highscalability.com

Latency is Everywhere and it Costs You Sales - How to Crush it - High Scalability -

27-34 minutes

Update 8: The Cost of Latency by James Hamilton. James summarizing some latency info from Steve Souder, Greg Linden, and Marissa Mayer. *Speed [is] an undervalued and under-discussed asset on the web.*

Update 7: How do you know when you need more memcache servers?. Dathan Pattishall talks about using memcache not to scale, but to reduce latency and reduce I/O spikes, and how to use stats to know when more servers are needed.
Update 6: Stock Traders Find Speed Pays, in Milliseconds. Goldman Sachs is making record profits off a 500 millisecond trading advantage. Yes, latency matters. As an interesting aside, Libet found 500 msecs is about the time it takes the brain to weave together an experience of consciousness from all our sensor inputs.
Update 5: Shopzilla's Site Redo - You Get What You Measure. At the Velocity conference Phil Dixon, from Shopzilla, presented data showing a 5 second speed up resulted in a 25% increase in page views, a 10% increase in revenue, a 50% reduction in hardware, and a 120% increase traffic from Google. Built a new service oriented Java based stack. Keep it simple. Quality is a design decision. Obsessively easure everything. Used agile and built the site one page at a time to get feedback. Use proxies to incrementally expose users to new pages for A/B testing. Oracle

Coherence Grid for caching. 1.5 second page load SLA. 650ms server side SLA. Make 30 parallel calls on server. 100 million requests a day. SLAs measure 95th percentile, averages not useful. Little things make a big difference.
Update 4: Slow Pages Lose Users. At the Velocity Conference Jake Brutlag (Google Search) and Eric Schurman (Microsoft Bing) presented study data showing delays under half a second impact business metrics and delay costs increase over time and persist. Page weight not key. Progressive rendering helps a lot.
Update 3: Nati Shalom's Take on this article. Lots of good stuff on designing architectures for latency minimization.
Update 2: Why Latency Lags Bandwidth, and What it Means to Computing by David Patterson. Reasons: Moore's Law helps BW more than latency; Distance limits latency; Bandwidth easier to sell; Latency help BW, but not vice versa; Bandwidth hurts latency; OS overhead hurts latency more than BW. Three ways to cope: Caching, Replication, Prediction. We haven't talked about prediction. Games use prediction, i.e, project where a character will go, but it's not a strategy much used in websites.
Update: Efficient data transfer through zero copy. Copying data kills. This excellent article explains the path data takes through the OS and how to reduce the number of copies to the big zero.

Latency matters. Amazon found every 100ms of latency cost them 1% in sales. Google found an extra .5 seconds in search page generation time dropped traffic by 20%. A broker could lose $4 million in revenues per millisecond if their electronic trading platform is 5 milliseconds behind the competition.

The Amazon results were reported by Greg Linden in his presentation Make Data Useful. In one of Greg's slides Google VP Marissa Mayer, in reference to the Google results, is quoted as saying "**Users really respond to speed**." And everyone wants responsive users. Ka-ching! People hate waiting and they're repulsed by seemingly small delays.

The less interactive a site becomes the more likely users are to click away and do something else. Latency is the mother of interactivity. Though it's possible through various UI techniques to make pages subjectively feel faster, slow sites generally lead to higher customer defection rates, which lead to lower conversation rates, which results in lower sales. Yet for some reason latency isn't a topic talked a lot about for web apps. We talk a lot about about building high-capacity sites, but very little about how to build low-latency sites. We apparently do so at the expense of our immortal bottom line.

I wondered if latency went to zero if sales would be infinite? But alas, as Dan Pritchett says, Latency Exists, Cope!. So we can't hide the "latency problem" by appointing a Latency Czar to conduct a nice little war on latency. Instead, we need to learn how to minimize and manage latency. It turns out a lot of problems are better solved that way.

How do we recover that which is most meaningful--sales--and build low-latency systems?

