Lessons from Giant-Scale Services

Eric A. Brewer
UC Berkeley & Inktomi Corporation
5/20/99

We summarize our experience with many giant-scale services such as web portals, focusing on new ways of thinking about scale, availability, evolution and growth. We also define a basic model that fits most such services and clarifies the fault model and semantics delivered in practice.

1 Introduction

The past few years has seen an explosive growth in the size of infrastructure services, particularly giant web sites and ISPs such as Yahoo!, AOL, Excite and others. Many new players are building these giant-scale services as well, including Microsoft, Disney, NBC, and CNN.

In this paper, we look at the basic model followed by these services and examine three of the very challenging problems encountered by these sites. We focus on generalizing and abstracting the key challenges faced by giant-scale services and summarize the approaches used in practice. This is very much an "experience" paper: most of these issues are not addressed in the academic literature, and most of the conclusions are in the form of principles and approaches rather than quantitative comparisons. Nonetheless, the challenges we explore will become increasingly important as more services arise and an increasing fraction of civilization depends on their operation on a daily basis.

Much of the detailed information of site operation is still considered confidential, so we tend to keep specific examples anonymous and we tend to report trends and approaches rather than specific combinations of features.

There are many reasons for the success of these services, but there are some key technical advantages that lead to their success. Infrastructure services have many fundamental benefits:

Access anywhere, anytime: the infrastructure is ubiquitous, including access from home, work, airports, cafes, and cell phones.

Available via portable or low-cost devices: infrastructure services support a wide range of devices such as set-top boxes, network computers, PDAs and smart phones [cite ericsson]. Because most of the processing is in the infrastructure, these devices can offer far more functionality for a given cost and battery life. Since these can also be accessed via regular PCs, we can use a real keyboard for data input (such as managing your address book). We can also lose the device without losing the data.

Enables groupware: Because the data for many users is centralized, we can easily offer group-based applications such as group calendars, chat, messaging (such as AOL's ICQ with 40M users), and teleconferencing.

Much cheaper overall cost: Although hard to measure, infrastructure services have a fundamental advantage in cost over designs based on device functionality. End-user devices, such as set-top boxes or smart phones, have a very low utilization (less than 4%), while the utilization of resources within the infrastructure is often above 80% [*]. This is effectively a 20x improvement in efficiency for anything moved from the device to the infrastructure. In general, the key is that infrastructure resources can be multiplexed well across the *active* users, while end-user devices serve at most one user (active or not). The centralization of administrative burden also reduces overall cost, but it is much harder to quantify.

Convergence with other infrastructure services: The IP infrastructure is subtly growing to include the voice network, cellular telephone systems, television distribution, and even the global positioning system. This "super convergence" enables many new capabilities such as integrated cell phone, pager, web/email access; location-based services such as maps and driving directions; and sophisticated teleconferencing that includes handouts and PowerPoint slides in addition to audio and video.

Can upgrade/add services in place: Perhaps the most powerful advantage in the long term is the ability to upgrade services or offer new services easily. Traditional applications and devices require physical distribution that is expensive and awkward. With infrastructure services, devices last longer and grow in usefulness over time. Web TVs are a good example: they benefit automatically from every new web service, without any physical redistribution for updates.

1.1 Infrastructure Services Use Clusters

Given the scale requirements, all giant-scale services use clusters. Table 1 shows some representative clusters and their traffic. As another example, a single infrastructure hosting site run by Exodus Communications houses several thousand nodes that support more than 40 different services [cote exodus]; AOL's new US$520M data center will be more than 220,000 square feet and filled almost entirely with clus-

Service	Nodes	Queries	Notes
AOL web cache	>200	2.5B/day	4-CPU DEC 4100s
Inktomi Search Engine	800	>40M/day	2-CPU Sun Workstations
Geocities	>300	>25M/day	PC Based
Anonymous web-based e-mail	>200	25M/day	FreeBSD PCs

Table 1: Example Clusters for Giant-Scale Services

ters [AOL99]. We will thus assume the use of clusters for giant-scale services, but it is useful to review the four driving forces:

Absolute Scalability: A successful network service must scale to support a substantial fraction of the world population. It is expected that most of the developed world, about 1.1 billion people, will have some form of infrastructure access in the next ten years. Furthermore, online time per user and queries/user/day are also going up. [cite AOL]

Cost/performance: Although a traditional reason for using clusters, cost/performance of the hardware is not really an issue for giant-scale services: there is no alternative solution to clusters that can match the required scale, and hardware cost is typically dwarfed by bandwidth and operational costs.

Independent Components: Users expect 24-hour service from systems consisting of thousands of hardware and software components. Transient hardware failures and software faults due to rapid system evolution are inevitable. Clusters simplify the problem by providing largely independent faults. Much of this paper focuses on how to leverage this independence into high availability.

Incremental Scalability: the uncertainty and expense of growing a service leads a strong desire for small *incremental* scaling as needed, preserving and augmenting existing investment. A node should last its entire three-year depreciation lifetime, and in general should be replaced when it no longer justifies its (expensive) rack space compared to new nodes.

Although these advantages are a useful starting point, they are only the start. This paper is essentially about bridging the gap between these basic attributes the real-world scalability and availability required by giant-scale services.

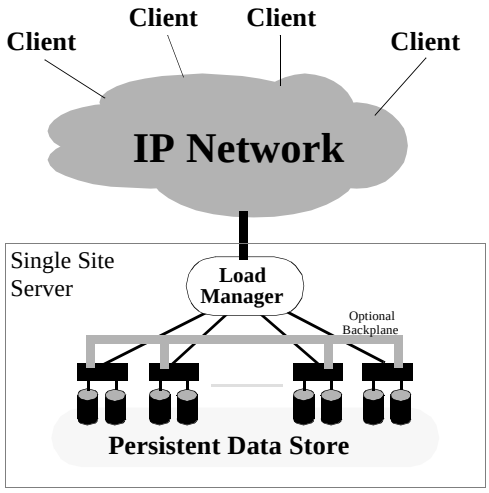

Figure 1: The Basic Internet Model

1.2 Challenges

There are significant challenges to deploying real infrastructure services. In this section we introduce the three challenges addressed in this work:

Load Management:

High Availability:

Evolution & Growth:

There are many important challenges that we explicitly do not address, either because they have been covered elsewhere or because their solution is very service specific. Some of these challenges include service monitoring and configuration [*], bandwidth and network quality of service, burstiness, logging and log analysis, and personalization.

After defining a basic model in Section 2 that fits most of the giant-scale services, we use it to focus discussion on our three big challenges: load management in Section 3, high availability in Section 4, and service evolution and growth in Section 5. We summarize our conclusions in Section 6.

