, however, NAT may hide a large population of clients behind a single IP address, confounding the balancer's ability to spread load. Conversely, a user behind multiple proxies may send a series of requests that appear to be from different IP addresses. Absent source address mapping, the L4 balancer can provide no affinity, and thus the back-end service accepts the responsibility to bring the data to the server (Section 5.3) or vice versa (Section 5.2).

L7 balancing works transparently for any client that implements HTTP cookies or a service-specific session protocol well enough to maintain the session when interacting with a single server. Because it can identify individual sessions, the L7 balancer can enforce client affinity.

In the case of **DNS load balancing**, however, clients and proxies misbehave in interesting ways that confound client affinity. DNS resolvers seem to cluster around a particular address when we would rather they didn't (Section 4.4); and many browsers implement "DNS pinning" to mitigate cross-site scripting attacks [18]. Despite these properties, client browsers cannot be relied upon to show affinity for a particular server, as we describe in Section 4.6.

In summary, HTTP redirection and L7 balancing can enforce client affinity. For L4 balancing and DNS balancing, we recommend that the service assume the front end offers no client affinity.

3.4 Scale-Up and Scale-Down Time

Scale-up time of the server farm is a crucial concern: can a farm grow quickly enough to meet the offered load of a flash crowd? Scale-down time, on the other hand, does not usually affect the user experience; it is important only for efficiency's sake. If the system can not quickly scale back down after a period of high load, the service is needlessly expensive: our innovator is paying for resources she doesn't need.

Bare SDNs have essentially instantaneous scale-up and scale-down time. Services like S3 always have enormous capacity to handle the aggregate load of all their customers. The magic of statistical multiplexing hides our garage innovator's peak loads in the noise.

HTTP Redirectors and **L4/L7 Load Balancers** have identical scale-up and scale-down behavior. Once the decision to increase capacity has been made, these systems must first wait for a new virtual machine instance to be created. Anecdotally, our experience with Amazon EC2 has shown this usually happens in about one minute. The moment the VM has been created, the load balancers can start directing traffic to them.

Scale-down time is a bit more difficult to pin down precisely. Once a scaling-down decision has been made, the load balancers can immediately stop directing new (incoming) sessions to the machines that are slated for termination. However, *existing* sessions, such as slow HTTP transfers or mail transactions, will likely be in progress. To avoid user-visible disruption, these long-lived sessions must be allowed to complete before shutting down the machine. In many cases, transport-level session terminations are hidden from the user by clients that transparently re-establish connections. A pedant might insist that the scale-down time is really the *worst-case* session length. Chen et al. [7] explore how to minimize the number of disrupted long-lived sessions by biasing the allocation of new sessions to servers even before the scale-down decision has been made.

DNS load balancing is the most problematic in its control over load balancing. Recall that in this scheme, back-end server selection is performed by the *client*—it selects a host from the multiplicity of DNS A-records it receives. These records are cached in many places along the path from the garage's DNS server to the client application. Unlike the situation with HTTP redirectors and L4/L7 load balancers, the entity making the load balancing decision does not have a fresh view of the servers that are available. This has a negative effect on scale-up time. While the new DNS record can be published immediately, many clients will still continue using the cached record until after the DNS TTL expires. Fortunately, the nature of a flash crowd means that most of the traffic is new. New users are more likely to have cold caches and thus see the new servers.

Scale-down time for DNS load balancing is even more problematic. As the disheartening study by Pang et al. showed [28], nearly half of clients inappropriately cache DNS entries far beyond their TTL, sometimes for as long as a day. Anecdotally, we have seen this effect in our deployments—servers continue to receive a smattering of requests for several days after they're removed from DNS. Therefore, to ensure no clients see failed servers, we must wait not just for the worst-case session time, but also the worst case DNS cache time.

3.5 Effects of Front-End Failure

Many distributed systems are vulnerable to major disruption if the nodes responsible for load balancing fail. We call the first-encountered node in the load balancing system the "front end." What happens when front-end nodes fail?

The SDN, being a large-scale and well-capitalized resource, typically has multiple, redundant, hot-spare load balancers as its front end. Failure is unlikely.

L4 and L7 load balancers are highly susceptible to failure; they forward every packet of every session. If a single node provides load-balancing and fails, the system experiences total failure—all new and existing sessions stop. If there are m load balancers, the effect of a failure depends on how the front-ends are, themselves, load balanced. If they are fully redundant hot spares (common with expensive dedicated hardware), there will be no effect. Companies like FlexiScale do offer this service, at utility pricing, as we mentioned in Section 2.2.4.