I'm excited that the topic of latency came up. There are a few good presentations on this topic I've been dying for a chance to reference. And latency is one of those quantifiable qualities that takes real engineering to create. A lot of what we do is bolt together other people's toys. Building high-capacity low-latency system takes mad skills. Which is fun. And which may also account for why we see latency a core design skill in real-time and market trading type systems, but not web systems. We certainly want our nuclear power plant plutonium fuel rod lowering hardware to respond to interrupts with sufficient alacrity. While less serious, trading companies are always in a technological arms race to create lower latency systems. He with the fastest system creates a sort of private wire for receiving and acting on information faster than everyone else. Knowing who has the bestest price the firstest is a huge advantage. But if our little shopping cart takes an extra 500 milliseconds to display, the world won't end. Or will it?

Latency Defined

My unsophisticated definition of latency is that it is the elapsed time between A and B where A and B are something you care about. Low-latency and high-latency are relative terms. The latency requirements for a femptosecond laser are far different than for mail delivery via the pony express, yet both systems can be characterized by latency. A system has low-latency if it's low enough to meet requirements, otherwise it's a high-latency system.

Latency Explained

The best explanation of latency I've ever read is still It's the Latency, Stupid by admitted network wizard Stuart Cheshire. A wonderful and detailed rant explaining latency as it relates to network communication, but the ideas are applicable everywhere.

Stuart's major point: *If you have a network link with low bandwidth then it's an easy matter of putting several in parallel to make a combined link with higher bandwidth, but if you have a network link with bad latency then no amount of money can turn any number of them into a link with good latency.*

I like the parallel with sharding in this observation. We put shards in parallel to increase capacity, but request latency through the system remains the same. So if we want to increase interactivity we have to address every component in the system that introduces latency and minimize or remove it's contribution. There's no "easy" scale-out strategy for fixing latency problems.

Sources of Latency

My parents told me latency was brought by Santa Clause in the dead of night, but that turns out not to be true! So where does latency come from?

- **Low Level Infrastructure**. Includes OS / Kernel, Processors / CPU's, Memory, Storage related I/O, and Network related I/O.
- **High Level Infrastructure**. Analysis of sources of latency in downloading web pages by Marc Abrams. The study examines several sources of latency: DNS, TCP, Web server, network links, and routers. Conclusion: In most cases, roughly half of the time is spent from the moment the browser sends the acknowledgment completing the TCP connection establishment until the first packet containing page content arrives. The bulk of this time is the round trip delay, and only a tiny portion is delay at the server. This implies that the bottleneck in accessing pages over the Internet is due to the Internet itself, and not the server speed.
- **Software Processing**. Software processing accounts for much of the difficult to squeeze out latency in a system. In very rough terms a 2.0 GHz microprocessor can execute a few hundred lines of code every microsecond. Before a packet is delivered to an endpoint many thousands of instructions have probably already been executed. Then the handling software will spend many thousands more processing the message and then sending a reply. It all can add up to a substantial part of the latency budget. Included in this category are support services like databases, search engines, etc.
- **Frontend**. 80-90% of the end-user response time is spent on the frontend, so it makes sense to concentrate efforts there before heroically rewriting the backend.
- **Service Dependency Latency**. Dependent components increase latency. If component A calls compont B then the latency is the sum of the latency for each component and overall availability is reduced.

- **Propagation Latency**. The speed at which data travels through a link. For fibre optic cable, the rate of signal propagation is roughly two-thirds the speed of light in vacuum. Every 20km takes about 100 microseconds of propagation latency. To reduce latency your only choice is to reduce the distance between endpoints.
- **Transmission Latency**. The speed at which a data is transmitted on a communication link. On a 1Gbps network a 1000 bit packet takes about one millionth of a second to transmit. It's not dependent on distance. To reduce latency you need a faster link.
- **Geographical Distribution**. BCP (Business Continuity Planning) requires running in multiple datacenters which means added WAN latency constraints.
- **Messaging Latency**. The folks at 29west provide a great list forces that increase message latency: Intermediaries, Garbage Collection, Retransmissions, Reordering, Batching, CPU Scheduling, Socket Buffers, Network Queuing, Network Access Control, Serialization, Speed of Light.
  Draw out the list of every hop a client request takes and the potential number of latency gremlins is quite impressive.