2 The Basic Internet Model

Our basic model for infrastructure services in shown in Figure 1. The goal of the model is to enable discussion of the issues facing these services and to capture the key elements of giant-scale servers in practice. We describe the role of each component and the impact they have on overall service requirements. One very important goal is to clarify the fault model and semantics of these services.

There are many important assumptions in this model. First, we assume that the service provider has little or no control over the clients or the IP Network. In some cases, such as intranets, stronger assumptions may be possible.

In the figure, we show only one site; however we will discuss the use of mirrors for global distribution and disaster tolerance. To first order, they are just replicas of the “Single

Site Server" box, with their own connections to the IP Network.

We also assume that these servers are built out of clusters, as discussed above. The unit of scaling is the *node*, which includes one or more CPUs and some number of internal or external disks. Due to the cost of rack space, the node is often a small PC multiprocessor, with 2–4 CPUs and 2–4 disks.

We also assume that service is driven by queries. This is inherent in most common protocols, including HTTP, FTP, NNTP, POP, IMAP, and variations of RPC. For example, the basic primitive of HTTP is the "get" command, which is by definition a query. Sometimes a sequence of queries from one client are grouped together: we define a *session* as a group that shares state among the queries. For example, HTTP is a "stateless" protocol and therefore does not have sessions,[1] while FTP requires an initial exchange that is remembered throughout the rest of the session.

We often also assume that these queries are "read mostly", that is, that read-only queries greatly outnumber updates (queries that affect the persistent data store). We will point out cases in which we assume read-mostly traffic.

2.1 Components

There are six components to the basic model:

Clients: The clients initiate the queries; they could be specific to the service, such as stand-alone e-mail readers, or general, such as web browsers.

IP Network: The network is best effort and based on IP. It could be the public Internet or some form of private network such as an intranet.

Load Manager: This component has two purposes. First, it is a level of indirection between the external name of the service and the physical names (IP addresses) of the nodes. This is required to preserve the availability of the external name in the presence of node faults. Second, the load manager balances load among the (up) nodes. We look at this component in more detail in Section 3.

Nodes: The nodes are the workers of the system, combining CPU, memory, and disks into an easy-to-replicate unit. The node is the unit of expansion and often the *field replacable unit*, which means that if anything in the node breaks (including disks), you replace the whole node and deal with the subcomponents off-line. Sometimes failed disks can be replaced without taking the node down.

Persistent Data Store: This is a replicated or partitioned "database" that is spread across the disks of the nodes. It might also include network-attached storage or RAID subsystems. The implementation of this store is a major variable in these systems and will be explored in Sections 3.1 and 4.4.

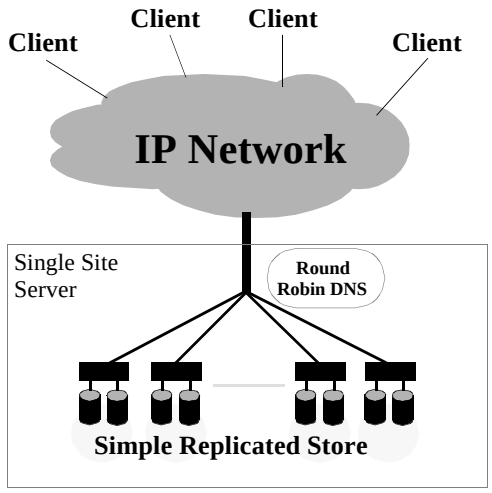

Figure 2: A Simple Web Farm

Optional Backplane: Many services use a system-area-network among the nodes, which functions like a backplane in a multiprocessor. The network handles inter-node traffic such as redirection to the correct node or coherence traffic for the persistent data store.

Auxiliary Systems (not shown): Nearly all services have several other service-specific pieces that we can largely ignore in the basic model. Examples include user-profile databases, ad servers, site management tools, and support for logging and log analysis. Many of these subsystems can be viewed as additional sets of nodes with their own persistent data store.

2.2 Examples

Figures 3 and 4 show two illustrative systems at opposite ends of the complexity spectrum: a simple web farm and complex server similar to a search-engine cluster. They differ in their load management, their use of a backplane, and their persistent data store.

The web farm is shown in Figure 2. First, note that the load manager is not actually in the flow of the traffic. *Round-robin DNS* returns different domain name to IP address mappings for different clients, thus roughly balancing the load at the time of DNS lookup, but providing little support for availability when a node fails (see Section 3.3). Second, the persistent data store is implemented by simple replication of all content to all nodes, which works well when the total amount of content is small. Finally, there is no need for a backplane, since all servers can handle all queries and there is no coherence traffic. In practice, even simple web farms often have a second LAN (backplane) to simplify the manual

1: "Cookies" are a way to simulate sessions in HTTP, but they still don't require the server to preserve state across queries. Instead the relevant state is passed back and forth each time. This can also be done using "Fat URLs", in which client-specific state is embedded in the URL.

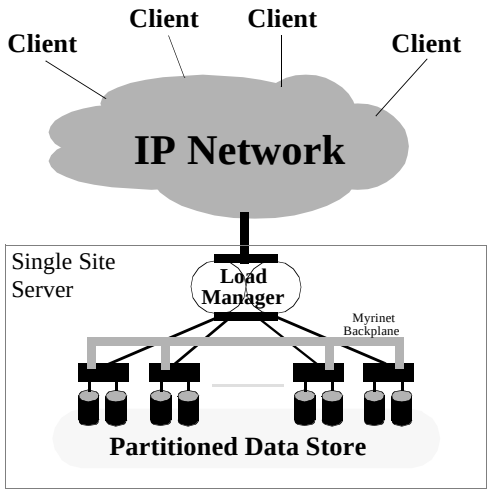

Figure 3: A More Complex Server

updating of the replicas. In this version, node failures reduce the capacity of the system, but not the availability of its data.

A more complex example is shown in Figure 3. The load management actually is in the path of the traffic and therefore has to be fault tolerant. Here we show a pair of "level 4" switches that automatically fail over to each other. These switches, which are available from many vendors, rewrite TCP connections from the external IP addresses to one of the internal node names. They can balance load based on outstanding connections and can respond quickly to failed nodes by avoiding them for new connections.

The persistent store is partitioned across the nodes, possibly without any replication. This means that node failures reduce the effective size of the store as well as its overall capacity. It also means that the nodes are no longer identical and some queries may need to go to specific nodes.

The backplane enables queries to get to the right node or nodes. Sometimes the load manager can pick the right node directly, but this requires service-specific and query-specific knowledge in the load manager. If the service uses caching of data from other nodes, the backplane is used for the cache coherence traffic.

