Functionality, Availability, Agility, Manageability, Scalability -- The new priorities of application design[1]

Jim Gray
Microsoft Research
15 April 2001

0. Introduction

Traditionally, enterprise systems have worried a great deal about scalability**,** availability, and manageability. There have been heated debates and competition in the scalability arena, with proponents of ScaleUp and ScaleOut each making cogent arguments. Paradoxically, scalability is the least of our problems today. Today, I and many others sort the requirements in the following order:

1. Functionality: The system does what is required.
2. Availability: The system is always up.
3. Agility: The system is easy to evolve as the business changes.
4. Manageability: The operations cost is modest.
5. Scalability. The system can grow without limits as demand increases.

This is a controversial statement, but I will try to convince you that we have "solved" the scalability problem. The other problems are just as challenging as they ever were. Indeed protecting against sabotage is a more difficult task than it was in pre-Internet days. There is also the old argument that "I can make the system arbitrarily fast if it doesn't have to produce useful and correct results" – this also scalability which is just one aspect of performance.

1. Functionality

Who cares if your application is up all the time? Only the people who want to use it. If you have no users, then none of the other issues matter. So the first requirement for any system is that it has useful applications. As a low-level systems guy I do not have much to say about these high-level apps. The best I can do is make it easy for the folks at SAP, Great Plains, AOL, Yahoo!, eBay, Fidelity, USGS, Amazon, MSN, … to build great applications, and to provide a substructure (see items 2-5 above) that makes application developer's lives easy – or at least tolerable.

2. Availability

System availability is the fraction of tasks that are preformed within the designated response time. The easy way to measure it is to count the number of 9's. Class 2 is 99%, class 5 is 99.999% and so on.

Something paradoxical happened in the last few years – availability and availability expectations have declined. It used to be that telephones delivered class-5 availability, and commercial computers delivered class-4 availability. When a bank's ATM network went out, it was front-page news. But, now we have cell phones that fade in and out and web sites are often down or unresponsive. People have come to expect Class-2 availability. Indeed, one eBusiness bragged in their quarterly report that they had finally achieved 99% availability in the most recent quarter – that's about 100 minutes of downtime per week.

Systems fail for prosaic reasons: hardware, software, operations mistakes, and environmental problems (power failures, storms,..). Recently, sabotage has become a more pressing issue with hacker attacks on systems.

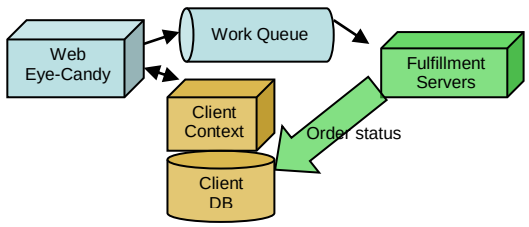
One can easily mask most hardware, software, and environmental errors with redundancy. The first step is to make the storage reliable by using mirroring or RAID5 to store the data on at least two disks. This

[1] Astute readers comment that I have left out Security, Privacy, Predictability, Performance, and Confidence. Sigh! The, list is long!

protects the data from any single disk failure. The next step is to have redundant servers and a load balancing system that can redistribute the load as servers fail. Persistent data is stored in a database *partitioned* among several servers, and these servers are *packed*, so that if one server fails, another member of the pack can provide access to the failed server's database. Indeed, the typical large web site has hundreds or thousands of anonymous *clone* front-end web servers. The incoming requests are spread among these clones. Some connections are sticky (SSL connections, web page connection objects) but stateless middle-tier servers to give a more scaleable and fault-tolerant design. All state is stored in a packed and partitioned back-end database that supports the failover mentioned earlier.

This design pattern of cloned front-ends, and packed-and-partitioned backend database servers is becoming extremely common [Devlin]. A slight refinement seems to add an order of magnitude improvement to perceived availability from class-3 to class-4. Request fulfillment often requires accessing other web services or databases (a travel site might need to access airline, auto, and hotel servers, a bookseller might need to access the various publisher warehouses). The client database has a cache of all the data needed to take the client orders and is completely decoupled from the backend server that deals with other less-reliable services. Email servers are a classic example of this design, but this design pattern is increasingly popular as workflow systems like BizTalk™ and XLANG become more mature.