More commonly, redundant L4/L7 front-ends are DNS load-balanced. In this case, $1/m$th of sessions experience up to a three minute delay (see our experiment in Section 4.3) until they fail over to another front-end. $1/m$ is often large because m is often small; m is small because front-end redundancy is typically used for failure resilience, not scaling.

HTTP Redirectors fail in almost exactly the same way as L4/L7 load balancers, with one exception: *existing* sessions are not affected. The redirector is no longer in the critical path after a session begins. *New* sessions have the same failure characteristics as in the L4/L7 balancer case.

DNS load balancing is also highly susceptible to failure if there is only one authoritative nameserver. Nameserver caches will be useful only in rare cases because the TTLs must be kept low (so as to handle scale-up, scale-down, and back-end node failures). Few new sessions are likely to succeed. Existing TCP streams will continue to work, however, since DNS is no longer involved after session setup.

This gloomy-sounding scenario for DNS can, however, be easily overcome. As we mentioned in Section 2.1.4, DNS replication services are plentiful and cheap. Widespread replication of DNS is easy to achieve, even on a garage budget. Furthermore, as we demonstrated in one of our microbenchmarks (Section 4.2), most DNS clients recover from DNS server failures *extremely* quickly—in our experiment, 100% of DNS clients tested failed over to a secondary DNS server within 2.5 seconds. This means that a front-end failure has virtually no observable effect.

This scenario is the most compelling argument for DNS load balancing: At very low cost, there is no single point of complete failure.

3.6 Effects of Back-End Failure

We next consider how the load-balancing scheme affects the nature of user-visible disruptions to the service when a back-end node fails. Recall that by "back-end" node, we mean a member of the n-sized pool of machines that can accept connections from clients. (Distinguish this from the "front end" of the load balancing scheme, which is the load balancer itself.)

The SDN is managed entirely by the service provider, so its back-end failures are not a concern to the garage innovator. The architecture of S3 is said to be highly redundant at every layer. Our experience is that occasional writes do fail (perhaps 1% of the time) but nearly always work on the first retry.

HTTP redirector and **L4/L7 load balancers** offer the best performance for garage-written services in the case of back-end node failure. *Newly arriving sessions* see no degradation at all: the redirector or load balancer knows within moments that a back-end node has failed, and immediately stops routing new requests to it. *Existing sessions* see only transient failures. Users of an HTTP-redirected service who are stuck on a dead node might need to intervene manually (that is, go back to the dispatching URL, such as `mail.live.com`). Load-balanced services potentially see only a transient failure. If the client tries to re-establish the failed TCP connection to what it thinks is the same host, it will be transparently forwarded to an operational server.

DNS load balancing suffers the worst performance in the case of back-end failure. Unlike load balancers and HTTP redirectors, which stop requests to a failed server immediately, DNS load balancing can continue to feed requests to a failed server for more than a day, as we saw in the previous section. If n servers are deployed, $1/n$th of sessions will be unlucky enough to pick the failed server. Unfortunately, when this happens, our experiments have shown that some combinations of client and proxy take up to three minutes to give up and try a different IP address; see Section 4.3.

4 Experimental Micro-Benchmarks

In this section, we flesh out the details of some of the more complex experiments we performed in support of our analysis in the previous section.

4.1 An EC2-Integrated HTTP Redirector

To better understand HTTP redirection performance,we built and evaluated a load-balancing HTTP redirector. It monitors the load on each running service instance, resizes the farm in response to load, and routes new sessions probabilistically to lightly loaded servers.

Servers send periodic heartbeats with load statistics to the redirector. The redirector uses both the presence of these heartbeats and the load information they carry to evaluate the liveness of each server. Its redirections are probabilistic: the redirector is twice as likely to send a new session to one server whose run queue is half as long as another's. When the total CPU capacity available on servers with short run queues is less than 50%, the redirector launches a new server; when the total CPU capacity is more than 150%, the redirector terminates a server whose sessions are most stale.