In the case of an Inktomi search engine cluster (there are currently five worldwide), the backplane is Myrinet (1.6 Gb/s) and it connects 100 nodes, each with 2 CPUs and multiple disks. The store is fully partitioned with some replication for key data; different nodes get different amounts of data according to their relative capabilities (nodes differ in age and therefore capability). The backplane is used for subqueries that are merged by the primary node, but there is no caching of remote data other than answers to whole queries.

There are many more examples: nearly every web-based services fits the basic model. The key variations are the load manager, the persistent store, and the auxiliary systems.

2.3 Fault Model and End-to-End Semantics

One of the most important reasons to have a basic model is to look at its fault model and the end-to-end semantics it provides. We also define common extensions to the basic model where appropriate.

There are three basic tenets:

1) Focus on locally measured availability

The first issue is the best-effort nature of the IP Network, which means that a client may be partitioned from the server. To the client, the service is down and it is largely out of the control of the service provider. A trivial example is a broken modem connection at the client, which partitions it from all services.[2] Because of this effect, we distinguish between the *end-to-end availability* and the *service availability*. End-to-end availability is the "correct" measure, at it includes failures in the IP Network that affect end users. Service availability is measured at the service itself (or perhaps just outside it) and is a more useful internal metric tied to the uptime of the service. Service providers thus have direct control over service availability and only limited indirect control over end-to-end availability. End-to-end availability is strictly less than service availability, since it merely adds faults in the clients and IP Network. In practice, providers aim for high service availability and use a large number of independent network connections to decrease the probability of being partitioned from a large number of clients. In the rest of this paper, we will thus focus on service availability.

Extension: client failover to an equivalent server

A partitioned client may be able to reach equivalent servers, such as mirror sites. There is currently no general way for a browser to understand groups of equivalent servers, which is a prerequisite for failover (and wide-area load balancing). Work on "smart clients" [YDE+97] shows how to do this for applets and client-side plug-ins, in which case the service can control some of the code on the client. For HTTP services, a standardized header could solve this, such as "Alternate-hosts: mirror1.com mirror2.com", in which the client can learn about mirror groups when it is not partitioned. Since none of these mechanisms are ubiquitous, we leave this as an extension to the basic model.

2) Reload Semantics: node failures drop the queries in progress at that node

The second tenet of the basic model is that it is OK to lose active connections when a node fails, as long as the probability of success on retry (reload) is high. Thus, the basic model is not fault tolerant, but merely highly available. End-to-end fault tolerance *depends* on the user retrying the

2: This distributed responsibility has several complex side effects. For example, browser manufactures get technical support calls when sites go down, and sites get calls when the user's ISP is down, their browser is broken, or their PC is out of virtual memory.

query, and that query going to a different node that is up (which is the job of the load manager).

Extension: Preserve queries in progress

This extension essentially requires a transaction processing monitor (TP Monitor), which is a transactional RPC mechanism that is responsible for retrying RPCs that fail on a different node [GR97]. An IP-based implementation would be to mirror TCP state information so that a failed connection can be replayed to a new node, but this is extraordinarily difficult in practice and we know of no real systems that do this *[note to referees: Sun's full-moon clustering project may do this, but we have not been able to verify this yet*]. TP monitors are themselves non-trivial, since their components can also fail. It is much simpler to fall back to reload semantics, and leave this as an extension.

3) Basic Model updates are "at least once" semantics

This means that queries in the basic model have "at least once" semantics, which can be quite bad in the worst case. For example, if your connection dies in the middle of a credit card transaction, should you hit reload or not? If the transaction already committed (but you weren't told), you will buy the same item again.

Extension: Use transaction ids to detect already completed updates.

A useful extension to the basic model is to include transaction ids to detect repeat transactions. This is certainly possible today but rarely done in practice, so we leave it as an extension to the basic model. This extension requires that the commit is durable in the persistent store, so a transaction is the right mechanism. This can also solve the problem of users going to bookmarked pages that cause a transaction, but the transaction id needs be part of the URL.

To summarize the semantics of the basic model: the service provider focuses on locally high availability with independence for retried queries and at-least-once semantics. The extensions show how to improve the semantics, but they are typically expensive or difficult and therefore generally avoided in practice. The real value of the basic model is that allows us to understand what we mean by "availability", "fault tolerance", and "online evolution", and in general provides insight into to how to think about faults. A formal specification of the Basic Model is beyond the scope of this paper, but would be very useful as it would further drive the analysis of the tradeoffs.

Given this model, we revisit the key challenges of giant-scale services.

3 Load Management

We first look at the load management problem; it is the simplest and there are relatively recent products that provide robust solutions. The load manager has three responsibilities:

Provide the External Name: the external name can be a domain name or a set of IP addresses depending on the approach. The challenge is to make the external name highly available despite failure of some of the nodes and the corresponding loss of their internal names.

Load Balance the Traffic: This can be done with or without feedback from the nodes. The goals are higher overall utilization and better average response time.

Isolate Faults From Clients: This is the hard part, as it requires the manager to detect faults and dynamically change the routing of traffic to avoid down nodes. The reaction time is a key metric, as some clients lose service until the detection and failover occurs.

Before examining load management options, we first define two sets of metrics, one for load balancing and one for availability. We then use these metrics to evaluate load managers.

3.1 Basic Metrics

The most fundamental issue in load management is whether or not the manager understands the distribution of data across the nodes. There are three choices:

Symmetric: This is the constraint that all nodes are equal in functional capability (but perhaps not query capacity). Symmetric nodes greatly simplify load management because any query can go to any node.

Asymmetric: In this case, nodes vary in functionality and the load manager must correctly map each query to a node that can handle it. The common case is a partitioned database, in which the load manager must understand the partitioning to route the queries correctly.

Symmetric with Affinity: This is the symmetric case with an optimization for locality. Due to caching effects, it is very useful to try to partition the queries even in the symmetric case so that a given node tends to get the same queries repeated. Although the manager understands the partitioning, it is not required for correctness and any node can still handle any query.

All three styles are used in practice. There are four useful properties by which to think about these choices:

- *Database aggregation*, which is the property that adding nodes increases the database size rather than just adding replicas for throughput. Asymmetric systems always have this property and it is the main reason they are used.
- *Single-query throughput*, which refers to whether or not the throughput of a single query scales with the size of the cluster. This is normally associated with symmetric systems, since every node can serve the

target query. This is very important in practice due to short-term extremely popular content, such as the Starr Report, or the Star Wars movie trailer [Mov99]. In the asymmetric case, only a subset of the nodes (often only one) can handle the query.