Figure 1: A service where clients submit requests that are fulfilled asynchronously by a backend workflow business process engine – classic queued transaction processing. In this system, the service is always available even if the backend server is unable to contact the related services. Of course the front-ends are cloned and the backend database servers are packed and partitioned.



Sites must be geo-plexed to achieve more than four 9s of availability – that is the data is stored in two different geographic locations, and clients are routed to be fallback site if the primary site is down. Ideally, the two sites are on independent power grids, in different weather cells, are on different earthquake faults, and are independent from one another in every way possible. But, they have to have exactly the same data (or nearly so). Ten or twenty years ago, this was a very sophisticated feature. Visa, for example, custom-built this architecture and it is essential to their high-availability. But Visa implemented this all from bare metal. Now most modern database systems offer database replication – and the feature is widely used to geoplex data and to provide fallback service.

Partitions, clones, and GeoPlexs aid in one of the thorniest operations problems: incremental test and deployment of new functionality. The operations staff can do rolling upgrades of most application changes by first installing them on a few nodes and closely monitoring the behavior. Should something go wrong it will impact a small part of the user community – often a part that has signed up for this dog-food service. If that part of the deployment goes well, the changes can be gradually rolled out over the space of days or weeks, with the constant option of falling back on the old system if the changes create problems.

Clones, packs, load balancing, failover, and fallback all improve availability – and they are all automatic. So, why has availability gotten worse? To be honest, I do not know – but here is what I conjecture. Some of these might be due to novel technology, e.g., wireless transmission in the case of cell phones and wireless data services. But, even classical applications have gotten worse. Taking Microsoft as an example, the most spectacular MSN outages of late have been due to operations mistakes (misconfigured a router) and sabotage (denial of service attacks). A geoplex might have helped the first problem, and indeed MSN has now contracted with a third party to geoplex the DNS service. But traditional techniques do not really address the sabotage problem. We have been operating the TerraServer (www.terraserver.microsoft.com) for 3 years now. It is now a cloned front-end with a 4-way 3-active, 1-standby server pack. The SQL backend has delivered 5-9s of availability over the last year -- even though one node had some serious hardware problems. But, due to some operations and sabotage problems, the end-user has seen more like Class-2 service.

To reiterate, modern technologies mask the common faults we designed for a decade or more ago – we have won the war we fought a decade ago. Now operations mistakes and sabotage are the main sources of failure. So, now we have a new war to fight. Operations tasks have to be automated and eliminated – and systems should just be self-protecting and self-healing, both at the hardware level (IBM's recent announcements) and at the software level (we all have just begun …).

Dealing with sabotage is problematic. In the past, sabotage was a bomb blast or a disgruntled employee. Now it might be a hostile government or an offshore criminal organization or simply a "script kiddie". Denial of Service attacks, virus attacks, spoofing attacks each pose completely different threats. Internal threats have always been problematic (disgruntled or dyslexic systems administrators.) But, today there are many people inside the firewall who can do harm to your system.

The problem looks incredibly complex and there seems to be no general solution: firewalls here, mail filters there, intrusion detection over there, and so on. Clearly, these problems have magnified in the recent past and it seems likely that they will increase. It is probably the case that we need to go to a much more secure infrastructure: ipSec everywhere, no anonymous access, careful tracing of all traffic, and other draconian measures to make it much more difficult to launch these attacks.

3. Agility

So, you finally get your system working and in fact it is delivering class-5 availability. The business is prospering. Things are going so well that the company decides to (1) do a leveraged buyout of a much larger company and asks you to integrate your site with theirs, (2) form a consortium with two other companies to provide some new services, (3) internationalize the operation offering services in the 20 major languages and 50 major currencies and 500 tax codes, and (4) embrace the latest technology fad, which today is UDDI, WSDL, schemas, XLANG, SOAP, and XML. Of course, they tell you that this last move will make jobs 1, 2, and 3 easy.

This story sounds bizarre, but it is being acted out in thousands of companies right now this minute. Anyone who has a successful application has to extend it to add new services, and to support new technologies.

