DOI:10.1145/3547137

Looking past inessential complexities to explain the Internet's simple yet daring design.

BY JAMES MCCAULEY, SCOTT SHENKER, AND GEORGE VARGHESE

Extracting the Essential Simplicity of the Internet

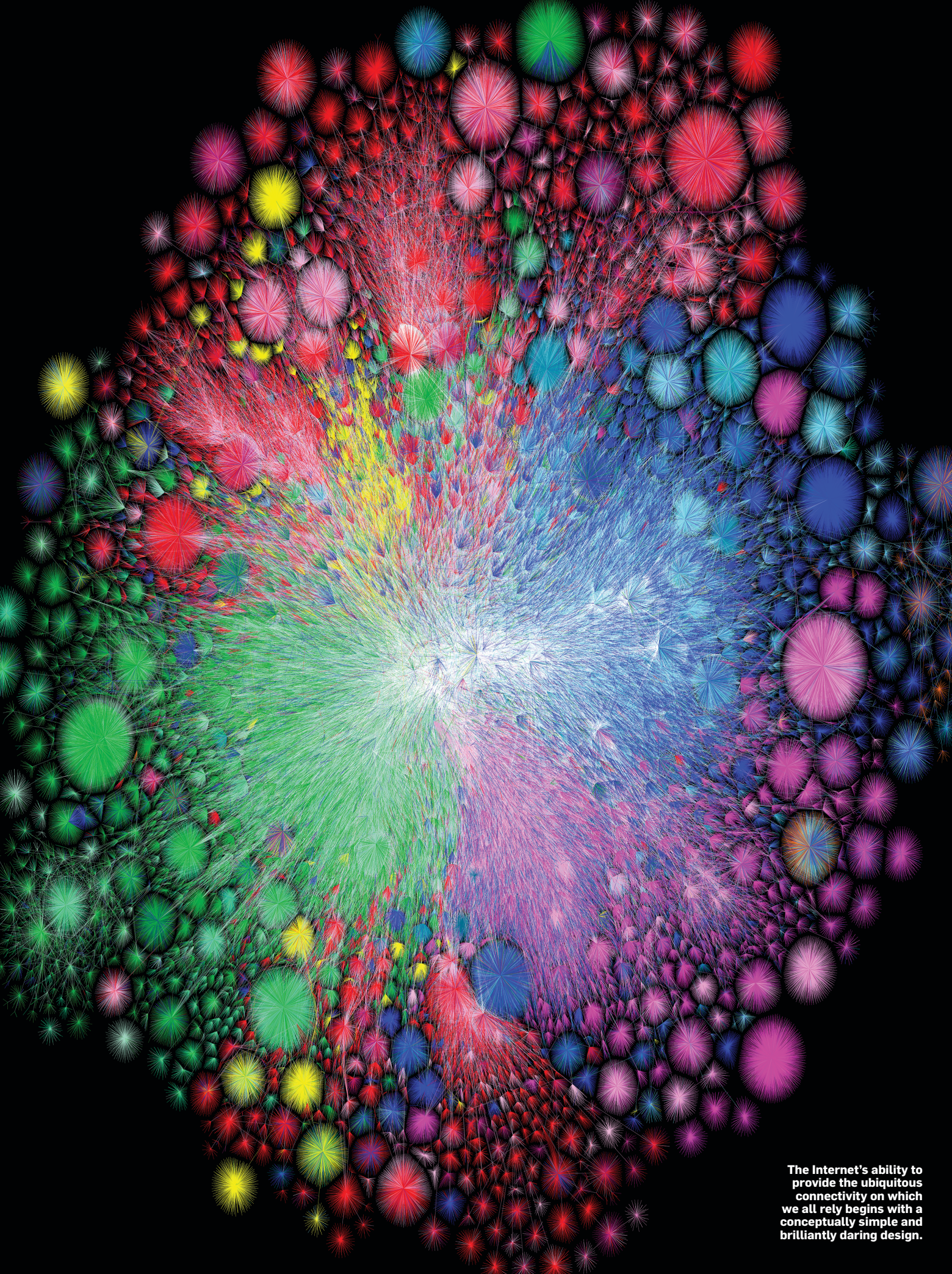
EVERYONE KNOWS THAT the Internet provides the ubiquitous connectivity on which we now rely, and many know that the underlying design was invented in the 1970s to allow computers to exchange data. However, far fewer understand the remarkable foresight that guided its design, which has remained essentially unchanged since its adoption in 1983 (see the sidebar "Brief Internet Timeline") while gracefully accommodating radical changes in applications, technologies, size, scope, and capacity. The Internet's design is underappreciated because its beauty is buried beneath an avalanche of implementation details. Most undergraduate networking courses teach students how the current Internet works, which requires they master an alphabet soup of protocols such as IP, TCP, BGP, OSPF, DVRP, DNS, STP, and ARP. These protocols are complex and riddled with details that are more historical than principled. As a result, much of the computer science community thinks that the Internet is merely a collection of complicated protocols, not a conceptually simple and brilliantly daring design.

This article attempts to extract the Internet's essential simplicity by motivating the fundamental Internet design choices from first principles without delving into all the ensuing complications. This allows us to focus on the *why* of the Internet rather than the *how*. Our first-principle analysis does not reflect the historical reasons for making these decisions (for such a discussion, see Cerf and Kahn[4] and Clark[6]) but instead justifies those decisions, as embodied in today's Internet (such as in the aforementioned protocols), with the full benefit of hindsight and from our own personal perspectives. While our purpose is primarily pedagogical, we also wrote this article as an homage to the pioneers[a] who shaped the Internet's design. Many of

a We do not have enough space to accurately and adequately credit the many people who helped create the Internet and guided its early evolution, but please see Leiner et al.[14] for more details on the who, what, and when of the Internet.