- *Query locality*, which is property that a node tends to receive a subset of the queries and thus has a smaller working set and better cache performance. Asymmetric systems have this by definition, and the Symmetric with Affinity is designed for it explicitly. As shown by LARD [PAB+98], query locality greatly improves overall throughput by scaling the effective cache size with the number of nodes. In contrast, the vanilla Symmetric case spreads the queries evenly across the nodes, requiring each node to support the complete working set.
- *Even utilization*, which measures the effectiveness of the load balancing. Pure symmetric systems are much easier to maintain even load, while asymmetric systems are typically less effective since the manager cannot balance load across nodes with differing functionality. In general, closed-loop approaches that have information on the current load of a node perform well [Mit98], while open-loop approaches, such as fixed distribution or round-robin distribution have more variance in per-node utilization.

The simple approaches to load management are full replication and simple partitioning. Full replication provides symmetry, even load, and single-query throughput, but no aggregation. Locality-aware query mapping, as in LARD, can add query locality. Partitioning provides aggregation and query locality, but low single-query throughput and potentially uneven utilization.

Coupled-cluster databases, as shown in Figure 3, are symmetric but provide an aggregated database. Typically, all nodes can handle all queries, but they depend on data at other nodes to complete the query. In practice, the cluster backplane and inter-node caching enable high single-query throughput, even though the data is partitioned. Locality-aware query distribution can improve query locality here as well.

Although the coupled-cluster approach provides both symmetry and aggregation, it is much more complicated as nodes are no longer independent. To regain some independence, giant-scale systems are often built as replicas of relatively small coupled clusters. For example, one major ISP uses a single rack as the unit of replication: nodes within the rack form a coupled cluster, but the racks are independent replicas. Thus both the rack and the overall system provide symmetry, but aggregation is limited to the capacity of a single rack, and single-query throughput is limited by the number of racks (since only one node in each rack can handle the query).

Even utilization of replicas is simple, but even utilization within a cluster is harder and often depends on the database layout; randomized layout works well for even utilization, but can reduce locality without some locality-aware distribution. LARD was able to achieve both even utilization and query locality using adaptive partitioning [PAB+98].

In general, the relative requirements of database size versus throughput determine the clustering strategy. Typically the two are independent, so the architect can pick coupled-cluster size based on the database requirement (e.g., the database fits on six nodes), and then replicate that cluster to meet the overall throughput requirement. For example, for web caches, the working set size determines the cluster grouping, and the overall traffic determines the number of replicas.

Finally, if the whole database fits on one node, then the coupled-cluster case degenerates to the fully replicated model. Conversely, if the database is much larger than the number of nodes required by the traffic, it degenerates to the simple partitioning model. Both of the simple cases avoid the need for a backplane and provide independent nodes.

3.2 Availability Metrics

We now consider the availability metrics: failover response time, load-manager availability, and load shedding. *Failover response time* is the time it takes the load manager to detect and avoid a faulty node (or link to that node). During this window, some clients will get no queries through, effectively seeing a “down” service. Reducing this reaction time thus directly contributes to the effective uptime of the service.

Load-manager availability refers to the uptime of the load manager itself. A down load manager may directly result in a down service (if all traffic goes through it), or it may simply increase the failover response time and the variation in node utilization.

There is a fundamental tradeoff between failover response time and tolerance to the loss of the load manager. If load manager decisions are cacheable, then load-manager faults can be masked if they are short relative to the expiration time of the cached decisions. However, longer expiration times increase the failover response time, since load continues to a failed node until the decision expires. Invalidating cached entries that point to failed nodes helps, but typically each client has to do so independently and must wait for a relatively long timeout to discover the fault.[3]

A subtle issue in failover response time is determining what constitutes a fault. For example, it is possible for a site to be “up” but not functioning correctly. Thus the load manager ideally needs to know the health of the application rather than that of the node. There is at least one product,

3: Curiously, browsers do not seem to apply this optimization. When they detect a down server, they should reresolve the domain name and see if they get a different IP address. If so, they can retry the query with the new address. Currently, reloading the page just retries the same IP address.

WebSpective's *Traffic Management* [WS99], that provides application-level feedback to load managers, and there are several custom-built load managers that also use application-level feedback.

The third availability metric of a load manager is whether or not it does admission control or *load shedding*. When the system is saturated, it is very important to degrade gracefully. In general, the excess queries should be dropped as soon as possible, to minimize the resources they waste. Thus, the best place to shed load is in the load manager, *before* the traffic reaches the nodes. The load manager is typically in the best position to detect saturation as well. Unfortunately, so far only custom load managers support load shedding, but it is an important goal nonetheless. Done well, load-manager based admission control allows the system to maintain maximum throughput (albeit still insufficient) as the offered load well exceeds the saturation level. We look at graceful degradation further in Section 4.5.

3.3 Load Managers

There are three basic approaches to load management: DNS-based approaches, intelligent switch/router solutions, and the LARD front end mentioned above.

DNS approaches depend on the resolution of the domain name to IP addresses to handle both load balancing and availability. Load balancing decisions are thus made only at name resolution time, which means that balancing is rough at best. Worse, the failover requires updating the DNS mapping and reresolution by all clients, which make the failover response time poor, although it can improved by reducing the time-to-live value. This approach requires symmetry and there is no query locality, since the mapping is not query specific. Load manager availability is good because of primary/secondary DNS failover and DNS caching (at least with long TTL values).

More advanced DNS servers, such as Cisco's Distributed Director [Cis99], try to measure the health of IP addresses themselves using techniques such as ping messages. This allows them to increase the accuracy of the DNS map, but it does not reduce the failover response time, which is still tied to the TTL value. This is a severe restriction, and several users of this approach end up setting the TTL to zero, which means that *every* query has to do DNS resolution, which can be more expensive than the cost of the query itself.

A better solution is *virtual IP address failover*, which is a technique that can be used on a LAN. The idea is to have nodes on the same LAN watch each other; when a nodes goes down, other nodes take over its IP address(es). This makes the IP address persistent; the failover response time can be less than ten seconds. This technique is used in some cluster-based web caches [*] and in paired switches that use hot failover.[4]

The second general approach to load management is to use an inline intelligent switch (or router), generally referred to as "level 4+" (L4) switches, after the ISO layers above transport. The switch handles all incoming connections and directs them to the nodes, typically by rewriting TCP flows on the fly ("network address translation"). The switch thus forms the external name of the service, and it handles both load balancing and availability. Load balancing is typically done based on the number of outstanding TCP connections to each node. The switch may also partition the query space (typically URL space) across the nodes by using the query as part of the routing decision.

