acmqueue A Guided Tour through Data-center Networking

A good user experience depends on predictable performance within the data-center network.

Dennis Abts, Bob Felderman, Google

The magic of the cloud is that it is always on and always available from anywhere. Users have come to expect that services are there when they need them. A data center (or warehouse-scale computer) is the nexus from which all the services flow. It is often housed in a nondescript warehouse-sized building bearing no indication of what lies inside. Amidst the whirring fans and refrigerator-sized computer racks is a tapestry of electrical cables and fiber optics weaving everything together—the data-center network. This article provides a "guided tour" through the principles and central ideas surrounding the network at the heart of a data center — the modern-day loom that weaves the digital fabric of the Internet.

DATA-CENTER DEVELOPMENT

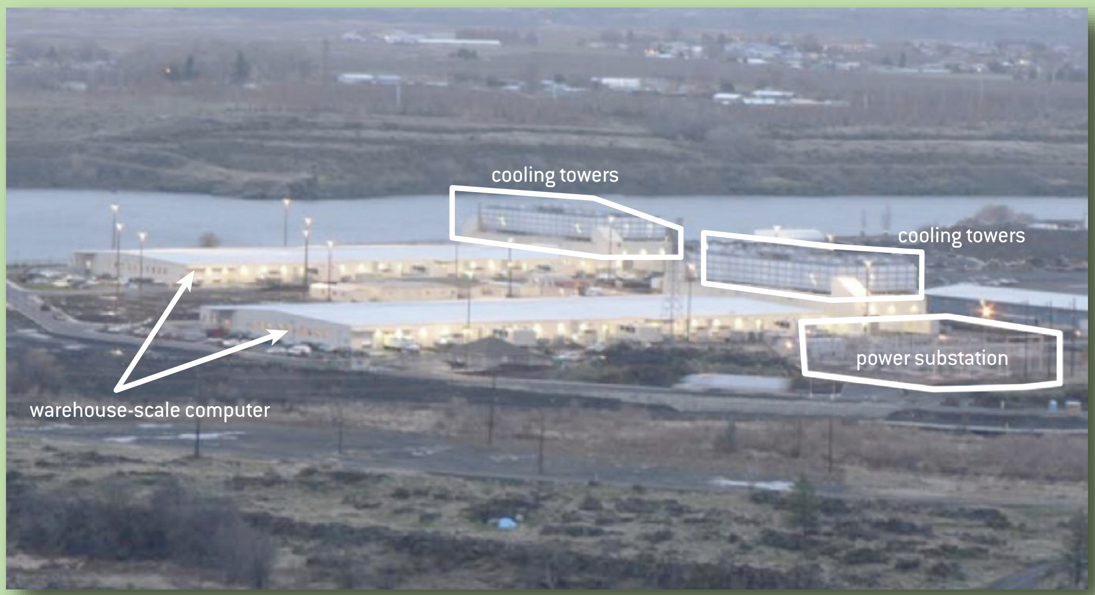
Large-scale parallel computers are grounded in HPC (high-performance computing) where kilo-processor systems were available 15 years ago. HPC systems rely on fast (low-latency) and efficient interconnection networks capable of providing both high bandwidth and efficient messaging for fine-grained (e.g., cache-line size) communication. This zealous attention to performance and low latency migrated to financial enterprise systems, where a fraction of a microsecond can make a difference in the value of a transaction.

In recent years, Ethernet networks have made significant progress toward bridging the performance and scalability gap between capacity-oriented clusters built using COTS (commodity-off-the-shelf) components and purpose-built custom system architectures. This is evident from the growth of Ethernet as a cluster interconnect on the Top500 list of most powerful computers (top500.org). A decade ago high-performance networks were mostly custom and proprietary interconnects, and Ethernet was used by only 2 percent of the Top500 systems. Today, however, more than 42 percent of the most powerful computers are using Gigabit Ethernet, according to the November 2011 list of Top500 computers. A close second place is InfiniBand, which is used by about 40 percent of the systems. These standards-based interconnects combined with economies of scale provide the genetic material of a modern data-center network.

A modern data center,[13,17,24] as shown in figure 1, is home to tens of thousands of *hosts,* each consisting of one or more processors, memory, network interface, and local high-speed I/O (disk or flash). Compute resources are packaged into racks and allocated as *clusters* consisting of thousands of hosts that are tightly connected with a high-bandwidth network. While the network plays a central role in the overall system performance, it typically represents only 10-15 percent of the cluster cost. Be careful not to confuse cost with value—the network is to a cluster computer what the central nervous system is to the human body.

Each cluster is homogeneous in both processor type and speed. The thousands of hosts are orchestrated to exploit thread-level parallelism central to many Internet workloads as they divide

FIGURE 1 **An Example Data Center and Warehouse-Scale Computer**



incoming requests into parallel subtasks and weave together results from many subtasks across thousands of cores. In general, all parallel subtasks must complete in order for the request to complete. As a result, the maximum response time of any one subtask will dictate the overall response time.[25] Even in the presence of abundant thread-level parallelism, the communication overhead imposed by the network and protocol stack can ultimately limit application performance as the effects of Amdahl's law[2] creep in.