» key insights

- **This article's thesis is that the Internet can best be understood as a small set of design choices: a service model, a four-layer architecture, and three crucial mechanisms (routing, reliability, and resolution).**
- **This article focuses on the rationale behind each of these design choices rather than on the details of the Internet's many protocols. While this "why, not how" approach is not sufficient to understand the complexities of today's Internet, it does enable readers to design a new Internet with approximately the same properties.**
- **This approach also illuminates the Internet's brilliant and daring design, which contains lessons for all computer scientists.**



The Internet's ability to provide the ubiquitous connectivity on which we all rely begins with a conceptually simple and brilliantly daring design.

Brief Internet Timeline

1969: First exchange of data between remote computers. On October 29, 1969, a connection was established between UCLA and the Stanford Research Institute (SRI).

1972: Public demonstration of ARPANET. This was the first time the public was exposed to the promise of packet networks.

1974: Publication of the first paper describing the TCP/IP protocols.[4] These are the foundational protocols of today's Internet.

1982: DNS deployed, replacing the centralized distribution of a file that listed the IP address of every named host.

1983: ARPANET transitions to TCP/IP. On a global "flag-day" (January 1, 1983), all ARPANET systems finished converting to a four-layer architecture, which remains in use today.

1985: Spanning Tree Protocol (STP) invented, allowing Ethernets to safely interconnect by eliminating network loops.

1989: BGP design published. BGP is the interdomain routing protocol that today interconnects the Internet's many ASes.

1990s: The Internet becomes available to the public. The foundations of the Web were released in 1993, Yahoo was founded in 1994, and Akamai and Google were founded in 1998.

The Internet was established through the interconnection of several pioneering packet switched networks, key among them being ARPANET. Through a series of intermediate stages, including the development and integration of JANET in the U.K. and NSFNET in the U.S., these networks eventually became the public Internet. While the Internet infrastructure has changed greatly (in terms of speed, scope, and technologies), the basic technical design (in terms of its service model, architecture, and core protocol IP) has remained almost completely unchanged since 1983.

their basic design decisions were in direct conflict with the prevailing wisdom, arising from the telephone network, about how to build a global-scale communication infrastructure. Our attempt to extract simplicity in no way detracts from the brilliance of their achievement; on the contrary, we hope our presentation lets readers see that brilliance more clearly.

As background, we note that the Internet is a combination of network infrastructure (for example, links, switches, and routers) that delivers data between hosts, and host software (for example, applications along with networking support in libraries and operating systems) that processes that data. In this article, we start our search for simplicity by describing the delivery service supported by the network infrastructure, which we call the "service model." We then describe the Internet's "architecture," which is the modular decomposition of Internet functionality into four logical layers. This architecture requires three key mechanisms: **routing** (how to forward packets through the Internet so they reach their destination), **reliable delivery** (how to make data transfers resilient to packet losses), and **name resolution** (how to translate from names used by people and applications to addresses used by routing). After describing the designs of these three mechanisms, we then identify the key factors behind the Internet's success and end with a discussion the Internet's current weaknesses.

Service Model

All host software must rely on the infrastructure's delivery service (that is, its service model) when communicating with other hosts. Thus, the design of a service model requires a compromise between what would best suit host software and what the infrastructure can feasibly support. In designing a data-transfer service model, we must choose its unit of transmission and its performance assurances. Given the bursty nature of computer communications, a small unit of transmission is necessary to achieve efficient resource utilization. The Internet uses a collection of bits called a packet, which today is typically no larger than 1.5kb.

Internet applications have a wide variety of network performance requirements, from low latency (interactive video or closed-loop control) to high bandwidth (transferring large datasets) and reliable delivery (file transfers). A natural engineering instinct would be to support the union of these requirements with a network infrastructure that guarantees specific bounds on latency, bandwidth, and reliability. However, the Internet, as its name connotes, must interconnect a wide variety of local packet networks—for example, wireless and wired, shared access, and point-to-point—many of which cannot guarantee performance. The lowest common denominator of what these packet networks can support is "best-effort" packet delivery, where the network attempts to deliver all packets but makes no assurances about if, when, or at what rate packets will be delivered.

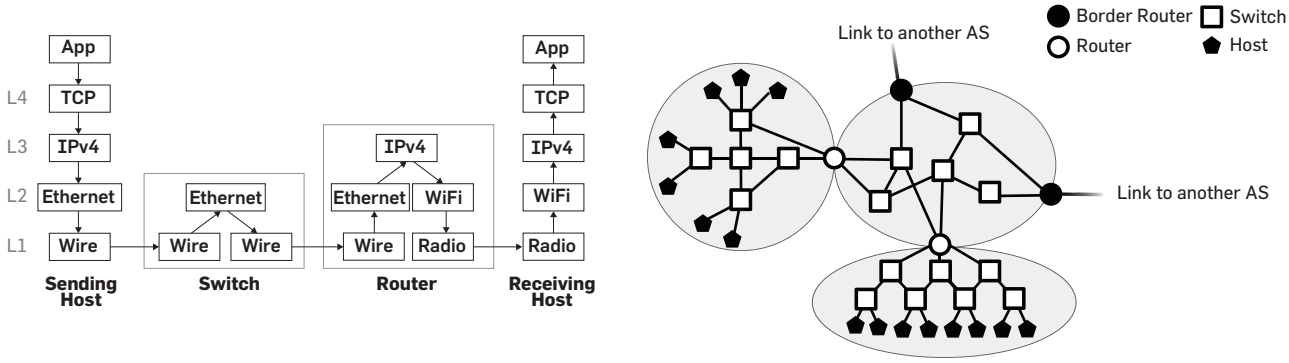
Thus, the fundamental question about the service model is whether it is better to be technologically ambitious and support all application requirements (but limit the possible packet networks that the Internet can incorporate) or accommodate all packet networks and provide only a modest, lowest-common-denominator, best-effort service model. As engineers, we naturally gravitate toward the intellectual challenge of ambitious technology goals. The Internet's designers instead opted for modesty, adopting the best-effort service model.