There are many such switches on the market and they have remarkable performance. The Alteon switch [Alt99], for example, can handle 200,000 simultaneous connections, while the Arrowpoint switch claims 5 Gb/s switching capacity while switching based on query content (not just TCP/IP information) [Arr99]. Both switches use custom ASICs for URL processing and both support hot-failover to a second switch for load-manager availability. The failover response times are quite low and are really only limited by the normal-case query response time, since it takes a while to decide that the server is down. In general, these switches are a robust solution, providing partitioning, load balancing and high availability. When used symmetrically, they provide single-query throughput, but no query locality, while when used asymmetrically, they provide query locality but poor single-query throughput.

The final load management approach is that of LARD, *locality-aware request distribution* [PAB+98]. The basic approach is load balancing across a symmetric cluster. The key extension is to target query locality by sending similar queries to the same node. They showed that they could achieve both good load balancing and good query locality simultaneously. A variant, *LARD with replication*, maps queries to a set of nodes, which allows single-query throughput to scale with the size of the cluster, as the target set can grow until all nodes handle the popular query. This use of replication also avoids hot spots. When used with coupled clusters, LARD would also allow database aggregation, although this has not been demonstrated to our knowledge.

Tables 2 and 3 summarize the relative merits of these three approaches. Most services use some form of L4 switch, as it is a simple and robust solution. So far, only custom solutions have attacked the problem of load shedding, but we expect that future switches will provide this as well. The DNS-based solutions are falling out of favor because of their poor failover response time and because of their inability to route traffic based on the query. The LARD approach works well for load balancing, but does not address the availability issues. We expect the LARD ideas to migrate into L4 solutions in the near future, further promoting them as the right solution.

4: In practice, each node has several IP addresses and those of the failed node are distributed across the remaining nodes, thus spreading the extra load more evenly. See the load redirection problem in Section 4.4.

Approach	Style	DA	SQT	QL	LB
DNS	Sym	N	Y	N	rough
L4 w/ part.	Asym	Y	N	Y	hot spots
LARD	SA	N	N	Y	hot spots
LARD with rep	SA	N	Y	Y	good
L4 + coupled clusters	Sym	Y	Y	N	good
L4 w/ part. + coupled clusters	SA	Y	N	Y	hot spots
LARD w/ rep + coupled clusters	SA	Y	Y	Y	good

Table 2: Approaches versus the Basic Metrics
Style = {Symmetric, Asymmetric, Symmetric with Affinity}, DA = Database Aggregation, SQT = Single Query Throughput, LB = effectiveness of load balancing.

Approach	Failover Response Time	Manager Availability	Load Shedding
DNS	very poor	good	none
DNS + VIP failover	good	good	none
Single Switch	good	poor	none
Paired Switches	good	excellent	none

Table 3: Summary of Availability Metrics

4 High Availability

High availability is one of the major driving forces of giant-scale system design. Other infrastructures—such as the telephone, rail, water and electricity systems—have extreme availability goals that should apply to IP-based infrastructure services as well. Most of these systems plan for failure of components and for natural disasters. However, information systems must also deal with constant rapid evolution in feature set (often at great risk) and rapid and somewhat unpredictable growth. British Telecom traditionally has used a 25-year planning horizon for the deployment of telephone infrastructure [?]; Internet companies (and analysts) have had trouble with even three-year roadmaps.

In this section, we develop basic ways to think about availability for giant-scale systems and cover some basic obstacles to high availability. In the next section, we focus on availability in the presence of rapid growth and change.

Component	Failure Rate
Disks	
Motherboards	
Cables	
Switches	

Table 4: Failure rates (after burn-in) of components

4.1 Some Basics

Although most of this section focuses on managing the impact of faults, it is worth reviewing some practical issues related to minimizing the likelihood of faults. These are basic techniques that are broadly useful for all highly available systems, and they form a kind of prerequisite before dealing with the management of faults.

Figure 4 gives some typical failure rates for various components. These are the steady-state failure rates under best-case conditions (covered later). Most of the failures actually occur either on arrival or in early use (the classic “bathtub” curve [cite]). Because of this most components, especially switches and disks, must be put through burn-in testing to detect these early failures and thus avoid them in live use. Thus these failure rates explicitly discount the initial failures.

The presence of people is the first source of failures. A prerequisite to low failure rates in general is the removal of people from the machine room [cite].

Cables are a huge source of failures, especially if they are moved at all. The main reason for using internal disks is not rack density, but rather cable elimination. Internal disks not only avoid the failures of cables, they also have much better impedance matching and less noise, thus better tolerating inconsistencies in disks and motherboards.

Temperature is also critical, especially for disks. An IBM rule of thumb is that failure rates of disks double with every 0-degree (Celsius) rise in temperature. This holds until the temperature is so low that the lubricants stiffen up.

Figure 4 shows a cluster designed for high availability. By design, there are no people, very few cables, almost no external disks, no monitors and extreme symmetry. All of these reduce the number of failures in practice. In addition, the cluster is actually remotely managed from offsite, and the temperature and power variations are limited contractually.

Given these basic failure reduction techniques, none of which are new, we move to systematic ways to manage the impact of faults on giant-scale systems.

Figure 4: 100-Node 200-CPU Cluster
Key points: no people, no monitors, no visible cables, extreme symmetry, internal disks.

4.2 Availability Metrics

The traditional metric for availability is *uptime*, which is simply the fraction of time that the site is up. Uptime is typically measured in *nines*: "4 9s" implies 0.9999 uptime, or 60 seconds of downtime per week (or less). Traditional infrastructure systems such as the phone system aim for 4 or 5 nines. Two related metrics are *mean-time between failure* (MTBF) and *mean-time-to-repair* (MTTR). In particular, it is useful to think of uptime as:

$$uptime = \frac{MTBF - MTTR}{MTBF} \quad (1)$$

Equivalently, *downtime = MTTR/MTBF*. The consequence of this equation is that we can improve uptime *either* by reducing the frequency of failures or reducing the time to fix them. Although the former is more pleasing aesthetically, the latter is much easier for systems under constant evolution. For example, to even *tell* if a component has a MTBF of one week requires well more than a week of testing under (heavy) realistic load; and if it fails, you have to start over, possibly repeating the process many times. Conversely, measuring the MTTR takes minutes or less and achieving a 10% improvement takes orders of magnitude less total time due to the very fast debugging cycle. Thus it is very useful for giant-scale systems to focus hard on MTTR and simply apply best effort to MTBF. We will see this fundamental tradeoff repeated in many forms.

We define *yield* as the fraction of queries that are completed:

$$yield = \frac{queries\ completed}{queries\ offered} \quad (2)$$

This is typically very close to uptime numerically (and also unitless), but it is more useful in practice because it directly maps to user experience and because it correctly reflects that not all seconds are of equal value. Being down for a second that had no queries has no impact on users or yield, but reduces uptime. Similarly, being down for one second at peak and off-peak times have the same uptime, but vastly different yields, since there is often more than a 4:1 ratio of peak to minimum traffic. Thus we will focus on yield rather than uptime.