The Downsides of Latency

Lower sales may be the terminal condition of latency problems, but the differential diagnosis is made of many and varied ailments. As latency increases work stays queued at all levels of the system which puts stress everywhere. It's like dementia, the system forgets how to do anything. Some of the problems you may see are: Queues grow; Memory grows; Timeouts cascade; Memory grows; Paging increases; Retries cascade; State machines reset; Locks are held longer; Threads block; Deadlock occurs; Predictability declines; Throughput declines; Messages drop; Quality plummets. For a better list take a look at The Many Flavors of System

Latency.. along the Critical Path of Peak Performance by Todd Jobson. A great analysis of the subject.

Managing Latency

The general algorithm for managing latency is:

- Continually map, monitor, and characterize all sources of latency.
- Remove and/or minimize all latency sources that are found.
Hardly a revelation, but it's actually rare for applications to view their work flow in terms of latency. This is part of the Log Everything All the Time mantra. Time stamp every part of your system. Look at mean latency, standard deviation, and outliers. See if you can't make the mean a little nicer, pinch in that standard deviation, and chop off some of those spikes. With latency variability is the name of the game, but that doesn't mean that variability can't be better controlled and managed. Target your latency slimming efforts where it matters the most and you get the most bang for your buck.

Next we will talk about various ideas for what you can do about latency once you've found it.

Dan Pritchett's Lessons for Managing Latency

Dan Pritchett is one of the few who has openly written on architecting for latency. Here are some of Dan's suggestions for structuring systems to manage latency:

- **Loosely Couple Components**. Loose coupling has a number of benifits: Tightly coupled systems are impossible distribute across data centers, tightly couples systems fail together, and loosely coupled systems can be independently scaled and engineered for latency.
- **Use Asynchronous Interfaces**. Set an expectation of asynchronous behavior between components. This allows you to

add latency when you need to make changes. Getting users on hooked on synchronous low-latency interactions doesn't allow for architecture flexibility. So start from the beginning with asynch semantics.

- **Horizontally Scale from the Start**. It's very difficult to change a monolithic schema once you meet a scaling wall. Start with a horizontal architecture so you don't build in too many problems that will be hard to remove later.
- **Create an Active/Active Architecture**. Most approaches to BCP take an active/passive approach, only one data center is active at a time. Creating an active/active system, where all data centers operate simultaneously allows users to be served from the closest data center which decreases latency.
- **Use a BASE (basically available, soft state, eventually consistent) Instead of ACID (atomicity, consistency, isolation, durability) Shared Storage Model**. BASE is derived from the CAP Theorem which is the highly counter intuitive notion that database services cannot ensure all three of the following properties at once: Consistency, Availability, Partition tolerance. A BASE based system is more tolerant to latency because it is an inherently partitioned and loosely coupled architecture and it uses eventual consistency. With eventual consistency you can make an update to one partition and return. You don't have to coordinate a transaction across multiple database servers, which makes a system have a higher and more variable latency.
Clearly each of these principles is a major topic all on their own. For more details please read: Dan Pritchett has written a few excellent papers on managing latency: The Challenges of Latency, Architecting for Latency, Latency Exists, Cope!.

GigaSpaces Lessons for Lowering Latency

GigsSpaces is an in-memory grid vendor and as such is on the vanguard of the RAM is the New Disk style of application building. In this approach disk is pushed aside for keeping all data in RAM. Following this line of logic GigaSpaces came up with these low latency architecture principles:

- Co-location of the tiers (logic, data, messaging, presentation) on the same physical machine (but with a shared-nothing architecture so that there is minimal communication between machines)
- Co-location of services on the same machine
- Maintaining data in memory (caching)
- Asynch communication to a persistent store and across geographical locations
  The thinking is the primary source of latency in a system centers around accessing disk. So skip the disk and keep everything in memory. Very logical. As memory is an order of magnitude faster than disk it's hard to argue that latency in such a system wouldn't plummet.