In hindsight, this was the superior choice for two reasons. First, the looser service model imposed minimal requirements on the infrastructure, enabling the Internet's extremely rapid growth. In short, being modest in performance requirements allowed the Internet to be ambitious in the speed and scope of its deployment. Second, the belief that applications have strict network requirements was a holdover from the telephone network, which had unintelligent end systems attached to a smart network that was fully responsible for meeting the strict performance requirements of telephony—that is, fixed-rate transmissions and reliable delivery. With increasingly powerful computers as the endpoints, Internet applications can be designed to adapt to various levels of performance, thereby loosening their requirements. Network operators can also ensure reasonable performance in most circumstances by providing sufficient bandwidth—that is, by deploying more and faster links so that the network is rarely congested.

Architecture

Architecture is about the organization of functionality not its implementation. Modularity is a guiding principle for architecture, calling for the decomposition of a system's goal into

Left: A packet's path through the layers in hosts, routers, and switches. Right: An AS with switches forwarding at L2 within networks (denoted by shaded ovals), routers forwarding between networks at L3, and border routers connecting to other ASes.



smaller tasks with clean interfaces. The Internet's goal—allowing applications on two different computers to communicate—can first be decomposed into two components: (i) what the network infrastructure does (best-effort packet delivery between hosts) and (ii) what the networking support software on the host does (making it easier for applications to use this best-effort delivery). We now discuss these two components and address four additional questions.

Network infrastructure. The infrastructure's job of host-to-host delivery can be decomposed into three different tasks arranged in layers, with the higher layers having broader geographic scope and the lower layers handling more local tasks. The process of delivering a packet from sending host to receiving host thus involves stitching together a set of these local delivery tasks. The most local and low-level task in providing best-effort packet delivery is the transmission of bits over links or broadcast media, which requires digital-to-analog conversion and error-correction, neither of which are specific to the Internet. This local-delivery task is handled by what is called the Physical Layer (L1).

Given L1's ability to send bits over links, the next task is to enable communication within a local packet network (hereafter, the term "network" will refer to locally scoped designs such as an Ethernet or a wireless network). This involves two steps: (i) forming packets out of sets of bits prepended with a packet header that describes where those bits should be delivered, and then (ii) delivering those packets to the appropriate destination within the local network. Limiting this second step to local (not global) delivery enables the use of non-scalable techniques such as *broadcast*, where the transmission medium itself ensures broadcast packets reach all hosts (as in wireless), and *flooding*, where the network ensures that flooded packets reach all hosts. This task is handled by what we call the Network Layer (traditionally called the Link Layer, but that terminology is no longer appropriate) or L2. In non-broadcast networks, this task is implemented by switches within the local network that forward packets to their destination.

The last task is to carry packets from the sender's network to the destination's network, leveraging L2's ability to deliver packets within these networks to individual hosts. This interconnection of networks is handled by the Internetworking Layer (L3) and is implemented by routers that are connected to two or more networks at L2 (by connected at L2, we mean that each of these networks can be reached from the router without going through another router). Packets are forwarded through a series of routers (with router-to-router delivery supported by L2 within the network to which both routers are connected) until they reach the destination network. This layer interconnects the networks, which is memorialized in the term "Internet." It also defines the Internet's service model, in that L3 delivers data to the higher-layer host software. See the figure above for a depiction of how a packet travels through these layers as it traverses the Internet.

Network support in host software. The network infrastructure's host-to-host service model does not specify to which host application the packets should be delivered, and its best-effort and packet-oriented nature can be hard for applications to use without additional help. To rectify these problems, host operating systems offer the Transport Layer (L4) in addition to other networking support. L4 delivers packets to individual applications on the host based on metadata in the packet header called a "port." To make best-effort service easier for applications to use, some common transport protocols offer three functions. The first is a byte-stream API so applications can write and read data using a simple file-like interface rather than having to explicitly send and receive individual packets. The second is controlling a host's rate of packet transmissions to prevent network overloads; this is called congestion control and involves a control loop, where transport protocols reduce their sending rate if they detect network congestion—for example, via dropped packets or increased delay. There are many congestion-control algorithms, differing in how they detect and respond to congestion, but the use of such control loops is conceptually (not practically) simple, so we do not delve into their details. The third, and most fundamental, is reliable packet delivery, so applications need not cope with packet losses. Such losses are a natural part of the network infrastructure's best-effort service model but are unacceptable to many applications. Although reliability, for the applications that require it, can be implemented in applications themselves or in other supporting li-

braries, for the sake of clarity, we focus on the very common and traditional arrangement, where reliability is implemented in L4.

