Parallel Computing for the Internet

Eric A. Brewer
Inktomi Corporation

1 Motivation

The rise of the Internet has brought with it the need for services that can support hundreds of millions of users, 24 hours a day, with rapid evolution and growth. The sheer scale of these services demands parallel computing on a grand scale. More importantly, this new class of applications requires the solution of many difficult problems that are new to the parallel computing arena, including high availability, graceful degradation, and online evolution and growth of the service.

Given the scale requirements, these 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; AOL's new US$520M data center will be larger than three football fields and filled almost entirely with clusters [cite]. There are four underlying reason for the use of clusters:

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 but providing largely independent faults. Much of this paper focuses on how to leverage this independence in to 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.

Service	Nodes	Queries	Node HW
AOL web cache	>300	>3.3B/day	4-CPU DEC 4100s
Inktomi Search Engine	500	>40M/day	2-CPU Sun Workstations
Geocities	>300	>25M/day	PCs
Anonymous web-based e-mail	>400	50M/day	PCs

Table 1: Example Clusters for Giant-Scale Services

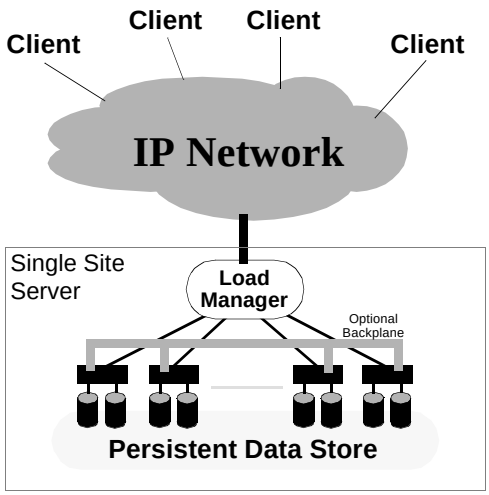

Figure 1: The Basic Internet Model

roadmap

2 The Basic Internet Model

Our basic model for infrastructure services in shown in Figure 1. The goal of the model is 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.

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. 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.

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 servers. This is required to preserve the availability of the external name in the presence of server faults. Second, the load managers balances load among the (up) servers.

Servers: The servers are the workers of the systems, combining CPU, memory, and disks into an easy-to-replicate unit. The server is the unit of expansion and often the *field replacable unit*, which means that if anything in the server breaks (including disks), you replace the whole server and deal with the subcomponents off-line.

Persistent Data Store: This is a replicated or partitioned "database" that is spread across the disks of the servers. It might also include network-attached storage such as RAID systems.

Backplane: The services with non-trivial parallelism use a system-area-network among the servers, which functions like a backplane in a multiprocessor. The network handles inter-server traffic such as redirection to the correct server or coherence traffic for the persistent data store.

Auxiliary Systems: 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.

2.1 Two Examples

Figures 2 and 3 show two illustrative systems at opposite ends of the complexity spectrum: a simple web farm and complex server similar to the Inktomi search engine cluster. They differ in their load management, their use of a backplane, and their persistent data store. Nearly every web-based service fits this model.

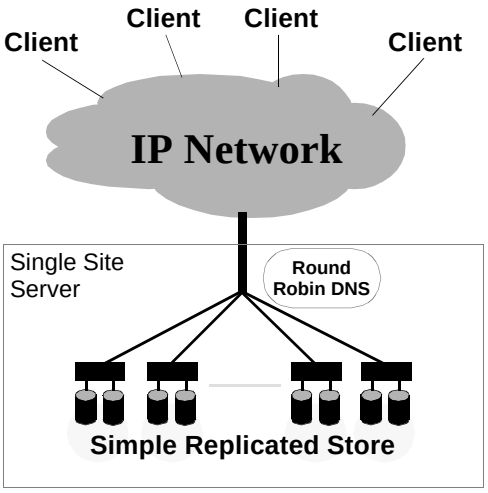

Figure 2: A Simple Web Farm

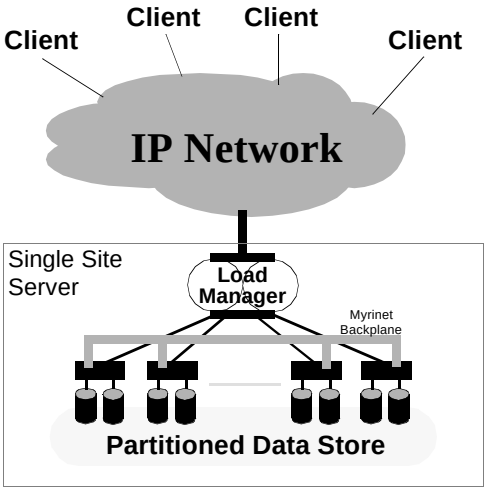

Figure 3: A More Complex Server

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. 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 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 server names. They can balance load based on outstanding connections and can quickly respond to failed nodes by avoiding them for new connections.