Latency is minimized because objects are in kept memory and work requests are directed directly to the machine containing the already in-memory object. The object implements the request behavior on the same machine. There's no pulling data from a disk. There isn't even the hit of accessing a cache server. And since all other object requests are also served from in-memory objects we've minimized the Service Dependency Latency problem as well.

GigaSpaces isn't the only player in this market. You might want to also take a look at: Scaleout Software, Grid Gain, Teracotta, GemStone, and Coherence. We'll have more on some of these products later.

Miscellaneous Latency Reduction Ideas

- **Cache**. No, really? Well it had to be said. See A Bunch of Great Strategies for Using Memcached and MySQL Better Together.
- **Use a CDN**. No, really? See What CDN would you recommend?.
- **Use a Caching Proxy Server**. At least this is a little less obvious. See Strategy: Front S3 with a Caching Proxy.
- **Enhance Your Web Operations Capability**. There are plenty of available tools to help you pinpoint and correct operation related problems. See Velocity Conference for more information.
- **Use Yslow to Make Your Pages Go**. Yslow is a tool to show sources of latency on the client side and suggest ways to fix any problems found. See Yslow to speed up your web pages.
- **Use an Edge DNS Accelerator**. This type of service "will ensure that a name server most accessible to the end user will pick up the request and respond." See Edge Acceleration Strategies: Akamai.
- **Optimize Virtual Machines**. People often forget VM's exact a performance tax. Virtualized I/O can suffer a substantial performance penalty. See if it can't be tuned.
- **Use Ajax to minimize perceived latency to the user.** Clever UI design can make a site feel faster than it really is.
- **Use a faster network**. A high speed InfiniBand link can have an end-to end latency of about 1 microsecond. Another option is a 10 GigE network.
- **Scale up**. Faster processors means less software induced latency.
- **Optimize firewalls**. An often hidden latency enhancer is your firewall system.
- **Use Small Memory Chunks When Using Java**. GC in Java kills latency. One way to minimize the impact of garbage collection on latency is to use more VMs and less memory in each VM instead of

VM with a lot of memory. This prevents a large GC run and makes latency more predictable.

- **Use a TCP Offload Engine (TOE)**. TOE tech offloads the TCP/IP stack from the main CPU and puts it on the network controller. This means network adapters can respond faster which means faster end-to-end communication. Network adapters respond faster because bus wait time is reduced as the number of transactions across the system I/O bus and memory bus are reduced.
- **Design low latency network topoligies**. Phil Dykstra in Issues Impacting Gigabit Networks:
Why don't most users experience high data rates? pinpoints poor network design as one major source of latency: *On a single high performance network today, measured latencies are typically ~1.5x - 3x that expected from the speed of light in fiber. This is mostly due to taking longer than line-of-site paths. Between different networks (via NAPs) latency is usually much worse. Some extra distance is required, based on the availability of fiber routes and interconnects, but much more attention should be given to minimizing latency as we design our network topologies and routing.*
- **Make TCP Faster**. FastTCP, for example, tweaks TCP to provide smoother and faster data delivery.
- **Copy Data Zero Times**. Efficient data transfer through zero copy. Copying data kills. This excellent article explains the path data takes through the OS and how to reduce the number of copies to the big zero.
- **Increase the speed of light**. Warp capability could really help speed up communication. Get to work on that!

Application Server Architecture Matters Again

With the general move over the past few years to a standard

shared nothing two-tierish architecture, discussion of application server architectures has become a neglected topic, mainly because there weren't application servers anymore. Web requests came in, data was retrieved from the database, and results were calculated and returned to the user. No application server. The web server became the application server. This was quite a change from previous architectures which were more application server oriented. Though they weren't called application servers, they were call daemons or even just servers (as in client-server).
Let's say we buy into RAM is the New Disk. This means we'll have many persistent processes filled with many objects spread over many boxes. A stream of requests are directed at each process and those requests must be executed in each process. How should those processes be designed?

Sure, having objects in memory reduces latency, but it's very easy through poor programming practice to lose all of that advantage. And then some. Fortunately we have a ton of literature on how to structure servers. I have a more thorough discussion here in Architecture Discussion. Also take a look at SEDA, an architecture for highly concurrent servers and ACE, an OO network programming toolkit in C++.