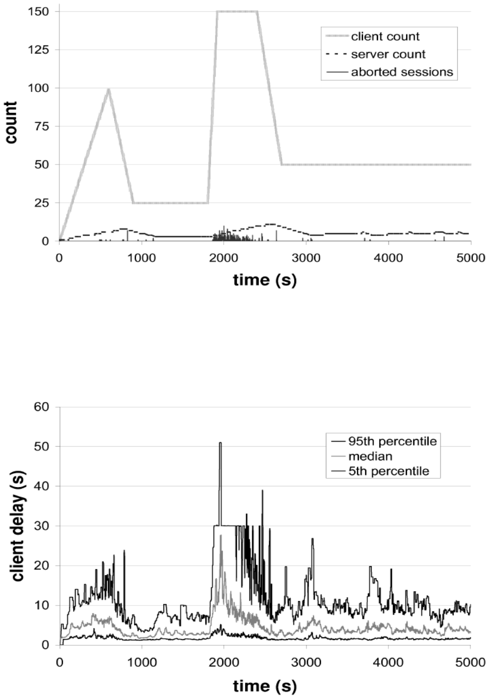

Figure 1: HTTP redirect experiment (§4.1). As client load spikes, the redirector launches new servers and directs new sessions to them.

The servers in the experiment run our Inkblot web service (see Section 5.3). The client load is presented from a separate machine. Each client simulates a user session with a state machine: It logs into the service, accesses random features for a random duration, and then logs out. Each such client session first accesses the service by the redirector's URL. Each session is recorded as having completed successfully or having been interrupted by a failure such as an HTTP timeout.

The top line in Figure 1a shows how we varied the number of simulated clients as the experiment evolved, and the dotted line shows the number of servers allocated by the redirector to handle the demand.

Figure 1b shows the 5th, 50th, and 95th percentiles of client latency; the server run queues (not shown) track these curves closely as Little's result predicts [20]. Our simplistic redirector allocates one server at a time, bounding its response rate to a slope of one server every 90–120 seconds. Around 2000 seconds, the load grows quite rapidly, and client response time suffers badly, with many sessions aborting (bottom curve in Figure 1a).

During these experiments, the redirector consumed around 2% of its CPU when serving 150 clients. Thus, for this application, we expect to be able to serve 7,500

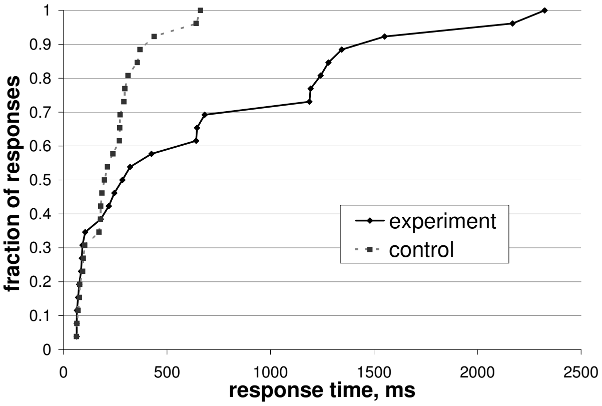

Figure 2: DNS servers fail over very quickly when an upstream server fails (§4.2).

client sessions per redirector node. However, this ratio is sensitive to our choice of client workload simulation parameters. In an application where sessions are longer and require only a single redirection during initialization, the overhead of the redirector will shrink. The important observation is that the redirector is simple, cheap, and applies to many web-based applications.

4.2 DNS server failover behavior

To determine the delay associated with nameserver failure, we configured a DNS subdomain with two NS records: one legitimate and the other pointing at an IP address with no DNS server. As a control, we repeated the same experiment with a fresh subdomain containing only the NS record for the functioning server. In each case, we sent recursive queries to 26 geographically-distributed public DNS servers at seven ISPs. Figure 2 shows the cumulative distribution of response times. The worst-case response time was less than 2.5s; of course, this cost is incurred only on a DNS cache miss.

4.3 Web client DNS failover behavior

To determine the user-visible effect of DNS-based recovery from failed back-end servers, we set up the following experiment: A browser script resolves a series of fresh hostnames, each of which resolves to two fresh IP addresses. One IP address has a live web server; the other does not. We vary the numerical order of the IPs to avoid biasing misbehaving resolvers (Section 4.4). The script repeats this experiment 20 times.

Often, the browser is lucky and tries the operational IP address first. When it doesn't, the user experiences delays from 3 to n seconds, as shown in Table 2. The delay appears to be coupled to the OS resolver library rather than the browser. We repeated the experiments with the browser behind a Squid 2.6.16 proxy running on Linux 2.6.17, where the delay seemed to be defined by the proxy.