Four additional questions. The four-layer Internet architecture—Physical Layer (L1), Network Layer (L2), Internetworking Layer (L3), and Transport Layer (L4)—is the natural result of modularity. Each layer only interacts with the layers directly above and below. Note that since packets always arrive via physical media at L1 (as shown in the figure), hosts must implement all four layers; routers implement the first three while switches only implement the first two. To complete our discussion, we must address four questions.

Question #1: What kind of diversity does this layered architecture enable and how is it managed? As long as two implementations of a layer offer the same upward and downward interfaces, they are architecturally interchangeable. In addition, the Internet architecture's modularity allows the coexistence of multiple transport protocols at L4, multiple local network designs at L2, and multiple physical technologies at L1, each offering their own distinct interfaces. The L1 and L2 technologies are chosen by the network provider, who can ensure that their interfaces are compatible—that is, that the local network design can use the set of link technologies. An application chooses its desired L4 protocol when it invokes the operating system's network API. Applications and network providers make their choices independently—that is, an application does not know about the network in which it currently resides, and the network provider does not know which applications will be used on its network. This works seamlessly if and only if there is a single set of interfaces at L3 that all L2 designs and L4 protocols are compatible with. Thus, we must have a single protocol at L3, which is the Internet Protocol (IP), currently IPv4. The next version, IPv6, is gaining popularity and has essentially the same interface as IPv4. IP functions as the "narrow waist" of the Internet architecture, and its singularity enables plurality (and thus innovation) at all other layers.

Question #2: What identifiers does the Internet use? The Internet must enable L2 and L3 packet headers to identify destinations for routing and allow users and applications to identify the services they want to access. This results in three kinds of names or addresses, which we now describe in turn. Typically, each Internet-reachable hardware network interface on a host (such as Wi-Fi, cellular, or Ethernet cards) has a permanent and unique address called a MAC address, which is used by L2 to identify destinations. L3 uses IP addresses, which specify a unique network within the Internet, and a specific host interface currently using the address within that network (such address assignments can change over time, but for clarity, we do not consider that detail here). Users and applications employ application-level names to refer to host-based services. To give some degree of permanence to such names, they are independent of whichever machine(s) the service is based on as well as where those hosts might be placed in the network. Of these three identifiers—application-level names, IP addresses, and MAC addresses—the first two must be resolved to the identifier at the next-lowest level. Thus, when an application on one host attempts to send a packet to an application on another host, it must resolve the application-level name into an IP address. When a packet arrives at a network, it is sent via L2 to the destination host or (as we explain below) to the next-hop router. In both cases, an IP address must be resolved into a MAC address. We describe the resolution of application-level names and the resolution of IP addresses later in this article.

Question #3: How is the Internet infrastructure organized? The Internet is not just an unstructured set of L2 networks interconnected by routers. Instead, it is a collection of Autonomous Systems (ASes), also called domains, each of which contains a set of L2 networks managed by a single entity. Examples of ASes include enterprises, universities, and commercial Internet Service Providers (ISPs). These ASes control their own internal L3 routing and must engage in a routing protocol with other ASes to provide host-to-host delivery between ASes. Thus, L3 involves two routing tasks: (i) routing between networks *within* an AS (intradomain routing), which is handled by routers in that AS, and (ii) routing *between* ASes (interdomain routing), which is handled by so-called border routers that connect two or more ASes. As we will discuss, these two routing tasks have different requirements and thus require different routing paradigms and protocols.

Question #4: How do all these pieces fit together? The process of delivering a packet across the Internet starts with an application resolving an application-level name to an IP address and then invoking a host's networking support to send data to that destination IP address. This results in a call to L4, which packetizes the data and invokes L3 to deliver those packets. At L3, a packet is forwarded through a sequence of routers until it reaches its destination network (as identified by the destination IP address in the packet header). Each router has a forwarding table that maps a destination IP address into the IP address of the next-hop router. Upon receipt of a packet, a router looks up the appropriate next-hop router in the forwarding table and then sends the packet to that next-hop node by calling L2. L2 must first resolve the IP address of the next-hop router into a MAC address and then deliver the packet to the next-hop router (either by broadcast or by forwarding the packet through a series of switches as explained in the next section), which then returns the packet to L3. The key technical challenge in this process is setting up the L3 forwarding tables so that the set of next-hops always results in the packet arriving at the appropriate destination.

Our discussion to this point has identified three nontrivial mechanisms—routing, reliability, and resolution—that are necessary to implement the Internet architecture. The following three sections discuss these in turn.

Routing

In this section, we use the term "routing" to refer to the generic problem of forwarding packets through the Internet to an intended destination host, whether that occurs at L3 and is implemented by routers, or at L2, where it is implemented by switches and thus is often called switching rather than routing. We start by considering L3 intradomain routing, where we assume: (i) each L3 packet header contains a destination IP address, (ii) each router has a

set of neighboring routers to which it is connected at L2, and (iii) each router has a forwarding table that correctly indicates whether the router is connected (at L2) to the packet's destination network and, if not, deterministically maps the destination of an arriving packet to the neighboring next-hop router the packet should be forwarded to. Before discussing how the forwarding table is established, we focus on the conditions under which a static set of forwarding tables successfully guides packets toward their destination. We refer to the set of interconnections between routers as the network graph, which we assume is connected; the set of forwarding tables as the forwarding state; and the union of these as the routing state. We say that a given instance of the routing state is *valid* if it always directs packets to their destination; we say it has a *loop* if there are cases (that is, a starting position and a destination address) where a packet can return to a router it already visited.