The high-level system architecture and programming model shape both the programmer's conceptual view and application usage. The latency and bandwidth "cost" of local (DRAM) and remote (network) memory references are often baked into the application as programming tradeoffs are made to optimize code for the underlying system architecture. In this way, an application organically grows within the confines of the system architecture.

The cluster-application usage model, either dedicated or shared among multiple applications, has a significant impact on SLAs (service-level agreements) and application performance. HPC applications typically use the system in a dedicated fashion to avoid contention from multiple applications and reduce the resulting variation in application performance. On the other hand, Web applications often rely on services sourced from multiple clusters, where each cluster may have several applications simultaneously running to increase overall system utilization. As a result, a data-center cluster may use virtualization for both performance and fault isolation, and Web applications are programmed with this sharing in mind.

Web applications such as search, e-mail, and document collaboration are scheduled resources and run within a cluster.[4,8] User-facing applications have soft realtime latency guarantees or SLAs that the application must meet. In this model, an application has roughly tens of milliseconds to reply to the user's request, which is subdivided and dispatched to worker threads within the cluster. The worker threads generate replies that are aggregated and returned to the user. Unfortunately, if a portion of the workflow does not execute in a timely manner, then it may exceed a specified timeout delay—as a result of network congestion, for example—and consequently some portion of the coalesced results will be unavailable and thus ignored. This needlessly wastes both CPU cycles and network bandwidth, and may adversely affect the computed result.

To reduce the likelihood of congestion, the network can be overprovisioned by providing ample bandwidth for even antagonistic traffic patterns. Overprovisioning within large-scale networks is prohibitively expensive. Alternatively, implementing QoS (quality of service) policies to segregate traffic into distinct classes and provide performance isolation and high-level traffic engineering is a step toward ensuring that application-level SLAs are satisfied. Most QoS implementations are implemented by switch and NIC (network interface controller) hardware where traffic is segregated based on priority explicitly marked by routers and hosts or implicitly steered using port ranges. The goal is the same: a high-performance network that provides predictable latency and bandwidth characteristics across varying traffic patterns.

DATA-CENTER TRAFFIC

Traffic within a data-center network is often measured and characterized according to *flows*, which are sequences of packets from a source to a destination host. When referring to Internet protocols, a flow is further refined to include a specific source and destination port number and transport type—UDP or TCP, for example. Traffic is asymmetric with client-to-server requests being abundant but generally small. Server-to-client responses, however, tend to be larger flows; of course, this, too, depends on the application. From the purview of the cluster, Internet traffic becomes highly aggregated, and as a result the *mean* of traffic flows says very little because aggregated traffic exhibits a high degree of variability and is non-Gaussian.[16]

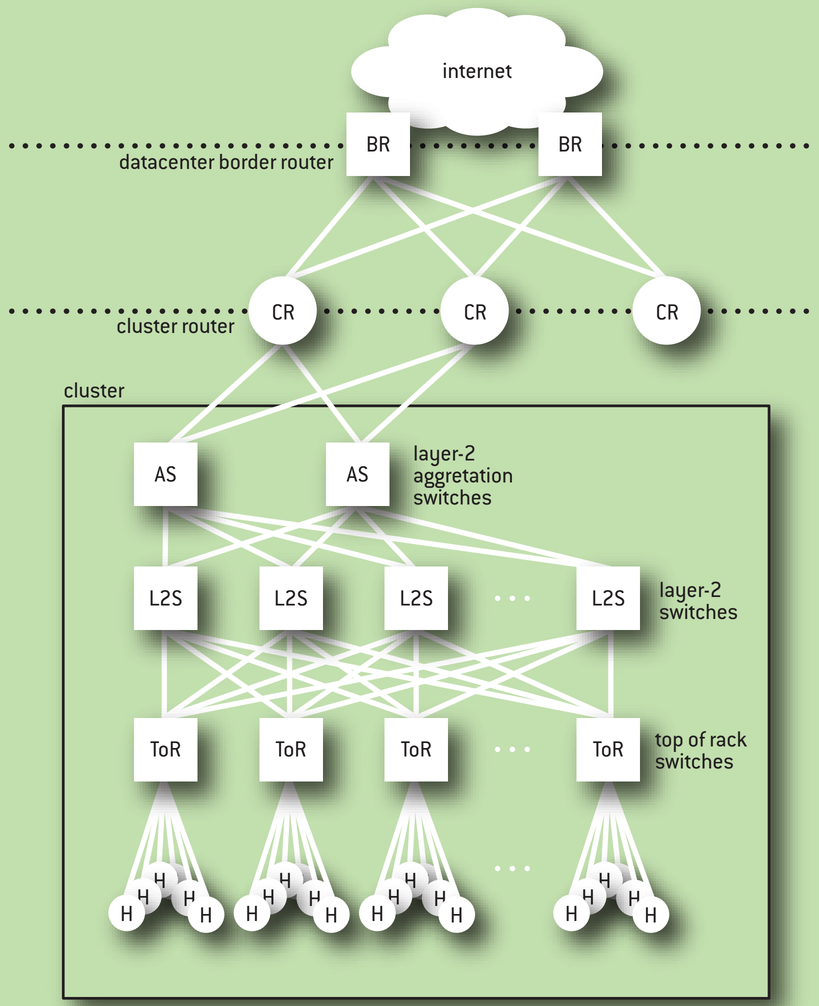
As a result, a network that is only 10 percent utilized can see lots of packet discards when running a Web search. To understand individual flow characteristics better, applications are instrumented to "sample" messages and derive a distribution of traffic flows; this knowledge allows you to infer a taxonomy of network traffic and classify individual flows. The most common classification is bimodal, using the so-called "elephant" and "mice" classes.