The initial experiment used fresh names and IP addresses to ensure the client DNS resolvers always had cold cache behavior: If resolvers cache knowledge of failure, then the experiment would show no delay. Surprisingly, in some configurations (both the unproxied browsers on Linux and any browser using a Linux Squid cache) the resolver behaved *worse* with a warm cache than with a cold one. Queries that were slow in the first run showed no change. However, on queries that were fast in the first run exhibited delays of 20–190s. We are unsure why these configurations exhibit different timeout behavior for a cold request than for a warm one.

To summarize: Depending on operating system and proxy cache configuration, using DNS to failover from a non-responsive server causes clients experience delays from 3 to 190 seconds.

4.4 Badly-behaved resolvers defeat load balancing

This same experiment also revealed that in every configuration except the two browsers running uncached on Windows XP, the browser (or proxy) always tried the lower-numbered IP address before the higher-numbered one, regardless of the order they were returned by the DNS server. This behavior has important implications for using DNS as a load-balancing mechanism.

A conventional DNS server always returns the same complete list of machines. With this approach, many machines will flock to the lowest-numbered IP address in the list. Ignoring proxies, the 8% of the desktop market running MacOS X will cause significant problems once the clients present 12 server's worth (that is, 1/0.08) of load.

Alternately, a custom DNS server might return only a single IP address to each query, exchanging fault tolerance for load balancing. A further refinement returns k servers of the available n. This provides fault-tolerance against $k - 1$ failures, but ensures that even the badly-behaved resolvers spread themselves across the first $n - k$ machines.

4.5 The maximum size of DNS replies

The scalability of DNS load balancing is potentially limited by the number of "A" (Address) records that can be put into a single DNS reply. The DNS RFC only guarantees transmission of a 512-byte reply when in UDP mode [23][§2.3.4], and some past work has reported that DNS clients do not always fall back to using TCP to retrieve larger responses [16]. At 16 bytes per A-record, and counting other overhead, a 512 byte DNS reply would limit us to about 25 back-end servers—potentially a bottleneck.

To test this limit in practice, we populated a single name with over 2200 A records, and then probed it via

OS	Browser	no-proxy delay (s)	proxied delay (s)
Windows XP	Firefox 2.0.0.6	21	9–18
Windows XP	IE 6.0	21	9–12
Linux 2.6.17	Firefox 2.0.0.3	3	9
Linux 2.6.17	Epiphany 2.16	3	*not measured*
MacOS X 10.4.10	Firefox 2.0.0.11	15–75	9–17
MacOS X 10.4.10	Safari 2.0.4	12–75	9–18

Table 2: DNS timeout test for web sites that have multiple A records. The table reports timeouts observed with various browser and OS combinations before the client attempts to use a second IP address if no reply is received from the first.

26 public DNS servers in seven ISPs. Every server correctly returned the entire list, indicating its client had used TCP DNS transfers to fetch the complete reply from our nameserver. We have not explored whether some clients (e.g., web browsers) will fail to fetch large replies. This result suggests that DNS implementations pose no limit to the number of back-end servers served by a DNS load balancing solution; indeed, other limits of DNS behavior obviate including the entire server list in DNS replies (Section 4.4).

4.6 Client affinity observations

One of our real services (Section 5.2) is equipped to tolerate client non-affinity, even though we expected affinity to be the common case. We instrumented the service to measure client affinity, and sampled for a period of two months in a 2-server configuration. About 5% of the requests received by our web service arrive at the “wrong” server. From this, we infer that 10% of clients exhibited no affinity, and half of the requests such clients generate arrived at the “right” server by chance.

5 Application Design and Flash Crowd Experiences

In this section, we report three case studies demonstrating the design, implementation, and flash-crowd response of “garage-scalable” web services. Over the past year, our research group created three web sites that experienced sudden surges in popularity, including one that was (literally) Slashdotted. For each web site, we describe our design and implementation decisions, report how they responded to a flash crowd, and extract lessons learned from the experience.

5.1 MapCruncher

In May of 2006, our research group developed MapCruncher [10], a new web authoring tool that makes it easy for non-experts to convert their own maps into AJAX-style interactive maps.

The output of this tool is a set of static content (ordinary .png, .html and .js files) that require no special server-side support. Serving an interactive map generated by MapCruncher requires nothing more from an HTTP server than its most basic function: reading a file off disk and returning it to a client in response to an HTTP GET. All of the dynamic behavior of the application is implemented in the client browser.