A few general suggestions:

- **Stop Serializing/Deserializing Messages**. It boggles my mind why we still serialize and deserialize messages. Leave messages in a binary compressed format and decode only on access. Very few activities waste more CPU and cause more lock contention through the memory library than does serialization.

- **Load Balance Across Read Replicas**. The more copies of objects you have the more work you can perform in parallel. Consider keeping objects replicas for both high availability and high scalability. This is the same strategy distributed file systems use handle more load. It works in-memory as well.

- **Don't Block**. The goal for a program is to use its whole CPU time quanta when it's scheduled to run. Don't block. Don't give the processor back to the OS for someone else to get it. Block for any reason and your performance tanks because not only do you incur the latency of the operation but there's added rescheduling latency as well. Who knows when your thread will get scheduled again?
- **Minimize Paging**. Thrashing is when a system experiences excessive page faults. More work is spent on moving memory around than is being given to tasks to perform real work. It's usually three orders of magnitude slower to access a page from disk instead of memory. Unfortunately, memory managers in most languages make reducing paging difficult as you have no control over where memory is placed or how it us used. With CPU speeds these days basically an operation is free when you are operating on paged-in memory.
- **Minimize/Remove locking.** Locks add latency and variability to a processing pipeline. A lock is a blocking operation. So you are choosing not to run when you have the CPU which means you incur a number of different forms of latency. Select a server architecture that minimizes the need for locks.

Colocate

Locating applications together reduces latency by reducing data hops. The number and location of network hops a message has to travel through is a big part of the end-to-end latency of a system. For example, from New York to the London Stock Exchange a round trip message takes 84 milliseconds to send, from Frankfurt it take 18 milliseconds, and from Tokyo it takes 208 milliseconds. If you want to minimize latency then the clear strategy is to colocate your service in the London Stock Exchange. Distance is minimized and you can probably use a faster network too.

Virtualization technology makes it easier than ever to compose separate systems together. Add a cloud infrastructure to that and it becomes almost easy to dramatically lower latencies by colocating applications.

Minimize the Number of Hops

Latency increases with each hop in a system. The fewer hops the less latency. So put those hops on a diet. Some hop reducing ideas are:

- **Colocation**. Colocation is one hop reducing strategy. It reduces the number of WAN links, routers, etc that a message has to go through. If a router takes 400 microsecond for each packet, for example, getting rid of that router reduces latency. Colocation also works for code and data, as in the GigaSpaces architecture. They maintain a sharded in-memory object cache so an extra database hop is avoided when executing an operation.
- **Simplify Software Architecture**. Remove intermediate daemons, brokers and other latency adding components. Dispatch work to where it will be processed as simply and fast as possible. Peer-to-peer architectures and sharding approaches are good at this. Avoid sending work into a hub for central dispatching. Dispatch as far out at the edge as possible.
- **Open a New Datacenter**. Facebook opened a new datacenter on the east coast in order to save 70 milliseconds.

Build Your own Field-programmable Gate Array (FPGA)

This one may seem a little off the wall, but creating your own custom FPGA may be a killer option for some problems. A FPGA is

a semiconductor device containing programmable logic. Typical computer programs are a series of instructions that are loaded and interpreted by a general purpose microprocessor, like the one in your desk top computer. Using a FPGA it's possible to bypass the overhead of a general purpose microprocessor and code your application directly into silicon. For some classes of problems the performance increases can be dramatic.
FPGAs are programmed with your task specific algorithm. Usually something compute intensive like medical imaging, modeling bond yields, cryptography, and matching patterns for deep packet inspections. I/O heavy operations probably won't benefit from FPGAs. Sure, the same algorithm could be run on a standard platform, but the advantage FPGAs have is even though they may run at a relatively low clock rates, FPGAs can perform many calculations in parallel. So perhaps orders-of-magnitude more work is being performed each clock cycle. Also, FPGAs often use content addressable memory which provides a significant speedup for indexing, searching, and matching operations. We also may see a move to FPGAs because they use less power. Stay lean and green.