Elephant flows have a large number of packets and are generally long lived; they exhibit "bursty" behavior with a large number of packets injected in the network over a short time. Traffic within a flow is generally ordered, which means that elephant flows can create a set of "hotspot" links that can lead to tree saturation or discarded packets in networks that use lossy flow control. The performance impact from elephant flows can be significant. Despite the relatively low number of these flows—say less than 1 percent—they can account for more than half the data volume on the network.

The transient load imbalance induced by elephant flows can adversely affect any innocent-bystander flows that are patiently waiting for a heavily utilized link common to both routes. For example, an elephant flow from A to B might share a common link with a flow from C to D. Any long-lived contention for the shared link increases the likelihood of discarding a packet from the C-to-D flow. Any packet discards will result in an unacknowledged packet at the sender's transport layer and be retransmitted when the timeout period expires. Since the timeout period is generally one or two orders of magnitude more than the network's round-trip time, this additional latency[22] is a significant source of performance variation.[3]

Today's typical multitiered data-center network[23] has a significant amount of *oversubscription,* where the hosts attached to the rack switch (i.e., first tier) have significantly more—say an order of magnitude more—provisioned bandwidth between one another than they do with hosts in *other* racks. This *rack affinity* is necessary to reduce network cost and improve utilization. The traffic intensity emitted by each host fluctuates over time, and the transient load imbalance that results from this varying load can create contention and ultimately result in discarded packets for flow control. Traffic *between* clusters is typically less time-critical, so it can can be staged and scheduled. Inter-cluster traffic is less orchestrated and consists of much larger payloads, whereas intra-cluster traffic is often fine-grained with bursty behavior. At the next level, between data centers, bandwidth

FIGURE 2 **A Conventional Tree-Like Datacenter Network Topology**



is often very expensive over vast distances with highly regulated traffic streams and patterns so that expensive links are highly utilized. When congestion occurs the most important traffic gets access to the links. Understanding the granularity and distribution of network flows is essential to capacity planning and traffic engineering.

DATA-CENTER NETWORK ARCHITECTURE

The network *topology* describes precisely how switches and hosts are interconnected. This is commonly represented as a graph in which vertices represent switches or hosts, and links are the edges that connect them. The topology is central to both the performance and the cost of the network. The topology affects a number of design tradeoffs, including performance, system packaging, path diversity, and redundancy, which, in turn, affect the network's resilience to faults, average and maximum cable length, and, of course, cost.[12] The *Cisco Data Center Infrastructure 3.0 Design Guide*[6] describes common practices based on a tree-like topology[15] resembling early telephony networks proposed by Charles Clos,[7] with bandwidth aggregation at different levels of the network.

A fat-tree or folded-Clos topology, similar to that shown in figure 2, has an aggregate bandwidth that grows in proportion to the number of host ports in the system. A *scalable* network is one in which increasing the number of ports in the network should linearly increase the delivered bisection bandwidth. Scalability and reliability are inseparable since growing to large system size requires a robust network.