5.1.1 The Web Site

To show off MapCruncher’s functionality, we created a gallery of sample maps: about 25 gigabytes of image data spread across several hundred thousand files. We put this archive on reasonably powerful web server: a 2005-vintage Dell PowerEdge 2650 with 1GB of RAM, a 2.4 GHz Intel Xeon processor, and several SCSI disks, running IIS 6.0. We did not anticipate performance problems since IIS is performant and the content we were serving was all static.

5.1.2 The Flash Crowd

After MapCruncher was released, Microsoft published a press release describing it, which was picked up by various bloggers and Internet publications. Crowds soon arrived. Nearly all visitors viewed our map gallery, which became unusably slow. Our server logs showed that, at peak, we were serving about 100 files per second. (Far more images were probably requested.)

We were surprised that our web server had failed to keep up with the request stream until we realized the dataset was many times larger than the machine’s memory cache. In addition, there was very little locality of reference. Our sample maps were enormous, and each visitor zoomed into a different, random part of one. The resulting image requests were almost entirely cache misses. We could not serve files any faster than the disk could find them. Perhaps not coincidentally, the sum of typical seek time, settling time, and rotational delay of modern disks is about 10ms (100 requests per second).

The next day, we published our maps to Amazon S3, and had no further performance problems. The lesson we learned was the power of an SDN’s statistical multiplexing: Rather than one disk seeking, the SDN spread files across huge numbers of disks, all of which can be seeking in parallel. Rather than one buffer cache thrashing during peak load, the SDN dedicated gigabytes of buffer cache from dozens of machines to us.

S3's utility-computing cost model was as compelling to us as to a garage innovator. During a normal month, we pay virtually nothing; the nominal cost of storing our 25GB gallery is under $4/month. In the case of a flash crowd, we pay a one-time bandwidth charge of about $200. Statistical multiplexing makes it economically viable for Amazon to charge only for capacity used. This is much more efficient than the traditional model of paying for a fixed reserve that remains idle before the flash crowd, and yet may still be insufficiently provisioned when a crowd finally arrives.

5.2 Asirra

In April of 2007, our research group designed a web service called Asirra, a CAPTCHA that authenticates users by asking them to identify photographs as being either cats or dogs [9]. Our database has over 4 million images and grows by about 10,000 every day. Images come to us already classified (by humans) as either cat or dog, thanks to our partnership with Petfinder.com, the world's largest web site devoted to finding homes for homeless animals. Petfinder provides the Asirra project ongoing access to their database as a way to increase the number of adoptable pets seen by potential new owners.

5.2.1 The Web Site

Asirra's metadata must remain secret to maintain the security of the CAPTCHA. Consequently, Asirra is implemented as a web service rather than a distributable code library. It has a simple API that allows webmasters to integrate our CAPTCHA into their own web pages. Webmasters include Asirra's JavaScript in their HTML forms, where it first creates the visual elements of the challenge in the user's browser. It then sends AJAX requests to our web service to create a new Asirra session, retrieve one or more challenges, and submit user responses for scoring. If the response is correct, the client earns a "service ticket" (an unguessable string) and presents it to the webmaster in a special field of the HTML form. Because the client is not trusted, the webmaster's back-end then verifies that the ticket presented is valid by checking it with a different interface of Asirra's web service. The web service is implemented in Python and currently deployed at Amazon EC2.

There were several issues we considered in deciding how to enable Asirra to scale. The first was how to scalably store and serve the images themselves. The CAPTCHA's growing collection of 4 million JPEG images consumes about 100GB. Based on our experience with MapCruncher, using Amazon's S3 was an easy choice.

The second consideration was how to enable scalable access to all the metadata by every service instance. The master copy of the metadata is stored in a SQL Server database at our offices at Microsoft. However, as we discussed in Section 2.1.5, it is difficult to make a fully relational database arbitrarily scalable. We solved this in Asirra by observing that the web service treats the image metadata as read-only. (The only database writes occur off-line, when a nightly process runs to import new images into the database.) In addition, the web service does not need complex SELECT statements; when a CAPTCHA request arrives, Asirra simply picks 12 image records at random. We therefore decided to keep our fully relational (and relatively slow) SQL Server database in-house. Every time our off-line database update process runs, it also produces a reduced BerkeleyDB [27] database, keyed only by record number, that contains the relevant information about each image. (BerkeleyDB is essentially an efficient B-tree implementation of a simple key-value store.) The BerkeleyDB file is then pushed out to each running web service instance, which does local database queries when it needs to generate a CAPTCHA.