Because these systems are typically based on queries, we can also measure the completeness of the queries, that is, how much of the database is reflected in the answer. We define this fraction as the *harvest* of the query

$$harvest = \frac{data\ available}{complete\ data} \quad (3)$$

A perfect system would have 100% yield and 100% harvest: every query would complete and would reflect the entire database.

The key insight is that we can affect whether faults impact yield or harvest (or both). For example, replicated systems tend to map faults to reduction in capacity (and thus yield at high utilizations), while partitioned systems tend to map faults to reduction in harvest, as parts of the database temporarily disappear, but the capacity in queries/sec remains the same.

4.3 The DQ Principle

The DQ Principle is simple:

Data per query * Queries/sec ~ constant

This is a principle rather than a literal truth, but it is a remarkably useful tool for thinking about giant-scale systems. The intuition behind this principle is that the overall capacity of the system tends to have a particular physical bottleneck, such as total I/O bandwidth or total seeks per second, that is tied to the movement of data. The DQ value is the total amount of data that has to be moved per second on average and it is thus bounded by the underlying physical limitation; at the high utilization typical of giant-scale systems, it approaches this limitation.

The DQ value is also measurable and tunable. Adding nodes or implementing software optimizations are useful exactly because they increase the DQ value, while faults reduce the DQ value. The absolute value of DQ is not that important typically, but the relative value under various changes provides a useful guide:

- The best possible result under multiple faults is a linear reduction in DQ.
- DQ often scales linearly with the number of nodes, which means that early tests on single nodes tend to have predictive power for overall cluster performance.
- All proposed hardware/software changes can be evaluated by their DQ impact.

- We can translate future traffic and feature predictions into future DQ requirements and thus into hardware and software targets.

There are two useful corollaries:

harvest * capacity ~ constant
harvest * yield ~ constant (at high utilization)

These follow from the DQ principle because the harvest is usually proportional to the average data per query, and capacity is just the total queries per second. When utilization is high, decreases in capacity cause decreases in yield, giving them a linear relationship.

For availability, the value of these principles comes in the analysis of the impact of faults. As stated above, the best we can do is a degradation in DQ that is linear with the number of (node) faults. The *goal* of a design for high availability is thus to control how DQ reductions affect our three availability metrics. (This assumes that we've already taken all of the basic steps above to minimize faults.)

4.4 Replication vs. Partitioning Revisited

Thus we return to the variations of replication and partitioning from the perspective of DQ and our availability metrics.

We start by considering a two-node cluster., which can be either two replicas or a partitioned database with two partitions. Traditionally, the replicated version is viewed as "better" because under a fault it maintains 100% harvest, while the partitioned version drops to 50% harvest. But the dual analysis is that the replicated version drops to 50% yield,[5] while the partitioned version remains at 100% yield. Even more effective is to realize that both versions have *the same initial DQ value and lose 50% of it* under one fault: replicas keep D the same and reduce Q (and thus yield), while partitions keep Q constant and reduce D (and thus harvest).

The traditional view of replication silently assumes that there is enough excess capacity to prevent faults from affecting yield. We refer to this as the *load redirection* problem: under faults the remaining replicas have to handle the queries formerly handled by the failed nodes. Under high utilization, this is unrealistic.

We can generalize this analysis to replica groups with *n* nodes:

Failures	Lost Capacity	Redirected Load	Overload Factor
1	$\frac{1}{n}$	$\frac{1}{n-1}$	$\frac{n}{n-1}$
k	$\frac{k}{n}$	$\frac{k}{n-k}$	$\frac{n}{n-k}$

Table 5: Impact of k losses in n-node replica groups

For example, a loss of 2 of 5 nodes in a replica group implies a redirected load of 2/3 extra load (two loads spread over three remaining nodes), and an overload factor for those nodes of 5/3 or 166% of normal load.

The key insight is that replication on disk is cheap but *accessing* that data requires DQ points; for true replication you need not only another copy of the data, but twice the DQ value. Conversely, there is no real savings to partitioning over replication. Although you need more copies of the data with replication, the real cost is in the DQ bottleneck not the storage space, and the DQ constant is independent of whether the database is replicated or partitioned. A consequence of this view is that *above some throughput, you should always use replicas*. In theory, you can spread the database more thinly as the capacity requirement increases and thus avoid the extra copies of replicas. But there is no DQ difference, so once the partitions are a convenient size, it makes more sense to replicate the data and enjoy more control over harvest and support for disaster recovery. It is also easier to grow systems via replication than by repartitioning onto more nodes; this idea is explored further in Section 5.1.

We can also vary the degree of replication based on the importance of the data, or more interestingly affect which data is lost in the presence of a fault. For example, for some extra disk space we can replicate key data in a partitioned system. Under normal use, one node handles the key data while the rest provide additional partitions. If that node fails, we can make one of the other nodes serve the key data. We still lose 1/n of the data, but it always one of the less important partitions. This intermediate version preserves the key data as in the normal replication case, but also allows us to use our "replicated" DQ capacity to serve other content during normal use.

Finally, we can exploit randomization to make our lost harvest a random subset of the data (and avoid hot spots in partitions). For example, many of the load balancing switches simply use a pseudo-random hash function to partition the data. In the presence of data of varying value, spreading the key data randomly makes our average- and worst-case losses the same: the value of the lost data is close to the average value of the data.

4.5 Graceful Degradation

It would be nice to believe that we could avoid saturation at a reasonable cost simply by good design. There are three major reasons that this is unrealistic:

- The peak to average ratio for giant-scale systems seems to be in the range of 1.6:1 to 6:1, which can make it expensive to build out capacity well above the peak.
- Single-event bursts, such as online tickets sales for Star Wars *Phantom Menace*, can be more than 10x

5: This is technically 50% capacity. Here we assume high utilization so that 50% capacity approximates 50% yield.

above the average. In fact, one such site, moviefone.com, actually added 10x capacity and still got overloaded. The event overloaded both their IP and telephone infrastructure, even overloading local phone circuits in some cities [Mov99].

- Some faults are not independent, such as key router failures or natural disasters. In these cases, DQ drops substantially and the remaining nodes become saturated.