NETWORK ADDRESSING

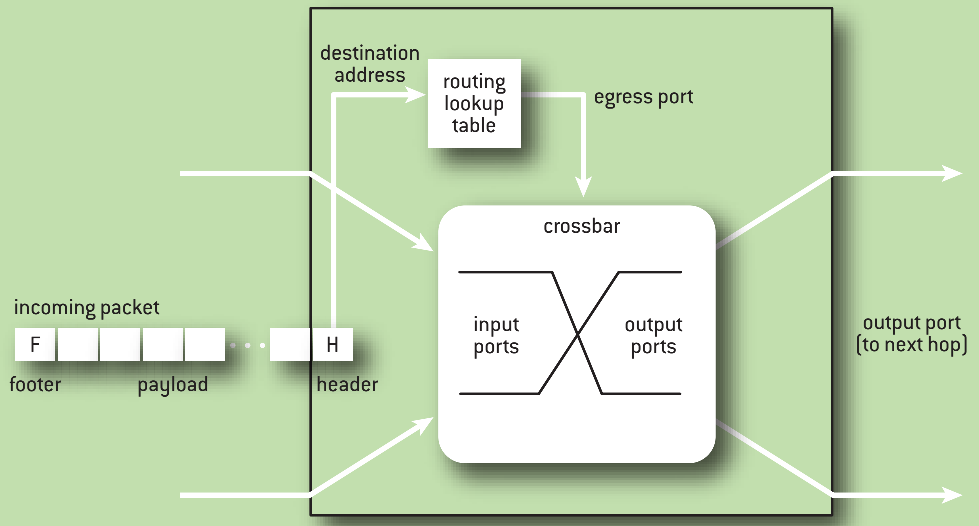
A host's *address* is how endpoints are identified in the network. Endpoints are distinguished from intermediate switching elements traversed en route since messages are created by and delivered to an endpoint. In the simplest terms, the address can be thought of as the numerical equivalent of a host name similar to that reported by the Unix `hostname` command.

An address is unique and must be represented in a canonical form that can be used by the *routing function* to determine where to route a packet. The switch inspects the packet header corresponding to the *layer* in which routing is performed—for example, IP address from layer 3 or Ethernet address from layer 2. Switching over Ethernet involves ARP (address resolution protocol) and RARP (reverse address resolution protocol) that broadcast messages on the layer 2 network to update local caches mapping layer 2 to layer 3 addresses and vice versa. Routing at layer 3 requires each switch to maintain a subnet *mask* and assign IP addresses statically or disseminate host addresses using DHCP (dynamic host configuration protocol), for example.

The layer 2 routing tables are automatically populated when a switch is plugged in and learns its identity and exchanges route information with its peers; however, the capacity of the packet-forwarding tables is limited to, say, 64K entries. Further, each layer 2 switch will participate in an STP (spanning tree protocol) or use the TRILL (transparent interconnect of lots of links) link-state protocol to exchange routing information and avoid transient routing loops that may arise while the link state is exchanged among peers. Neither layer 2 nor layer 3 routing is perfectly suited to data-center networks, so to overcome these limitations many new routing algorithms have been proposed (e.g., PortLand[1,18] and VL2 [11]).

FIGURE 3

Example Packet Routing Through a Switch Chip



ROUTING

The *routing* algorithm determines the path a packet traverses through the network. A packet's route, or path, through the network can be asserted when the message is created, called *source* routing, or may be asserted hop by hop in a distributed manner as a packet visits intermediate switches. Source routing requires that every endpoint know the prescribed path to reach all other endpoints, and each source-routed packet carries the full information to determine the set of port/link traversals from source to destination endpoint. As a result of this overhead and inflexible fault handling, source-routed packets are generally used only for topology discovery and network initialization, or during fault recovery when the state of a switch is unknown. A more flexible method of routing uses distributed *lookup tables* at each switch, as shown in figure 3.

For example, consider a typical Ethernet switch. When a packet arrives at a switch input port, it uses fields from the packet header to index into a lookup table and determine the *next* hop, or egress port, from the current switch.

A good topology will have abundant *path diversity* in which multiple possible egress ports may exist, with each one leading to a distinct path. Path diversity in the topology may yield ECMP (equal-cost multipath) routing; in that case the routing algorithm attempts to load-balance the traffic flowing across the links by spreading traffic uniformly. To accomplish this *uniform spreading*, the routing function in the switch will hash several fields of the packet header to produce a deterministic egress port. In the event of a link or switch failure, the routing algorithm will take advantage of path diversity in the network to find another path.

A path through the network is said to be *minimal* if no shorter (i.e., fewer hops) path exists;

of course, there may be multiple minimal paths. A fat-tree topology,[15] for example, has multiple minimal paths between any two hosts, but a butterfly topology[9] has only a single minimal path between any two hosts. Sometimes selecting a *non-minimal* path is advantageous—for example, to avoid congestion or to route around a fault. The length of a non-minimal path can range from min+1 up to the length of a Hamiltonian path visiting each switch exactly once. In general, the routing algorithm might consider non-minimal paths of a length that is one more than a minimal path, since considering *all* non-minimal paths would be prohibitively expensive.

NETWORK PERFORMANCE

This section discusses the etiquette for sharing the network resources—specifically, the physical links and buffer spaces are resources that require *flow control* to share them efficiently. Flow control is carried out at different levels of the network stack: data-link, network, transport layer, and possibly within the application itself for explicit coordination of resources. Flow control that occurs at lower levels of the communication stack is transparent to applications.