The third and most interesting design question was how to maintain session state. In between the time a user requests a CAPTCHA and the time the CAPTCHA is scored, Asirra must keep state in order to recognize if the answer was correct, and do other accounting. One possibility was to keep each session's state in its own S3 object, but we found that S3's write performance was somewhat slow; requests could take up to several seconds to complete. We next considered storing session state locally—on individual servers' disks. This led to an interesting question: how does session state storage interact with load balancing?

Client load is distributed across Asirra's server farm using the DNS load balancing technique described in Section 2.2.2. The first action performed by a client is session creation. Whichever machine is randomly selected by the client to execute this action becomes the permanent custodian of that session's state. The custodian stores the session state locally and returns a session ID to the client. The session ID has the custodian server's ID embedded in it.

As we discussed in Section 4.6, one of the disadvantages of DNS load balancing is that clients are not guaranteed to have affinity for back-end servers. Any session operation after the first one may arrive at a server other than the session's custodian. We address this by forwarding requests from their arrival server to the custodian server, if the two servers are different. That is, the arrival server finds the custodian server's ID embedded in the session ID, reissues the request it received to the custodian, and forwards the response back to the client.

Since the client is not trusted, session IDs are unguessable strings; a forged session ID will fail to find a corresponding session. Forging the identity of the custodian server will cause a request to be unnecessarily forwarded, but no corresponding session state will be found there. (As further protection, Asirra only forwards requests to servers that appear in a list of valid custodians.)

Our forwarding scheme ensures that at most two machines are ever involved in servicing a single request: the machine which receives the request from the client, and the machine that owns the session state and receives the sub-request. Asirra service is therefore readily scalable; the overhead of parallelization will never be more than 2x regardless of the total size of the farm.

In practice, we have observed lower overhead for two reasons. First, compared to satisfying a request, forwarding one takes very little time and requires no disk seeks. Even if every request required forwarding, the total overhead might not be more than 1.1x. Second, we have found that request forwarding is not the common case; as we described in Section 4.6, the rate of client affinity "failures" is about 10%.

5.2.2 The Flash Crowd

Shortly after its release, Asirra was shown publicly at an annual Microsoft Research technology showcase called TechFest. It received significant coverage in the popular press which resulted in a load surge lasting about 24 hours. During this time we served about 75,000 real challenges, plus about 30,000 requests that were part of a denial-of-service attack. Over the next few months, we saw a gradual increase in the traffic rate as sites began to use the CAPTCHA.

We learned several interesting lessons from this deployment. The first, as discussed in the previous section, was that poor client-to-server affinity was not as much of a problem for DNS-load-balanced services as we had initially feared. Second, there were some pitfalls in using EC2 as a utility for providing web services. Most problematic is that when EC2 nodes failed, as happens from time to time, they also gave up their IP address reservations. (This weakness of EC2's service was later corrected, in April 2008.) This is a problem when using DNS load balancing. As we saw in Section 3.4, a failed node can produce user-visible service degradation until all DNS caches—even the badly behaved ones—are updated. Also, recall that local storage on EC2 nodes is fast, but not durable. Though data can be cached locally, it is vital to keep anything valuable (e.g., user data, log files, etc.) backed up elsewhere.

The denial-of-service attack provided what was, perhaps, the most interesting lesson. In the short term, before a filtering strategy could be devised, the easiest defense was simply to start more servers. The solution required no development time beyond the scalability work we'd already done, and only cost us a few extra dollars for the duration of the denial-of-service attack. Before we had a chance to implement a denial-of-service filter, the attacker became bored (and, perhaps, frustrated that his attack was no longer working) and stopped his attack. We never actually got around to implementing a denial-of-service filter—a fascinating success of "lazy development." (The Perl community has been preaching laziness as a virtue for years!) As we will see shortly, this lesson had a surprising influence on the design of our next service.

5.3 InkblotPassword.com

In November 2007, our research group deployed InkblotPassword.com [34], a website that helps users generate and remember high-entropy passwords, using Rorschach-like images as a memory cue. The site lets users create accounts and associate passwords with inkblot images. Our site is an OpenID [30] authentication provider; users can use their inkblot passwords to log in to any web site that accepts OpenID credentials. Note that Inkblot must store dynamically generated information (the user accounts) durably. This requirement sets it apart from our previous two applications, which had static (pre-computed) databases and *ephemeral* state.