I confess to having drunk the .NET cool-aid and so agree that UDDI, WSDL, schemas, XLANG, SOAP, and XML help make applications more agile. This is object-oriented techniques of encapsulation and polymorphism applied on the scale of web services. WSDL and SOAP insulate clients from the implementation details of various web services. At a higher level XLANG and the tools for it allow application designers to express workflows and the ways they interact. If you are old enough to remember what Formats and Protocols means (FAP), schemas, WSDL, and SOAP are formats, XLANG is protocols.

The premise is that XML Web Services will be the basis for both adding new services to an application, and the basis for integrating applications. Fortunately we have a hundredfold improvement in processing power and bandwidth that we can squander it on this new abstraction layer – thank you Moore and Gilder.

Pat Helland has been evangelizing the model of *emissaries* and *fiefdoms* as a way of structuring web services both internally and externally. Fiefdoms are stateful services with clear business rules. Emissaries are stateless or stale-state services that gather information and ultimately present a proposal to one or more fiefdoms. Pat has some design patterns that are standard emissary-fiefdom and fiefdom - fiefdom interactions. One is a *monologue*: where the fiefdom sends out a sequential stream of information, and another is a *dialog* which is an XLANG style interaction between a fiefdom and another agent. Pat's ideas have growing currency within Microsoft at least. I believe that these ideas can help designers build more agile XML Web Services and applications.

4. Manageability

Operations costs have always been a major part of the expense of running a computer system. Some impressive systems run by tiny staffs, but the norm is that operations and management consumes huge budgets. Well-run and large-scale shops typically spend several thousand dollars per server per year in operations cost. Typical shops spent ten times that. As the price of hardware, software, facilities, and communications trends towards zero, the management-people costs dominant.

The availability discussion pointed out that many of the really big outages are caused by operations mistakes -- typically by mis-configuring the system, or not following good practices (e.g. not keeping a backup copy of the data somewhere safe).

The solution to both these problems, cost and reliability, is to reduce or eliminate operations tasks. Computers need to be more introspective, more self-tuning, and self-healing. Again there has been impressive progress in this area.

The first really successful version of this was in VMS. That operating system tuned itself as it observed the system load. More recently most database systems have become increasingly introspective. My personal experience is with Microsoft's SQL Server which I never tune. My only intervention is to add indices and occasionally give the query optimizer a hint. But, even that is being automated by the Index-tuning wizard that recommends physical database design and index designs [Chaudhuri]. Similarly, the Windows Update Service [WUS] automatically manages system change-control: the system can be configured to check for updates and download them. Browsers are perhaps the most sophisticated in this regard; they download applications and updates as needed and cache them on the client. Only the outermost container (which is a very thin shim) is immutable.

There is huge progress in this management space, but we still have a long way to go. Right now configuring security, configuring networking, configuring applications, configuring storage, and configuring middleware is each a separate task and skill set. My sense is that is easy to make things work if everything goes well. But, when things go wrong, it is VERY difficult to diagnose the problem or deduce the solution. There is also an aspect of agility here – re-configuring is very hard, moving from one configuration to another without forgetting some detail is very hard, knowing with confidence that all is taken care of is very hard or even impossibly hard.

6. Scalability

Fifteen years ago, I and many of my colleagues set out to achieve a thousand transactions per second – sixty thousand a minute, nearly a hundred million a day. At the time this was a huge number – it was more than *all* the banking transactions in the United States. In 1994, Oracle running on Vax/VMS broke the 1,000 tps barrier followed by an IBM/TPF benchmark and an RDB/VMS benchmark in the 3,000 tps range. These were huge numbers at the time involving computers costing thirty million dollars.