FLOW CONTROL

Network-level flow control dictates how the input buffers at each switch or NIC are managed: store-and-forward, virtual cut-through,[14] or wormhole,[19] for example. To understand the performance implications of flow control better, you must first understand the total delay, T, a packet incurs:

$$T = H(t_r + Lt_p) + t_s$$

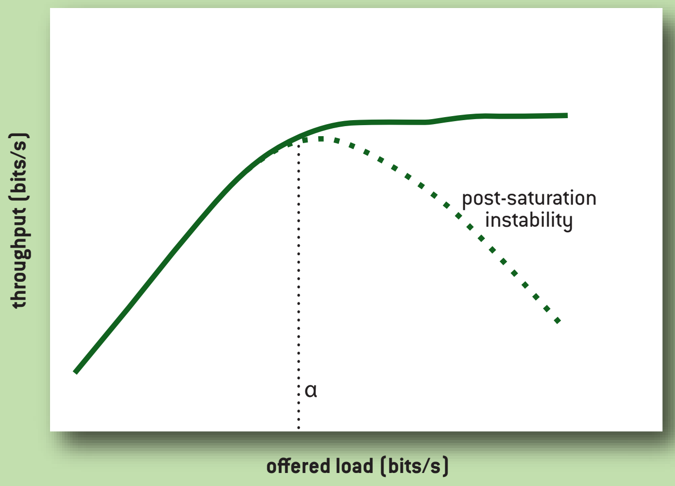
H is the number of *hops* the packet takes through the network; t_r is the fall-through latency of the switch, measured from the time the first flit (flow-control unit) arrives to when the first flit exits; and t_p is the propagation delay through average cable length L. For short links—say, fewer than 10 meters—electrical signaling is cost effective. Longer links, however, require fiber optics to communicate over the longer distances. Signal propagation in electrical signaling (5 nanoseconds per meter) is faster than it is in fiber (6 nanoseconds per meter).

Propagation through electrical cables occurs at subluminal speeds because of a frequency-dependent component at the surface of the conductor, or "skin effect," in the cable. This limits the signal velocity to about three-quarters the speed of light in a vacuum. Signal propagation in optical fibers is even slower because of dielectric waveguides used to alter the refractive index profile so that higher-velocity components of the signal (i.e., shorter wavelengths) will travel longer distances and arrive at the same time as lower-velocity components, limiting the signal velocity to about two-thirds the speed of light in a vacuum. Optical signaling must also account for the time necessary to perform electrical-to-optical signal conversion, and vice versa.

The *average* cable length, L, is largely determined by the topology and the physical placement of system racks within the data center. The packet's serialization latency, t_s, is the time necessary to squeeze the packet onto a narrow serial channel and is determined by the *bit rate* of the channel. For example, a 1,500-byte Ethernet packet (frame) requires more than 12 μs (ignoring any interframe gap time) to be squeezed onto a 1-Gb/s link. With store-and-forward flow control, as its name suggests, a packet is buffered at each hop before the switch does anything with it.

FIGURE 4

Throughput (Accepted Bandwidth) as Load Varies



$T_{sf} = H(t_r + Lt_p + t_s)$

As a result, the serialization delay, t_s, is incurred at *each* hop, instead of just at the destination endpoint as is the case with virtual cut-through and wormhole flow control. This can potentially add on the order of 100 µs to the round-trip network delay in a data-center network.

A *stable* network monotonically delivers messages as shown by a characteristic throughput-load curve in figure 4. In the absence of end-to-end flow control, however, the network can become unstable, as illustrated by the dotted line in the figure, when the offered load exceeds the saturation point, α. The saturation point is the offered load beyond which the network is said to be *congested*. In response to this congestion, packets may be discarded to avoid overflowing an input buffer. This *lossy* flow control is commonplace in Ethernet networks.

Discarding packets, while conceptually simple and easy to implement, puts an onus on transport-level mechanisms to detect and retransmit lost packets. Note that packets that are lost or corrupted during transmission are handled by the same transport-level reliable delivery protocol. When the offered load is low (less than α), packet loss as a result of corruption is rare, so paying the relatively large penalty for transport-level retransmission is generally tolerable. Increased traffic (greater than α) and adversarial traffic patterns will cause packet discards after the switch's input queue is exhausted. The resulting retransmission will only further exacerbate an already congested network, yielding an unstable network that performs poorly, as shown by the dotted line in figure 4. Alternatively, with *lossless* flow control, when congestion arises packets may be blocked or held at the source until resources are available.

A *global congestion control* mechanism prevents the network from operating in the post-saturation region. Most networks use *end-to-end flow control*, such as TCP,[5] which uses a windowing mechanism between pairs of source and sink in an attempt to match the source's injection rate with the sink's acceptance rate. TCP, however, is designed for reliable packet delivery, not necessarily timely packet delivery; as a result, it requires tuning (TCP congestion-control algorithms will auto-tune to find the right rate) to balance performance and avoid unnecessary packet duplication from eagerly retransmitting packets under heavy load.