Result: *An instance of routing state is valid if and only if there are no loops.*

Argument: *Assume there are no loops; because the network is connected and finite, any packet must eventually hit a router that is connected (at L2) to its destination. Assume there is a loop for a given destination; any packet addressed to that destination which enters the loop will never reach the destination.*

Why do we care about this obvious result? Because the Internet uses a variety of routing algorithms to compute the forwarding state, and the key conceptual difference between these routing algorithms lies in the way they avoid loops in steady state. Routing protocols are complicated distributed systems that solve the problem of how to recompute the forwarding state quickly and automatically as the network graph changes due to network links failing and recovering. Here, we avoid the complexities of these distributed protocols and only focus on the forwarding state that results when the algorithm has converged to a steady state on a static network. The methods for avoiding loops in steady state depend crucially on what information the routers share with each other in these algorithms.

For example, a router knows its neighboring routers and can use a flooding algorithm to share this local information with every other router. In steady state, every router could use this information to assemble the entire network graph. If all routers use the same loop-free path-finding algorithm on this shared network graph to compute their forwarding tables, the resulting routing state is always valid. This approach is called "link-state" routing.

Much of the computer science community thinks that the Internet is merely a collection of complicated protocols, not a conceptually simple and brilliantly daring design.

Another case is when each router informs its neighboring routers of its own distance to all other networks, according to some metric, such as latency. Router α can compute its distance to each destination network n as $d_\alpha(n) = \min_\beta [d_\beta(n) + d(\alpha, \beta)]$, where $d_\beta(n)$ is the distance from each neighboring router β to network n, and $d(\alpha, \beta)$ is the distance between the two routers α, β. The steady state of this distributed computation produces shortest-path routes to each destination, which cannot have loops. This is called "distance-vector" routing.

When there is a change in the network graph, temporary loops in the routing state can arise during the process of recomputing routes (that is, when the protocol has not yet converged to the steady state) in both distance-vector and link-state routing. If left unchecked, packets cycling endlessly around such loops could cause severe network congestion. To prevent this, the IP protocol wisely incorporated a field in the packet header that starts with an initial number set by the sending host and which is then decremented every time a packet reaches a new router. If this field reaches zero, the packet is dropped, limiting the number of times packets can traverse an ephemeral loop and thereby ensuring that ephemeral loops are not catastrophic. Without this simple mechanism, all routing protocols would need to preclude ephemeral loops, and this degree of care would likely complicate the routing protocol.

We next consider L3 interdomain routing. ASes clearly must carry all packets for which they are the source or the destination; all other packets are considered *transit* traffic, passing through an AS on the way from the source AS to the destination AS. ASes want the freedom to choose both what transit traffic they carry and which

routes they use to forward their traffic toward its destination. For ISPs, these two policy choices depend heavily on their economic arrangements with their neighboring ASes, so they want to keep these choices private to avoid revealing information that would help their competitors.

While interdomain routing choices are in practice implemented by border routers, and this involves some complicated intradomain coordination between them, we can simplistically model interdomain routing with the collection of interconnected ASes forming a graph and the routing decisions being made by the ASes themselves. Interdomain routing must: (i) enable ASes to make arbitrary policy choices, so we cannot use distance-vector, and (ii) keep these policy choices private, so we cannot use link-state routing, which would require ASes to make their policies explicit so that every other AS could compute routes using those policies. An alternate approach is to allow ASes to implement their policies by choosing to whom they advertise their routes (by sending a message to a neighboring AS saying, "You can use my path to this destination") and choosing which routes they use when several of their neighbors have sent them routes to a given destination. These are the same messages and choices used in distance-vector routing, but distance-vector allows no policy flexibility: Routes are advertised to all neighbors, and only the shortest paths are chosen. This local freedom provides policy flexibility and privacy, but how do we prevent steady-state loops when routes are calculated, given that ASes can make arbitrary and independent choices about which routes they select and to which neighbors they offer their routes? The Internet's solution is to exchange path information. When AS "A" advertises a route (for a particular destination) to neighboring AS "B", it specifies the entire AS-level path of that traffic to the destination. If AS "B" sees it is already on that path, it does not use that route, as it would create a loop. If all ASes obey this obvious constraint, the steady state of what we call "path-vector" routing will be loop-free regardless of what policy choices ASes make. Path-vector routing is used in BGP, which is the current interdomain routing protocol; as such, BGP is the glue that holds the Internet's many ASes together.

Being modest in performance requirements allowed the Internet to be ambitious in the speed and scope of its deployment.

This path-vector approach ensures that there are no loops in any steady state. However, it does not ensure that the routing protocol converges to a steady state—for example, the policy oscillations first described in Varadhan et al.[17] are cases where path-vector algorithms do not converge. Nor does it ensure that all resulting steady states provide connectivity between all endpoints—for example, all ASes could refuse to provide transit connectivity to a given AS. Researchers were confused as to why these anomalies were not observed in the Internet, but the theoretical analysis in Gao and Rexford[9] (and subsequent work) showed that typical operational practices (where there is a natural hierarchy of service providers, lower-tier providers are customers of higher-tier providers, top-tier providers are connected in a full mesh, and routes are chosen to maximize revenue and minimize costs) produce routing policies that will always converge to steady states that provide end-to-end connectivity between all endpoints.