The persistent store is partitioned across the servers, possibly without any replication. This means that nodes 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.

In the case of a typical Inktomi search engine cluster (their are currently five worldwide), the backplane is Myrinet (1.6 Gb/s) and it connects 100 nodes, each with 2 CPUs and 4 disks. The store is fully partitioned with some replication for key data; different nodes may get different amounts of data according to their relative capabilities (nodes may 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 the answers to whole queries.

2.2 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. This is also the primary area where Internet applications differ from traditional parallel computing. 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.[1] 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: Make the 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 load balancing). Work on "smart clients" [cite] shows how to do this for applets and client-side plug-ins, in which the service can control some of the code on the client.

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

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. 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 explore some of the key challenges of giant-scale services: high availability, graceful degradation and online evolution.

3 High Availability

High availability is one of the major driving forces of giant-scale system design. Other infrastructures—such as the telephone, highway, rail, water and electricity systems—

1: 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.

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

have extreme availability goals that should apply to IP-based infrastructure services as well. Most of the systems try to 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 unpredictable growth. British Telecom traditionally has used a 25-year planning horizon for the deployment of telephone infrastructure [cite]; Internet companies (and analysts) have had trouble with even two-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.

Figure x shows one of the Inktomi search engine clusters. Some of basics of high availability are visible in this photo: there are no people, monitors or visible cables. People are the primary cause of outages (often by breaking/moving cables), so it is wise to simply keep them out. The other principle visible in this picture is extreme symmetry: the only way to keep these giants systems simple enough to manage is to make them extremely regular and consistent. Any variation in software, hardware or rack space is suspect.

3.1 Availability Metrics

The traditional metric for availability is *uptime*, which simply the fraction of the 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 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 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.

3.2 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 sys-

tems. 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 is very predictive:

- 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 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.)

3.3 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 with a two-node cluster, shown in fig x. 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,[2] 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.

Finally, it is important to realize 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.

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	$\frac{k}{n}$	$\frac{k}{n-k}$	$\frac{n}{n-k}$

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.

We can also vary the 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 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 rather than sit idle.

Finally, we can exploit randomization to make our lost harvest a random subset of the data, (as well as to 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 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:

2: This is technically 50% capacity. Here we assume high utilization so that 50% capacity implies 50% yield. We will make this translation often.

- 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 above the average. In fact, one such site, moviephone.com, actually added 10x capacity and still got overloaded.[cite]
- 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.

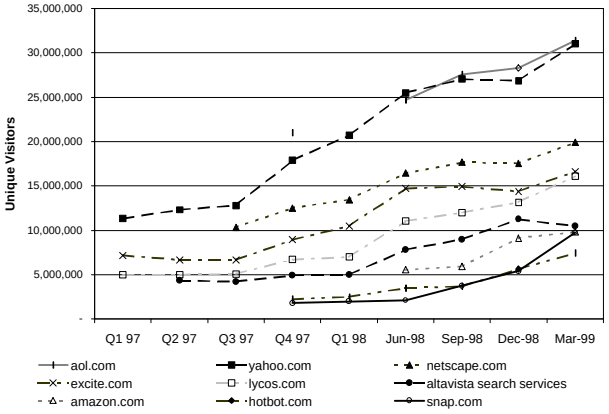

Figure 5: Growth in unique visitors for major giant-scale services. Based on company and Media Metrix data [cite].

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 product 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 task at hand is 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 failures. 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.

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 2: Typical leads times for new capacity

Once you have a target growth rate, 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 2 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. Once rack space is lined up, the second challenge is 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.

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, Inktomi 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 focus on big steps at a reduced frequency.

5.2 Online Evolution

6 Conclusions

7 References

[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.

aol98 online time