IMPROVING THE NETWORK STACK

Several decades ago the network designers of early workstations made tradeoffs that led to a single TCP/IP/Ethernet network stack, whether communicating over a few meters or a few kilometers. As processor speed and density improved, the cost of network communication grew relative to processor cycles, exposing the network stack as a critical latency bottleneck.[22] This is, in part, the result of a user-kernel context switch in the TCP/IP/Ethernet stack—and possibly additional work to copy the message from the application buffer into the kernel buffer and back again at the receiver. A two-pronged hardware/software approach tackled this latency penalty: OS bypass and zero copy, both of which are aimed at eliminating the user-kernel switch for every message and avoiding a redundant memory copy by allowing the network transport to grab the message payload directly from the user application buffers.

To ameliorate the performance impact of a user/kernel switch, OS bypass can be used to deposit a message directly into a user-application buffer. The application participates in the messaging protocol by spin-waiting on a doorbell memory location. Upon arrival, the NIC deposits the message contents in the user-application buffer, and then "rings" the doorbell to indicate message arrival by writing the offset into the buffer where the new message can be found. When the user thread detects the updated value, the incoming message is processed entirely from user space.

Zero-copy message-passing protocols avoid this additional memory copy from user to kernel space, and vice versa at the recipient. An interrupt signals the arrival of a message, and an interrupt handler services the new message and returns control to the user application. The interrupt latency—the time from when the interrupt is raised until control is handed to the interrupt handler—can be significant, especially if interrupt coalescing is used to amortize the latency penalty across multiple interrupts. Unfortunately, while interrupt coalescing improves message efficiency (i.e., increased effective bandwidth), it does so at the cost of both increased message latency and latency variance.

SCALABLE, MANAGEABLE, AND FLEXIBLE

In general, *cloud computing* requires two types of services: user-facing computation (e.g., serving Web pages) and inward computation (e.g., indexing, search, map/reduce, etc.). Outward-facing functionality can sometimes be done at the "border" of the Internet where commonly requested pages are cached and serviced by edge servers, while inward computation is generally carried out by a cluster in a data center with tightly coupled, orchestrated communication. User demand is diurnal for a geographic region; thus, multiple data centers are positioned around the globe to accommodate the varying demand. When possible, demand may be spread across nearby data centers to load-balance the traffic.

The sheer enormity of this computing infrastructure makes nimble deployment very challenging. Each cluster is built up rack by rack and tested as units (rack, top-of-rack switch, etc.), as well as in its entirety with production-level workloads and traffic intensity.

The cluster ecosystem grows organically over its lifespan, propelled by the rapid evolution of software—both applications and, to a lesser extent, the operating system. The fluid-like software demands of Web applications often consume the cluster resources that contain them, making *flexibility* a top priority in such a fluid system. For example, adding 10 percent additional storage capacity should mean adding no more than 10 percent more servers to the cluster. This linear growth function is critical to the *scalability* of the system—adding fractionally more servers results in a commensurate growth in the overall cluster capacity. Another aspect of this flexibility is the *granularity* of resource additions, which is often tied to the cluster packaging constraints. For example, adding another rack to a cluster, with, say, 100 new servers, is more manageable than adding a whole row, with tens of racks, on the data-center floor.

Even a modest-sized cluster will have several kilometers of fiber-optic cable acting as a vast highway system interconnecting racks of servers organized as multiple rows on the data-center floor. The data-center network topology and resulting cable complexity is "baked in" and remains a rigid fixture of the cluster. Managing cable complexity is nontrivial, which is immediately evident from the intricately woven tapestry of fiber-optic cabling laced throughout the data center. It is not uncommon to run additional fiber for redundancy, in the event of a cable failure in a "bundle" of fiber or for planned bandwidth growth. Fiber cables are carefully measured to allow some slack and to satisfy the cable's bend radius, and they are meticulously labeled to make troubleshooting less of a needle-in-a-haystack exercise.

RELIABLE AND AVAILABLE

Abstraction is the Archimedes lever that lifts many disciplines within computer science and is used extensively in both computer system design and software engineering. Like an array of nested Russian dolls, the network-programming model provides abstraction between successive layers of the networking stack, enabling platform-independent access to both data and system management. One such example of this type of abstraction is the *protocol buffer,*[21] which provides a structured message-passing interface for Web applications written in C++, Java, or Python.

Perhaps the most common abstraction used in networking is the notion of a *communication channel* as a virtual resource connecting two hosts. The TCP communication model provides this abstraction to the programmer in the form of a file descriptor, for example, where reads and writes performed on the socket result in the corresponding network transactions, which are hidden from the user application. In much the same way, the InfiniBand QP (queue-pair) verb model provides an abstraction for the underlying send/receive hardware queues in the NIC. Besides providing a more intuitive programming interface, abstraction also serves as a protective sheath around software when faults arise, depositing layers of software sediment to insulate from critical faults (e.g., memory corruption or, worse, host power-supply failure).