Avoiding loops also plays a role at L2 in non-broadcast networks, where flooding is often used to reach a destination. The spanning-tree protocol (STP) creates a tree out of the network (that is, eliminates all cycles from the network graph) by choosing to not use certain links. Once the network is a spanning tree, one can flood a packet to all hosts by having each switch forward the packet on all neighboring links that are part of the spanning tree, aside from the one by which the packet arrived. Such flooding allows hosts and routers to resolve IP addresses into MAC addresses by flooding an "address resolution protocol" (ARP) message that asks, "Which host or router has this IP address?"; the owner then responds with its MAC address. During this ARP exchange (and in fact whenever a host sends a packet), switches can learn how to reach a particular host without flooding by remembering the link on which they have most recently received a packet from that host. There is only one path between any two nodes on a spanning tree, so a host can be reached by sending packets over the link on which packets from that host

arrived. Thus, with such "learning switches," the act of resolving an IP address into a MAC address lays down the forwarding state between the sending and receiving hosts. When sending packets to a host where the MAC address has already been resolved, the network need not use flooding but can deliver the packets directly.

To summarize, all routing algorithms must have a mechanism to avoid loops in steady state (that is, after the protocol has converged), which in turn depends on what information is exchanged. To limit information exchanges to neighboring routers within ASes, one should use distance-vector, which results in shortest-paths to guarantee loop-freeness. To improve route flexibility in intradomain routing, a better choice is link-state, which requires flooding of neighbor information but allows an arbitrary, loop-free path calculation to be used. To allow ASes to exercise individual policy control in interdomain routing, they can exchange explicit path information to avoid loops. To enable flooding and learning routes on the fly in L2, it is useful to turn network graphs into spanning trees as they are inherently loop-free. There are other issues one might consider in routing, such as how to recover from failures without having to recompute routes (as in the use of failover routes[5]) and how one might use centralized control to simplify routing protocols (as in SDN[18]), but here our focus is on articulating the role of loop avoidance in commonly used routing paradigms.

Reliable Delivery

Our discussion of routing analyzed when routing state would enable the delivery of packets, but even with valid routing state, packets can be dropped due to overloaded links or malfunctioning routers. The Internet architecture does not guarantee reliability in the first three layers, but instead wisely (as argued in Saltzer et al.[15]) leaves this to the transport layer or applications themselves; dropped packets will only be delivered if they are retransmitted by the sending host.

Here we consider reliable delivery as ensured by a transport protocol that establishes connections which take data from one application and transmit it to a remote application. Several important transport protocols, such as the widely used TCP, provide the abstraction of a reliable byte-stream, where data in the byte-stream is broken into packets and delivered in order, and all packet losses are recovered by the transport protocol itself. The reliable byte-stream abstraction can be implemented entirely by host software that can distinguish packets in one byte-stream from packets in another, has sequence numbers on packets so they can be properly reordered before any data is handed to the receiving application, and retransmits packets until they have been successfully delivered.

Informally, a reliable transport protocol takes data from an application, transmits it in packets to the destination, and eventually either informs the application that the transfer has successfully completed or that it has terminally failed—and in both cases ceases further transmissions. For our discussion, we eliminate the possibility of announcing failure, since every transport protocol that eventually announces failure is by definition reliable. However, we assume that the underlying network eventually delivers a packet that has been repeatedly sent, so a persistent protocol can always succeed (more precisely, we assume that a packet that has been retransmitted infinitely often is delivered infinitely often). For this case, we ask: what communication is needed between the sender and receiver to ensure that the protocol can inform the application that it has succeeded if and only if all packets have been received? There are two common approaches: the receiver can send an acknowledgement (ACK) to the sender whenever it receives a packet, or it can send a non-acknowledgement (NACK) to the sender whenever it suspects that a packet has been lost.

Result: *ACKs are necessary and sufficient for reliable transport, while NACKs are neither necessary nor sufficient.*

Argument: *A reliable transport protocol can declare success only when it knows all packets have been delivered, and that can only be surmised by the receipt of an ACK for each of them. The absence of a NACK (which itself can be dropped) does not mean that a packet has been received.*

Note that NACKs can be useful, as they may provide more timely information about when the sender should retransmit. For example, TCP uses explicit ACKs for reliability and initiates retransmissions based on timeouts and implicit NACKs (when the expected ACK does not arrive).

Name Resolution

In addition to resolving IP addresses into MAC addresses via ARP, the Internet must also resolve application-level names into one or more IP addresses. These names are informally called hostnames and formally called fully qualified domain names. We use the nonstandard terminology application-level names because they neither refer to a specific physical machine (whereas MAC addresses do) nor do they directly relate to the notion of domains used in interdomain routing.

Any application-level naming system must: (i) assign administrative control over each name to a unique authority that decides which IP address(es) the name resolves to; (ii) handle a high rate of resolution requests; and (iii) provide both properties at a scale of billions of application-level names. To meet these challenges, the Internet adopted a hierarchical naming structure called the Domain Name System (DNS). The namespace is broken into regions called domains, which are recursively subdivided into smaller domains, and both resolution and administrative control are done hierarchically. Each naming domain has one or more nameservers that can create new subdomains and resolve names within its domain. Names can either be "fully qualified" (these resolve to one or more IP addresses) or not (their resolution points to one or more subdomain nameservers that can further resolve such names). This hierarchy begins with a set of top-level domains (TLDs) such as .cn, .fr, .ng, .br, .com, and .edu, which are controlled by a global authority (ICANN). Commercial registries allow customers to register subdomains under these TLDs. The resolution of TLDs to their nameservers is handled by a set of DNS root servers (whose addresses are known by all hosts), and the resolution proceeds

down the naming hierarchy from there. For example, www.myschool.edu is resolved first by a root server that points to a nameserver for .edu; the nameserver for .edu points to a nameserver for myschool.edu, which then resolves www.myschool.edu into an IP address. This hierarchical structure allows for highly parallel name resolution and fully decentralized administrative control, both of which are crucial for handling the scale of Internet naming.