5.3.1 The Web Site

Like Asirra, we implemented Inkblot in Python. However, unlike Asirra, we spent virtually no time optimizing its performance. The denial-of-service attack we suffered taught us a valuable lesson: Now that it's so cheap to run lots of servers for a day or two, there is no need to spend time on problems that can be solved that way. We reasoned that if our goal was simply to handle a single unexpected flash crowd, the best strategy was to forgo careful code optimization and simply plan to start plenty of extra servers for the duration of the load spike. If *ongoing* popularity and high nominal load followed, careful code optimization would then be economical.

Another difference between Asirra and Inkblot was our decision to store both the persistent user database and the ephemeral session state in S3; nothing was stored on the local disk. We chose S3 for the user database, despite its slowness we observed in Asirra, because of the requirement for database persistence. Fortunately, the particular write requirements of our application permit write-behind without exposing security-sensitive race conditions, hiding most of the write delay from users. We stored ephemeral session state in S3 entirely because of our new laziness philosophy: although less efficient than using the local disk, reusing the user-state storage code led to faster development.

Like Asirra, Inkblot was implemented and tested to run on multiple servers. We deployed it with two servers, to ensure that we were exercising cross-server interaction as the common case. DNS A-records provided load balancing among the servers. Updating our institutional DNS service required interacting with a human operator, so no automatic scaling was in place.

5.3.2 The Flash Crowd

Days after its release, Network World penned an article covering Inkblot [12]. That coverage was propagated to other tech magazines and blogs.

We had the very good fortune to be in a boring meeting the next day, when one of us happened upon the article about Inkblot moments after it appeared on the front page of Slashdot. We tried clicking through the link, and found our service unresponsive. Unfortunately, this happened before we implemented the code described in Section 4.1 that automatically expands the farm in response to load. The one responsive server reported a run queue length of 137; in a healthy system, it should be below 1.

Within minutes, we spun up a dozen new servers. We submitted a high-priority DNS change request to our institutional DNS provider which was fulfilled within half an hour. The new servers saw load almost immediately after the DNS update, and the original servers recovered in another 20 minutes. (The DNS TTL was one hour at the time of the Slashdotting.) For several hours, all 14 servers' one-minute-averaged run queue lengths hovered between 0.5 and 0.9. The site remained responsive. By the end of the day, the Inkblot service had successfully registered about 10,000 new users.

We kept the extra servers up for two days (just in case of an "aftershock" such as Digg or Reddit coverage). We then removed 10 out of the 14 entries from the DNS, waited an extra day for rogue DNS caches to empty, and shut the 10 servers down. The marginal cost of handling this Slashdotting was less than $150.

We were fortunate to survive the flash crowd so well, considering that our load-detection algorithm was "good luck." Indeed, this experience prompted us to carefully examine the alternatives for filling in the missing piece in that implementation; that examination led to the analysis and experiments that comprise this paper.

6 Conclusions

This paper surveys the contemporary state of utility computing as it applies to the low-capital garage innovator. It describes existing, utility-priced services. Our analysis characterizes four approaches to balancing load among back-end servers. We exhibit six experiments that highlight benefits and limitations of each approach. We report on our experiences deploying three innovations in garage style, and how those various deployments strategies fared the flash crowds that followed.

We conclude that all four load balancing strategies are available to the garage innovator using utility resources, and that no single strategy dominates. Rather, the choice of strategy depends on the specific application and its load and fault-tolerance requirements.

7 Acknowledgements

The authors wish to thank John Douceur for reviewing drafts of this paper, and MSR's technical support organization for deployment help.

References

[1] Akamai Technologies, Inc. Edgeplatform. http://www.akamai.com/.

[2] Amazon Web Services. EC2 elastic compute cloud. http://aws.amazon.com/ec2.

[3] Amazon Web Services. SimpleDB. http://aws.amazon.com/simpledb.

[4] Berry, G., Chase, J., Cohen, G., Cox, L., and Vahdat, A. Toward automatic state management for replicated dynamic web services. In *Netstore Symposium* (Oct. 1999).

[5] Brisco, T. DNS Support for Load Balancing. RFC 1794 (Informational), Apr. 1995.

[6] Challenger, J., Iyengar, A., Witting, K., Ferstat, C., and Reed, P. A publishing system for efficiently creating dynamic web content. In *INFOCOM 2000 Conference* (Mar. 2000).