Bad things happen to good software. Web applications must be designed to be fault aware and, to the extent possible, resilient in the presence of a variety of failure scenarios.[10] The network is responsible for the majority of the unavailability budget for a modern cluster. Whether the problem is a gamma ray causing a soft error in memory or an inattentive worker accidentally unearthing a fiber-optic line, the operating system and underlying hardware substrate work in concert to foster a robust ecosystem for Web applications.

The data-center network serves as a "central nervous system" for information exchange between cooperating tasks. The network's functionality is commonly divided into *control* and *data* planes.

The control plane provides an ancillary network juxtaposed with the data network and tasked with "command and control" for the primary data plane. The control plane is an autonomic system for configuration, fault detection and repair, and monitoring of the data plane. The control plane is typically implemented as an embedded system within each switch component and is tasked with fault detection, notification, and repair when possible.

For example, when a network link fails or has an uncharacteristically high number of transmission errors, the control plane will *reroute* the network to avoid the faulty link. This entails recomputing the routes according to the *routing algorithm* and emplacing new entries in the routing tables of the affected switches. Of course, the effects of this patchwork are not instantaneous. Once the routing algorithm computes new routes, taking into consideration the newfound faulty links, it must disseminate the routes to the affected switches. The time needed for this information exchange is referred to as *convergence time,* and a primary goal of the routing protocol is to ensure it is optimally confined to a small epoch.

Fault recovery is a very complicated subject and confounds all but the simplest of data-center networks. Among the complicating factors are *marginal* links that cause "flapping" by transitioning between active and inactive (i.e., up and down), repeatedly creating a deluge of error notifications and resulting route recomputation based on fluctuating and inconsistent link status. Some link-layer protocols allow the link speed to be adjusted downward in hopes of improving the link quality. Of course, lowering the link speed results in a reduced bandwidth link, which in turn may limit the overall bandwidth of the network or at the very least will create load imbalance as a result of increased contention across the slow link. Because of these complicating factors, it is often better to logically excise the faulty link from the routing algorithm until the physical link can be replaced and validated.

CONCLUSION

The data-center network is generally regarded as a critical design element in the system architecture and the skeletal structure upon which processor, memory, and I/O devices are dynamically shared. The evolution from 1G to 10G Ethernet and the emerging 40G Ethernet has exposed performance bottlenecks in the communication stack that require better hardware-software coordination for efficient communication. Other approaches by Solarflare, Myricom, and InfiniBand, among others, have sought to reshape the conventional sockets programming model with more efficient abstractions. Internet sockets, however, remain the dominant programming interface for data-center networks.

Network performance and reliability are key design goals, but they are tempered by cost and serviceability constraints. Deploying a large cluster computer is done incrementally and is often limited by the power capacity of the building, with power being distributed across the cluster network so that a power failure impacts only a small fraction—say, less than 10 percent—of the hosts in the cluster. When hardware fails, as is to be expected, the operating system running on the host coordinates with a higher-level hypervisor or cluster operating system to allow failures to be replaced in situ without draining traffic in the cluster. Scalable Web applications are designed to expect occasional hardware failures, and the resulting software is by necessity resilient.

A good user experience relies on *predictable* performance, with the data-center network delivering predictable latency and bandwidth characteristics across varying traffic patterns. With single-thread performance plateauing, microprocessors are providing more cores to keep pace with the relentless march of Moore's law. As a result, applications are looking for increasing thread-level parallelism and scaling to a large core count with a commensurate increase in communication among cores. This trend is motivating *communication-centric* cluster computing with tens of thousands of cores in unison, reminiscent of a flock darting seamlessly amidst the clouds.