Thus a critical part of delivering high availability is the design of mechanisms for graceful degradation under excess load. The DQ principle is again helpful: in the presence of excess capacity, we can either do admission control (ideally in the load manager as discussed above) to limit Q and thus maintain D, or we can reduce D and increase Q. The latter strategy has just started to be used in practice, but it makes a lot of sense: if we can reduce D dynamically then we can increase Q (capacity) and thus maintain yield at the expense of harvest. For example, we would expect cutting the effective database size in half to roughly double our capacity. This gives us new options for graceful degradation: we can focus on harvest with admission control or we can focus on yield with dynamic database reduction, or we can use a combination depending on the type of query. The larger insight is that graceful degradation is simply the explicit management how saturation reduces our availability metrics.

Here are some more sophisticated examples:

- If we have an estimate of query cost (measured in DQ!), which we do for search engines, then we can do more aggressive admission control *based on cost*. This reduces the *average* data required per query, D, and thus increases Q. Note that our admission control policy is affecting *both* D and Q: denying one expensive query may enable several inexpensive queries, giving us a net gain in harvest and yield. Admission control should be done probabilistically so that reloading hard queries eventually works.
- Under saturation of a financial site, we can make stock quote queries cacheable, which will make them stale but nonetheless reduces the offered load and thus increases yield at the expense of harvest (the cached queries don't reflect the current database).

To summarize, we can use the DQ principle as a tool for designing how saturation affects our availability metrics. First we decide which metrics to preserve (or at least focus on), and then we use sophisticated admission control to affect Q and the possibly reduce the average D, and we use aggressive caching and database reduction to reduce D and thus increase Q. In the next section, we see how this helps with disaster tolerance.

4.6 Disaster Tolerance

In the two previous sections, we saw how to think about replica groups and how to think about graceful degradation. Disaster tolerance is mostly a combination of these two.

We define a disaster as the complete loss of one or more replicas. For natural disasters, we expect to lose all the replicas at one physical location, while other disasters, such as fires or A/C failures may affect only one replica at a location. Under this model, the basic question is how many sites to have and how many replicas per site.

Given n replicas, we can use Table 5 to understand the load redirection problem when we lose k of these replicas. For example, with two replicas at each of three locations, we would expect to lose 2/6 replicas during a natural disaster, which implies that the remaining replicas must handle 50% more traffic. This will almost certainly saturate the site, which then have to recover from using our techniques for graceful degradation. For example, one plan would be to dynamically reduce D by 2/3 (to get 3/2 Q) on the remaining replicas. Another plan would be to reduce D to a specific 50% for any disaster, which is simpler but not as aggressive.

A harder problem for disaster recovery is surviving the loss of the load manager. If it is an inline approach, such as the L4 switches or LARD, the external name becomes unavailable and you have resort to DNS changes to redirect traffic to the other replicas, which as discussed above has a very slow failover response time (hours). With a smart client approach, one of our extensions to the basic mode, the clients can perform the higher-level redirection automatically and immediately [YCE+97].

4.7 Conclusions

In this section, we defined several useful availability metrics and the DQ principle. We then used these ideas to analyze high availability, graceful degradation and disaster tolerance. In all cases, the new tools lead to insights and powerful ways of thinking about the issues. In the next section, we apply these ideas to online evolution and growth.

5 Online Evolution & Growth

High availability is always a difficult challenge, and one of the traditional tenets of highly available systems is to aim for minimal change. This is in direct conflict with both the growth rates of these services and "Internet time" — the practice of extremely fast service release cycles. For giant-scale services, we have to plan for continuous growth and frequent updates in functionality. Worse still, the frequent updates mean that in practice the software is never perfect and that hard-to-resolve issues such as slow memory leaks and non-deterministic bugs tend to remain unfixed.

Thus the task at hand is to maintain high availability in the presence of expansion and frequent software changes. The philosophy is to make the overall system tolerant of individual node failures, but to try to avoid cascading fail-

Figure 5: Growth in unique visitors for major giant-scale services. Based on company and Media Metrix data [MM99].

ures. Thus "acceptable" quality software comes down to a target MTBF and the absence of cascading failures.

We first look at growth rates and how to think about capacity planning and then we examine online evolution, looking at several ways to upgrade a service with minimal impact on availability.

5.1 Growth and Capacity Planning

The remarkable growth of existing giant-scale services is shown in Figure 5. This is conservative in that it only measures the growth in unique visitors, and ignores increases in visits/user, work per query, and bandwidth per query, all of which have gone up over the past several years. Smaller sites, such as Snap! and Goto.com (not shown), have even higher growth rates. The growths are somewhat uneven, which complicates capacity planning, as you must plan for higher growth than you will probably achieve; for example, several of the quarters shown have 30-50% growth rates.

Graph interpolation works well for predicting the number of future queries, but it does not help with the impact of feature changes. For that we can use relative DQ values. For example, if a new feature requires 20% more DQ capacity per query, which can be measured on a small cluster using log playback, then we need an 20% expansion in addition to the organic growth in queries. Capacity planning can thus be viewed as the product of two factors: query count growth and DQ growth.

Once you have a target DQ value, it is a straightforward but complex task to achieve it. The hard part is understanding the lead times for all aspects of expansion, and then maintaining enough excess capacity to cover the lead times ahead of the growth curve. Excess capacity targets ("headroom") seem to range from 15-30%, which translates to about 45-90 days.

Table 6 shows some typical lead times. The key long-lead item is to make sure that you have enough data-center quality rack space. AOL is currently spending $520M for a third data center to build out rack space ahead of service growth [AOL99]. Once rack space is lined up, the second challenge is to manage hardware lead times and inventory. Most giant-scale services have to keep some inventory of all components, and they may keep large inventories of long-lead or variable-lead items. In some cases, because their size, they can get vendors to maintain the inventory for them, which is an advantage financially.

Item	Lead Time
New cluster from scratch (including location)	120-150 days
New rack space — existing location, but added power, A/C, bandwidth	90 days
New rack space only	60 days
New SMPs (ordered)	30-60 days
New PCs (ordered)	2 days
New nodes, in stock, on existing rack space	10 nodes/day/ person

Table 6: Typical leads times for new capacity

Given that rack space is the critical aspect of growth, it makes sense to expand clusters in larger chunks, at least whole racks if not whole rooms. This also amortizes many of the administrative tasks over a larger number of nodes.

The second challenge with small incremental steps is that they often require repartitioning the database, which may be expensive. Growing in fewer, larger steps, reduces the repartitioning overhead.

Because of the rack space and repartitioning issues, one major service now adds whole 100-node clusters rather than adding nodes to existing clusters. The new cluster is a replica of existing large clusters. This approach has the advantage of simplicity: each new cluster is a "cookie cutter" operation, thus reducing the logistics issues and avoiding repartitioning altogether.

The lesson from all this is that the long-lead times make capacity build-out a critical and complex task. Incremental scalability in practice is thus quite challenging, and giant-scale services tend to focus on big steps at a reduced frequency.