Soon thereafter the standard metric for transactions (the tpcA benchmark) was replaced with a six-times more demanding benchmark (tpcC). To keep the numbers rising, the metric was transactions per minute rather than per second ☺ -- an immediate 10x inflator in the numbers. In 1996 I helped with a 12,000 tps benchmark (loosely based on tpcA), which showed a billion banking transactions per day. This was more transactions than all the banks, all the airlines, and all the hotels in the world. One colleague asked me how many of these systems I expected to sell – I had to answer zero because the system really was larger than anything anyone needed at the time. But, then along came the Internet – and now AOL, MSN, Hotmail, Yahoo!, eBay, Google, and others have billions of page views a day. Depending how these page views and their ad impressions are logged and billed, there are billions of transactions a day that need to be recorded.

At the time, people observed that the billion-transactions-per-day system was not manageable: it was an array of 45 nodes with a manually partitioned database and with virtually no tools to manage the application or the system. That was true at the time, but we have been working over the last 5 years to build the requisite tools, and now the situation is much improved.

Today, we are at 600,000 tpmC (about a billion tpcC transactions per day), and the cost per transaction is hovering in the range of 5 microdollars – the hardware and software cost of the system equates to about five dollars per million transactions. This is about as close to free as you can get, and explains in large part why people can afford to operate web sites that get a penny or less in advertising revenue for each transaction. A penny is a LOT of microdollars.

The tpcC and the more recent tpcW benchmarks may be faulted for being stunts: they represent what a wizard can do. Still, they show a 3-tier architecture (1) fat client, (2) TP-monitor or web server, (3)

backend database system. They show the relative cost of each component, and they show the tradeoffs between ScaleUp and ScaleOut.

Focusing on the tpcC results, there are three styles. The **ScaleDown** style that configures systems that would be typical of what a real customer might do. Dell and Compaq for example both have low-end systems running Windows 2000 and SQL Server 2000 that cost about 100k$, have about 1TB of disk and can process about a million transactions per day. These systems are the workhorses of the Internet. They can handle all but the top 1,000 websites, and are adequate for about 95% of SAP installations.

There is a class of single-node **ScaleUp** systems from HP, IBM, and Sun, running DB2, Oracle, and Sybase that process about 300 million tpcC transactions per day on a computer costing about ten million dollars with a 20 TB database. These giant systems can handle the backend of all but the top 25 websites – and for those a ScaleOut of a ten or 100 of these systems would be adequate.

And then there is a class of multi-node **ScaleOut** clusters from IBM and Compaq running Windows2000 and SQL Server producing about a billion tpcC transactions per day on a 42 TB database. These ScaleOut systems are impressive. The Compaq system of 24 8-processor nodes costs about ten million dollars. These systems can easily handle any deployed workload of any corporation or government application.

The first thing to appreciate about all these systems is that they are beyond what anyone really needs – well not quite. In fact, we see companies buying multiple HP, or IBM, or Sun, or Compaq or Dell systems in this class.

What this *really* means is that you cannot just have a ScaleUp story – virtually everyone needs a ScaleOut story for packs of partitions (failover) and for GeoPlexs, and just to carry the load of an expanding application base. The only question is: "What size is your building block?"

I and others have long observed that big blocks cost big bucks. Big blocks have low volume and so high fixed engineering costs that have to be spread over very few units. The tpcC results show this fairly clearly, when you subtract out the disks from the equation (disk prices are nearly constant across all vendors) the residue systems vary in cost by a factor of ten or more. If high-volume Intel platforms price performance is normalized to 1, then the minis cost 10$ per unit and the mainframes cost 100$ per unit. So, I have long advocated ScaleOut clusters built with commodity bricks. Uniprocessor or at most four-processor nodes are fairly high volume units today – beyond that you pay a real premium.

Meanwhile there has been a real push for server consolidation. The chairman of IBM is calling for the return of the mainframe, asserting it is cheaper to run 1,000 Linux systems on a mainframe than it is to run 1,000 Intel boxes – mostly because the management costs dominate. Given current tools and prices, it is clear that one 4-way processor is cheaper to manage than four one-way processors. But, this need not be the case. Indeed, I am impressed that Google runs an 8,000 node Linux cluster, 5 data centers, an extensive network, and a rapidly evolving application all with a staff of 12. It would take LOT of IBM mainframes to match the power (3 teraops) and capacity (1.2 petabytes) of that server. But the battleground remains – the big-brick camp will have a good argument as long as node management is manual rather than automatic. Needless to say, Microsoft and others appreciate this and are working furiously to improve the situation.