REFERENCES

1. Al-Fares, M., Loukissas, A., Vahdat, A. 2008. A scalable, commodity data-center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication* (SIGCOMM '08): 63-74; http://doi.acm.org/10.1145/1402958.1402967.
2. Amdahl's law; http://en.wikipedia.org/wiki/Amdahl's_law.
3. Ballani, H., Costa, P., Karagiannis, T., Rowstron, A. 2011. Towards predictable data-center networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (SIGCOMM '11): 242-253; http://doi.acm.org/10.1145/2018436.2018465.
4. Barroso, L.A.; Dean, J.; Holzle, U. 2003. Web search for a planet: the Google cluster architecture. *IEEE Micro* 23 (2):22-28; http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1196112&isnumber=26907.
5. Cerf, V., Icahn R. E. 2005. A protocol for packet network intercommunication. *SIGCOMM Computer Communication Review* 35(2):71-82; http://doi.acm.org/ 10.1145/1064413.1064423.
6. *Cisco Data Center Infrastructure 3.0 Design Guide*. Data Center Design—IP Network Infrastructure; http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_3_0/DC-3_0_IPInfra.html.
7. Clos, C. 1953. A study of non-blocking switching networks. *The Bell System Technical Journal* 32(2):406–424.
8. Fitzpatrick, B. 2004. Distributed caching with Memcached. *Linux Journal* 2004(124); http://www.linuxjournal.com/article/7451.
9. Dally, W., Towles, B. 2003. *Principles and Practices of Interconnection Networks*. San Francisco, CA: Morgan Kaufmann Publishers Inc.
10. Gill, P., Jain, N., Nagappan, N. 2011. Understanding network failures in data centers: measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference* (SIGCOMM '11): 350-361; http://doi.acm.org/10.1145/2018436.2018477.
11. Greenberg, A., Hamilton, J. R., Jain, N., Kandula, S., Kim, C., Lahiri, P., Maltz, D. A., Patel, P., Sengupta, S. 2009. VL2: a scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication* (SIGCOMM '09): 51-62; http://doi.acm.org/10.1145/1592568.1592576.
12. Greenberg, A., Hamilton, J., Maltz, D. A., Patel, P. 2008. The cost of a cloud: research problems in data center networks. *SIGCOMM Computer Communications Review* 39(1):68-73; http://doi.acm.org/10.1145/1496091.1496103.
13. Hoelzle, U., Barroso, L. A. 2009. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.). Morgan & Claypool Publishers.
14. Kermani, P., Kleinrock, L. 1976. Virtual cut-through: a new computer communication switching technique, *Computer Networks* 3(4):267-286; http://www.sciencedirect.com/science/article/pii/0376507579900321.

15. Leiserson, C. E. 1985. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers* 34(10):892-901.
16. Mori, T., Uchida, M., Kawahara, R., Pan, J., Goto, S. 2004. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement* (IMC '04): 115-120; http://doi.acm.org/10.1145/1028788.1028803.
17. Mudigonda, J., Yalagandula, P., Mogul, J., Stiekes, B., Pouffary, Y. 2011. NetLord: a scalable multi-tenant network architecture for virtualized datacenters. *SIGCOMM Computer Communication Review* 41(4):62-73; http://doi.acm.org/10.1145/2043164.2018444.
18. Mysore, R. N., Pamboris, A., Farrington, N., Huang, N., Miri, P., Radhakrishnan, S., Subramanya, V., Vahdat, A. 2009. PortLand: a scalable fault-tolerant layer 2 data center network fabric. *SIGCOMM Computer Communication Review* 39(4):39-50; http://doi.acm.org/10.1145/1594977.1592575.
19. Ni, L. M., McKinley, P. K. 1993. A survey of wormhole routing techniques in direct networks, *Computer* 26(2):62-76; http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=191995&isnumber=4947.
20. Ousterhout, J., Agrawal, P. Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., Stutsman, R. 2010. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review* 43(4):92-105; http://doi.acm.org/10.1145/1713254.1713276.
21. Protocol buffers; http://code.google.com/apis/protocolbuffers/.
22. Rumble, S. M., Ongaro, D., Stutsman, R., Rosenblum, M., Ousterhout, J. K. 2011. It's time for low latency. In *Proceedings of the 13th Usenix Conference on Hot Topics in Operating Systems* (HotOS13).
23. Vahdat, A., Al-Fares, M., Farrington, N., Mysore, R. N., Porter, G., Radhakrishnan, S. 2010. Scale-out networking in the data center. *IEEE Micro* 30(4):29-41; http://dx.doi.org/10.1109/MM.2010.72.
24. Vahdat, A., Liu, H., Zhao, X., Johnson, C. 2011. The emerging optical data center. Presented at the *Optical Fiber Communication Conference*. OSA Technical Digest (CD); http://www.opticsinfobase.org/abstract.cfm?URI=OFC-2011-OTuH2.
25. Wilson, C., Ballani, H., Karagiannis, T., Rowtron, A. 2011. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference* (SIGCOMM '11): 50-61; http://doi.acm.org/10.1145/2018436.2018443.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DENNIS ABTS is a member of the technical staff at Google, where he is involved in the architecture and design of next-generation large-scale clusters. Prior to joining Google, Abts was a senior principal engineer and system architect at Cray Inc. He has numerous technical publications and patents in the areas of interconnection networks, data-center networking, cache-coherence protocols, high-bandwidth memory systems, and supercomputing. He received his Ph.D. in computer science from the University of Minnesota - Twin Cities and is a member of ACM and the IEEE Computer Society.

BOB FELDERMAN spent time at both Princeton and UCLA before starting a short stint at Information Sciences Institute. He then helped found Myricom, which became a leader in cluster-computing

networking technology. After seven years there, he moved to Packet Design where he applied high-performance computing ideas to the IP and Ethernet space. He later was a founder of Precision I/O. All of that experience eventually led him to Google where he is a principal engineer working on issues in data-center networking and general platforms system architecture.