5.2 Online Evolution

As with other high availability systems, we can think of maintenance and upgrades as a form of controlled failure. In particular, we view online evolution as a temporary controlled reduction in DQ value, and then minimize the impact of reduced DQ on our availability metrics. In general, online evolution requires a fixed amount of time per node, u, so that the total DQ loss for n nodes is:

$$\Delta DQ = n \cdot u \cdot DQ/node = DQ \cdot u \quad (4)$$

That is, the lost DQ (measured in DQ units * seconds) amount is simply the total DQ value times the upgrade time per node.

Assuming working space exists on the nodes for both versions, the upgrade time should match the MTTR, since everything can be in place before the downtime starts. Without sufficient workspace, the downtime is much greater, since the new version must be moved onto the node while it is down.

We next cover three approaches to online evolution that differ in their handling of this DQ loss and in how they deal with working space and backward compatibility.

Fast Reboot: The is the simplest version: simply quickly reboot all nodes into the new version. The guarantees some downtime, but is very simple. Because this is a controlled downtime, it is better to measure lost yield rather than downtime: by upgrading during off-peak hours we can reduce the yield impact for the same amount of downtime.

Rolling Upgrade: In this approach, we upgrades nodes one at a time in a "wave" that rolls through the cluster. This has the advantage that only one node is down at a time, thus minimizing the overall impact. In a partitioned system, we will have harvest reduction during the *n* upgrade windows, but in a replicated system (in off hours) we expect 100% yield and 100% harvest, since we can update one replica at a time and we probably have enough capacity in off hours to prevent lost yield. One disadvantage with this approach is that it requires the new version be backward compatible, since the two versions will coexist. The other two approaches do not have this restriction, because they only run one version at a time.

The Big Flip: The final approach is more complicated. The basic idea is to update the cluster one half at a time. In particular, we take down half the nodes and upgrade them. Then, during the "flip", we switch all of the traffic to the upgraded half, which can be done atomically with an inline load manager. We then upgrade the second half and then bring those nodes back into the live cluster. As with fast reboot, only one version runs at time. The big flip is quite powerful and can be used for all kinds of upgrades: hardware, OS, database schema, networking and even physical relocation. When used for physical relocation of a cluster, the big flip must typically use DNS changes to do the flip, and the upgrade time includes physically moving the nodes, which means the window of 1/2 DQ performance lasts for at least several hours. However, this is manageable over a weekend and has been done at least twice. Note that the 50% DQ loss can be translated into either 50% capacity for replicas (which might be 100% yield on weekends) or 50% harvest.for partitions.

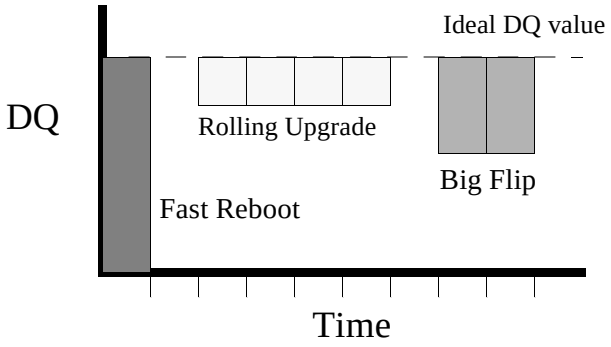

Figure 6: The shaded regions show how the three approaches map their DQ loss (down from the ideal value) over time. The area of the three regions is the same.

Because all three approaches have the same DQ loss (for a given upgrade), we can plot their effective DQ level versus time, which is shown in Figure 6. The area of the three curves is the same, the difference is in how the DQ loss is spread over time.

All three approaches are used in practice, with rolling upgrades probably the most popular. The heavyweight big flip is reserved for more complex changes and is used only rarely. All three benefit from the DQ analysis and from explicit management of the upgrade's impact on the availability metrics.

6 Conclusions

In this paper we have defined several tools for the design and analysis of giant-scale clusters. Starting with the basic model, we developed the fault model and semantics of these services used in practice and suggest several possible extensions.

Load management is a complex task with many goals: load balancing, manager availability, fast failover response time for nodes, single-query throughput, database aggregation, and load shedding. We covered three general approaches and compared them via our performance and availability metrics.

We also developed novel ways to think about high availability, including the DQ principle and the use of harvest and yield, in addition to uptime, to more precisely capture the impact of faults. We covered several sophisticated ways to control the impact of faults through combinations of replication and partitioning. Finally, we used these tools to analyze graceful degradation and disaster tolerance.

In the final part, we applied these ideas to online evolution and growth, looking at both capacity planning and approaches to service upgrades that minimize the impact of the upgrade on availability.

We have found these techniques to be very useful in practice. They seem to get to the core issues that matter in practice and provides ways to think about availability and

fault tolerance that are predictable, analyzable, and measurable in practice.

References

Note: some references remove for author anonymity

[Alt99] Alteon Corporation. *ACEdirector Data Sheet*. http://www.alteon.com/products/acedirector-2.html.

[AOL99] America Online. "Governor Gilmore and America Online Announce Selection Of Prince William County As Site For $520 Million Tech Center." Press Release, March 10, 1999.

[Arr99] Arrowpoint Communications. *CS-100 and CS-800 Data Sheets*. http://www.arrowpoint.com/products/data_sheets.html

[Cis99] Cisco Systems. *Distributed Director Overview*. http://www.cisco.com/warp/public/cc/cisco/mkt/scale/dist/prodlit/dd_ds.html.

[GR97] J. Gray and A. Reuter. *Transaction Processing*. Morgan-Kaufman, 1997.

[Mit98] M. Mitzenmacher. *How Useful is Old Information?* Digital SRC Technical Note 1998-002. February 8, 1998

[MM99] Media Metrix. http://www.mediametrix.com

[Mov99] MovieFone Corporation. "MovieFone Announces Preliminary Results From First Day of Star Wars Advance Ticket Sales." Company Press Release, May 12, 1999. http://biz.yahoo.com/bw/990512/ny_moviefo_1.html

[PAB+98] V. S. Pai, M. Aron, G. Banga, M, Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. "Locality-Aware Request Distribution in Cluster-based Network Servers." *Proceedings of ASPLOS '*98. San Jose, CA, Oct. 1998.

[WS99] WebSpective. *Traffic Management, WebSpective Data Sheet*. http://www.webspective.com.

[YDE+97] C. Yoshikawa, B. Chun, P. Eastham, A. Vahdat, T. Anderson, and D. Culler. Using Smart Clients to Build Scalable Services. *Proceedings of the USENIX 1997 Annual Technical Conference*, January 1997.