One debate that continues to rage is the shared-memory versus shared-disk versus shared-nothing cluster. This debate has evolved quite a bit over the last few years. At the beginning of the 1990s few SMPs scaled well beyond 8 processors. In retrospect this was due to a bus rather than switched memory interconnect, unsophisticated cache management, and ignorant software. Today, several vendors are shipping systems that scale to 32 or 64 processors, with either a uniform memory access (UMA) or non-uniform memory access (NUMA) architecture. These systems and their related software scale fairly well -- typically 75% of linear. Interestingly they achieve this scaling by recognizing that cache locality is critical, so sharing of memory is used very sparingly: indeed it is possible to view these systems as shared-nothing clusters where memory is used as a very fast (and very expensive) interconnect. Ideally, each of the 64 processors executes in its own cache and on its own partition of the shared memory, only occasionally reading and writing memory shared with other processors. So, paradoxically, shared memory systems have been successful by adopting a shared-nothing software architecture.

The shared-disk debate has been changed by the emergence of system-area-networks that allow high-bandwidth low-latency communications within the datacenter (Fiber Channel, Infiniband™, GigaNet, ServerNet, Gigabit per second Ethernet, and many others). Any node can now quickly and efficiently access any disk. But, in these systems, locality has become even more important. RAM is cheap and so the popular data is in RAM not on disk. It is generally faster to RPC to the cached version of the data than it is to read it off shared disk. So, shared disk systems have morphed to partitioned served-disk systems to improve locality. Even the server-consolidation boxes like EMC™, IBM-Shark™, and Compaq-ServerWorks™ that appear to be sharing disks are actually very impressive served disk systems with impressive cache management. Indeed, gradually they are running more and more of the database functionally within the box. So, paradoxically, shared-memory and shared-disk systems are becoming shared-nothing systems as they aim for better data locality. I believe the convergence point of this architectural shift will be a design where each disk is a full-blown application server providing a UDDI/WSDL/SOAP interface and hosting applications like Notes or Exchange or SAP. But that is a few years off.

The main theme of this section is that we have the performance problem well in hand. In part this is due to Moore's law, in part it is due to our finally bringing the software base to a modular and partitionable architecture that maps well to shared-nothing arrays (and hence to shared memory and shared disk arrays).

6. Summary

I have argued that we have a good story or at least good promises on many fronts, but that we have some major flaws in the story:

- Availability now hinges on
    1. **Automating management** so that human error is less of a factor in configuring and operating computer systems.
    2. **Fending off hacker-attacks** and other forms of sabotage. This was once a remote possibility, but today most popular websites are continuously under attack from dozens of adversaries at a time. Some of these adversaries are very competent and very expert.
- After functionality (actually getting the application to work), **manageability is the dominant cost** of operating most systems – there is no Moore's law for people. Yet the operations cost fall in concert with the reduced costs of hardware, software, networking, and faculties. The only way to do this is to automate these tasks.
- Agility is a special form of functionality. Agility is on the macro-level what we used to call extensibility on the micro-level.
- Scalability is addressed by manageability

So, the new role for "systems people" like you (if you got this far in the paper) and me is to focus on functionality, manageability, and defensive service. Making self-managing and self-healing systems, and making systems that detect and resist sabotage stand as our greatest system building challenge.

7. Acknowledgments

Thanks to Geotz Graefe, James Hamilton and George Spix for their constructive criticism of this memo.

8. References

B. Devlin et. al., "Scalability Terminology: Farms, Clones, Partitions, and Packs: RACS and RAPS," Microsoft Research Technical Report MSR-TR-99-85, December 1999 http://research.microsoft.com/scripts/pubs/view.asp?TR_ID=MSR-TR-99-85

S. Chaudhuri, et. al. "Self-Tuning Technology in Microsoft SQL Server". Data Engineering Bulletin 22(2): 20-26 (1999). http://www.informatik.uni-trier.de/~ley/db/journals/debu/ChaudhuriCGNZ99.html

Windows Update Service: (http://windowsupdate.microsoft.com/)