Secrets to the Internet's Success

We have tried to reduce the Internet's impenetrable complexity into a small set of design choices: (i) a service model, (ii) a four-layer architecture, and (iii) three crucial mechanisms (routing, reliability, and resolution). Understanding the reasoning behind these decisions is not sufficient to understand the complexities of today's Internet but is sufficient to design a new Internet with roughly the same properties. This is consistent with our goal, which is to extract the Internet's essential simplicity. Unfortunately, this simplicity is not sufficient to explain the Internet's longevity, so we now ask: *Why has the Internet's design been so successful in coping with massive changes in speed, scale, scope, technologies, and usage?* We think there are four key reasons:

Modesty. Rather than attempt to meet all possible application requirements, the Internet adopted a very modest but quite general service model that makes no guarantees. This gamble on smart hosts and unintelligent networks: (i) allowed new applications to flourish, because the Internet had not been tailored to any specific application requirements; (ii) leveraged the ability of hosts to adapt in various ways to the vagaries of best-effort Internet service (for example, by rate adaptation, buffering, and retransmission); and (iii) enabled a rapid increase in network speeds, since the service model was relatively simple. If the Internet had adopted a more complicated service model, it might have limited itself to the requirements of applications that were present at the time of creation and employed designs that could have been implemented with then-available technologies. This would have led to a complicated design tailored to a narrow class of applications and quickly outmoded technologies, a recipe for short-term success but long-term failure.

Modularity. The modularity of the four-layer Internet architecture resulted in a clean division of responsibility: The network infrastructure (L1/L2/L3) supports better packet delivery (in terms of capacity, reach, and resilience) while applications (assisted by L4) create new functionality for users based on this packet-delivery service model. Thus, the architecture allowed two distinct ecosystems—network infrastructure and Internet applications—to flourish independently. However, the Internet's modularity goes beyond its formal architecture to a more general approach of maximizing autonomy within the confines of its standards-driven infrastructure, which is in sharp contrast to the more rigid uniformity of the telephone network. For instance, the only requirement on an AS is that its routers support IP and participate in the interdomain routing protocol BGP. Otherwise, each AS can deploy any L1 and L2 technologies and any intradomain routing protocol without coordinating with other ASes. Similarly, individual naming domains must support the DNS protocol but otherwise can adopt whatever name management policies and name resolution infrastructures they choose. This infrastructural autonomy allowed different operational practices to arise. For example, a university campus network, a hyperscaled datacenter network, and an ISP backbone network have very different operational requirements; the Internet's inherent autonomy allows them to meet their needs in their own way and evolve such methods over time.

Assuming failure is the normal case. As systems scale in size, it becomes increasingly likely that, at any point in time, some components of the system have failed.[10] Thus, while we typically think of scaling in terms of algorithmic complexity or state explosion, handling failures efficiently is also a key scalability requirement. In contrast to systems that expect normal operation and go into special modes to recover from failure, almost all Internet mechanisms treat failure as a common occurrence. For instance, in basic routing algorithms, the calculation of routes is typically done in the same way whether it is due to a link failure or a periodic recalculation. Similarly, when a packet is lost, it must be retransmitted, but such retransmissions are expected to occur frequently and are not a special case in transport protocols. This design style—of treating failures as the common case—was the foundation of Google's approach to building its hyperscaled infrastructure,[2] and we should remember that it was first pioneered in the Internet.

Rough consensus and running code. While this article has focused on the Internet's technical aspects, the process by which it was designed was also key to its success. Rather than adopt the formal design committees then current in the telephony world, the Internet inventors explicitly chose another path: Encourage smaller groups to build designs that work, and then choose, as a community, which designs to adopt. This was immortalized in a talk given by David Clark, one of the Internet's architectural leaders, who said, "We reject kings, presidents, and voting. We believe in rough consensus and running code." This egalitarian spirit extended to the shared vision of the Internet as a uniform and unifying communication platform connecting all users. Thus, the development of the Internet was indelibly shaped not only by purely technical decisions but also by the early Internet community's fervent belief in the value of a common platform connecting the world and their communal ownership of the design that would realize that vision.

Nothing Is Perfect

There are many areas where the Internet's design is suboptimal,[6,7] but most of them are low-level details that would not change our high-level presentation. In this section, we review three areas where the Internet's design has more fundamental problems.

Security. Many blame the poor state of Internet security on the fact that security was not a primary consideration in its design, though survivability in the face of failure was indeed an important consideration.[6] We think this critique is misguided for two reasons: (i) security in an interconnected world is a far more complicated and elusive goal than network security, and the Internet can only ensure the latter, and (ii) while

the Internet architecture does not inherently provide network security, there are protocols and techniques, some of which are in wide use, that can largely achieve this goal.