[7] 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. In *to appear, Networked Systems Design & Implementation* (2008).

[8] Date, C. J. *An Introduction to Database Systems*, 8th ed. Addison-Wesley, 2004.

[9] Elson, J., Douceur, J. R., Howell, J., and Saul, J. Asirra: a CAPTCHA that exploits interest-aligned manual image categorization. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS), Alexandria, Virginia, USA* (2007), P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds., ACM, pp. 366–374.

[10] Elson, J., Howell, J., and Douceur, J. R. Mapcruncher: integrating the world's geographic information. *Operating Systems Review 41*, 2 (2007), 50–59.

[11] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.

[12] FONTANA, J. Forget sticky notes, microsoft using inkblots as password reminders. http://www.networkworld.com/news/2007/120407-microsoft-inkblots-passwords.html, Dec 2007.

[13] GOLDSZMIDT, G., AND HUNT, G. Scaling internet services by dynamic allocation of connections. In *Proc. Integrated Management (IM 99)* (May 1999).

[14] GRANITE CANYON GROUP, LLC. Public DNS. http://www.granitecanyon.com/.

[15] GRIBBLE, S. D., BREWER, E. A., HELLERSTEIN, J. M., AND CULLER, D. Scalable, distributed data structures for internet service construction. In *Fourth Operating Systems Design and Implementation* (Oct. 2000).

[16] GUDMUNDSSON, O. DNSSEC and IPv6 A6 aware server/resolver message size requirements. draft-ietf-dnsext-message-size-00, June 2000.

[17] HENDERSON, C. *Building Scalable Web Sites*. O'Reilly Media, 2006.

[18] JACKSON, C., BARTH, A., BORTZ, A., SHAO, W., AND BONEH, D. Protecting browsers from DNS rebinding attacks. In *Computer and Communications Security* (October 2007).

[19] JUL, E., LEVY, H., HUTCHINSON, N., , AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 109–133.

[20] KLEINROCK, L. *Queueing Systems*, vol. I. John Wiley & Sons, Inc, 1975.

[21] LIMELIGHT COMMUNICATIONS INC. Limelight networks. http://www.limelightnetworks.com/.

[22] LINUX VIRTUAL SERVER PROJECT. http://www.linuxvirtualserver.org/.

[23] MOCKAPETRIS, P. Domain names - implementation and specification. RFC 1035 (Standard), Nov. 1987.

[24] NEUSTAR. UltraDNS. http://www.neustarultraservices.biz/-solutions/externaldns.html.

[25] NIRVANIX INC. Nirvanix Web Services API developer's guide v1.0. http://developer.nirvanix.com/sitefiles/1000/API.html, Dec. 2007.

[26] NOELDNER, C., AND CULVER, M. Scalable media hosting with Amazon S3. Amazon Web Services Developer Connection, http://developer.amazonwebservices.com/-connect/entry.jspa?externalID=1073, Nov. 2007.

[27] ORACLE CORPORATION. Berkeley DB. http://www.oracle.com/database/berkeley-db.html.

[28] PANG, J., AKELLA, A., SHAIKH, A., KRISHNAMURTHY, B., AND SESHAN, S. On the responsiveness of DNS-based network control. In *Internet Measurment Conference* (2004), A. Lombardo and J. F. Kurose, Eds., ACM, pp. 21–26.

[29] RAYBURN, D. Cdn pricing data: What the cdns are actually charging for delivery. http://tinyurl.com/25muah.

[30] RECORDON, D., AND REED, D. OpenID 2.0: a platform for user-centric identity management. In *Digital Identity Management* (2006), A. Juels, M. Winslett, and A. Goto, Eds., ACM, pp. 11–16.

[31] RIGHT SCALE LLC. Rightscale dashboard. http://info.rightscale.com/.

[32] SCHINDER, T. *ISA Server 2006 Migration Guide*. Elsevier, 2007.

[33] SCHLOSSNAGLE, T. *Scalable Internet Architectures*. Sams Publishing, 2006.

[34] STUBBLEFIELD, A., AND SIMON, D. Inkblot authentication. Technical report MSR-TR-2004-85, Microsoft Research, Aug. 2004.

[35] XCALIBRE COMMUNICATIONS LTD. Flexiscale. http://www.flexiscale.com/.

[36] YU, H., AND VAHDAT, A. Design and evaluation of a continuous consistency model for replicated services. In *International Conference on Distributed Computing Systems (ICDCS)* (Apr. 2001).