In embedded projects FPGAs and ASICS (application-specific integrated circuit) are avoided like the plague. If you can get by with an off-the-shelf microprocessors (Intel, AMD, ARM, PPC, etc) you do it. It's a time-to-market issue. Standard microprocessors are, well, standard, so that makes them easy to work with. Operating systems will already have board support packages for standard processors, which makes building a system faster and cheaper. Once custom hardware is involved it becomes a lot of work to support the new chip in hardware and software. Creating a software only solution is much more flexible in a world where constant change rules. Hardware resists change. So does software, but since people think it doesn't we have to act like software is infinitely malleable.

Sometimes hardware is the way to go. If you are building a NIC that has to process packets at line speed the chances are an off-the-shelf processor won't be cost effective and may not be fast enough. Your typical high end graphics card, for example, is a marvel of engineering. Graphics cards are so powerful these days distributed computation projects like Folding@home get a substantial amount of their processing power from graphics cards. Traditional CPUs are creamed by NVIDIA GeForce GPUs which perform protein-folding simulations up to 140 times faster. The downside is GPUs require very specialized programming, so it's easier to write for a standard CPU and be done with it.

That same protein folding power can be available to your own applications. ACTIV Financial, for example, uses a custom FGPA for low latency processing of high speed financial data flows. ACTIV's competitors use a traditional commodity box approach where financial data is processed by a large number of commodity servers. Let's say an application takes 12 servers. Using a FPGA the number of servers can be collapsed **down to one** because more instructions are performed simultaneously which means fewer machines ar needed. Using the FPGA architecture they process **20 times more messages** than they did before and have **reduced latency from one millisecond down to less than 100 microseconds**.

Part of the performance improvement comes from the high speed main memory and network IO access FPGAs enjoy with the processor. Both Intel and AMD make it relatively easy to connect FPGAs to their chips. Using these mechanisms data moves back and forth between your processing engine and the main processor with minimal latency. In a standard architecture all this communication and manipulation would happen over a network.

FPGAs are programmed using hardware description languages like Verilog and VHDL. You can't get away from the hardware when programming FPGAs, which is a major bridge to cross for us software types. Many moons ago I took a Verilog FPGA programming class. It's not easy, nothing is ever easy, but it is possible. And for the right problem it might even be worth it.

Related Articles

- The Challenges of Latency by Dan Pritchett
- Latency Exists, Cope! by Dan Pritchett
- Architecting for Latency by Dan Pritchett
- BASE: An ACID Alternative by Dan Pritchett, eBay
- Comet: Sub-Second Latency with 10K+ Concurrent Users by Alexander Olaru
- It's the Latency, Stupid by Stuart Cheshire
- Latency and the Quest for Interactivity by Stuart Cheshire
- The importance of bandwidth versus latency by Dion Almaer
- Fallacies of Distributed Computing - The second fallacy is "Latency is Zero"
- List of device bandwidths
- Computing over a high-latency network means you have to bulk up by Raymond Chen
- AJAX Latency problems: myth or reality? by Jep Castelein
- RAM Guide: Part I DRAM and SRAM Basics and Part 2 by Jon "Hannibal" Stokes
- The Many Flavors of System Latency.. along the Critical Path of Peak Performance and Processors and Performance : Chips,

MIPS, and Sizing blips.. by Todd Jobson. A very detailed and helpful analysis of latency sources.

- Ethernet Latency: The Hidden Performance Killer by Kevin Burton
- Network latency vs. end-to-end latency by Nati Shalom
- Low-Latency Delivery Enters Mainstream; But Standard Measurement Remains Elusive by Andrew Delaney
- The three faces of latency by By Scott Parsons, Chief Scientist at Exegy, Inc.
- Architecture Discussion
- The JVM needs Value Types - Solving the next bottleneck. Value types use less space, less paging, less memory allocation, better cache usage, better garbage collection profile.
- Latency, Bandwidth, and Response Times by Chris Loosley.
- Anatomy of real-time Linux architectures by M. Time Jones.
- True Cost of Latency by GemStone