To be more precise, we say (similar to Ghodsi et al.[11]) that a network is secure if it can ensure the following properties when transferring data between two hosts: (i) connectivity between hosts is reasonably reliable (*availability*); (ii) the receiver can tell from which source the data came (*provenance*); (iii) the data has not been tampered with in transit (*integrity*); and (iv) the data has not been read by any intermediaries and no one snooping on a link can tell which hosts are exchanging packets (*privacy*). The latter three can be ensured by cryptographic protocols. The Internet's availability is vulnerable to distributed denial-of-service (DDoS) attacks, where many hosts (typically compromised hosts called bots) overload the target with traffic. There are techniques for filtering out this attacking traffic, but they are only partially effective. Availability is also threatened by BGP's vulnerability to attacks where ASes lie about their routes; cryptographic solutions are available, but they are not widely deployed. Thus, there are cryptographic protocols and mitigation methods that would make the Internet largely satisfy the definition of a secure network.

However, the failure to make the Internet an inherently secure network is most definitely not the primary cause of our current security issues, because a secure network does not prevent host software from being insecure. For example, if an application is designed to leak information about its users, or impersonate its users to execute unwanted financial trades, there is little the network can do to prevent such malicious actions. A host can be infected by a malicious application if it is unwittingly downloaded by a user or if some attack (such as a buffer overflow) turns a benign application into a malicious one. Comments about the Internet's woeful security typically refer to the ease of placing malicious software on hosts, but this is not primarily a network security problem.

In-network packet-processing. The early Internet designers argued[15] that the network infrastructure should generally focus on packet delivery and

Rather than adopt the formal design committees then current in the telephony world, the Internet inventors explicitly chose another path.

avoid higher-layer functionality, but this precept against what we call in-network packet-processing is currently violated in almost every operational network (see Blumenthal and Clark[3]). Modern networks have many middleboxes that provide more-than-delivery functions such as firewalling (controlling which packets can enter a network), WAN optimization (increasing the efficiency of networks by caching previous transfers), and load balancing (directing requests to underloaded hosts). A 2012 survey[16] revealed that roughly one-third of network elements are middleboxes, one-third are routers, and one-third are switches. Some of these in-network processing functions are deployed within a single enterprise to improve their internal efficiency; such violations only have impact within an enterprise network so are now widely viewed as an acceptable fact of life. In addition, as discussed below, several cloud and content providers (CCPs) have deployed large private networks that deploy in-network functions—particularly flow termination, caching, and load balancing—that lower the latency and increase the reliability seen by their clients.

Lack of evolvability. Since IP is implemented in every router, it is very difficult to change the Internet's service model. Even the transition to IPv6, which began in 1998 and has accelerated recently due to IPv4 address shortages, retains the same best-effort service model. The Internet has remained without significant architectural change for decades because its service has been "good enough"[12] and there was no viable alternative to the Internet. However, the large private networks deployed by the CCPs now offer superior service to their clients via their in-network processing, and most client traffic today either is served within its originating AS by a local CCP cache or goes directly from the source AS to a CCP's large private network with its specialized in-network processing.[1,18,13,19] The CCPs' large private networks are vertically integrated with cloud and content services that are now more dominant economic forces than ISPs. With the Internet unable to evolve to adopt similar functionality, the growth of

these private networks is likely to continue.

Thus, we might already be seeing the beginning of a transition to a new global infrastructure in which these private networks have replaced all but the last mile of the current Internet (that is, the access lines to residences and businesses) for almost all traffic. This would mark the end of the Internet's age of economic innocence, when it had no serious competitors and could settle for "good enough." While the Internet faces many challenges, we think that none are as important to its future as resolving the dichotomy of visions between the one that animated the early Internet community, which is that of a uniform Internet connecting all users and largely divorced from services offered at endpoints, and the one represented by the emerging large private CCP networks, in which the use of in-network processing provides better performance but only to clients of their cloud or content services.

Despite this gloomy prospect, we wrote this article to praise the Internet not bury it. The Internet is an engineering miracle, embodying design decisions that were remarkably prescient and daring. We should not let the complexities of today's artifact obscure the simplicity and brilliance of its core design, which contains lessons for us all. And we must not forget the intellectual courage, community spirit, and noble vision that led to its creation, which may be the Internet's most powerful lesson of all.

Acknowledgments

We thank our anonymous reviewers as well as our early readers—including Vint Cerf, David Clark, Deborah Estrin, Ethan Katz-Bassett, Arvind Krishnamurthy, Jennifer Rexford, Ion Stoica, and a dozen Berkeley graduate students who read the paper under duress—for their many helpful comments, but all remaining errors are entirely our responsibility. C

The Internet was indelibly shaped not only by purely technical decisions, but also by a fervent belief in the value of a common platform connecting the world.

References

1. Arnold, T. et al. Cloud provider connectivity in the flat Internet. In *Proceedings of the ACM Internet Measurement Conference*, Association for Computing Machinery (2020), 230–246; https://doi.org/10.1145/3419394.3423613.
2. Barroso, L.A. and Hoelzle, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines* (1st ed.), Morgan and Claypool Publishers (2009).
3. Blumenthal, M.S. and Clark, D.D. Rethinking the design of the Internet: The end-to-end arguments vs. the brave new world. *ACM Transactions on Internet Technology 1*, 1 (August 2001), 70–109; https://doi.org/10.1145/383034.383037.
4. Cerf, V.G. and Kahn, R.E. A protocol for packet network intercommunication. *IEEE Transactions on Communications 22*, 5 (1974), 637–648; https://doi.org/10.1109/TCOM.1974.1092259.
5. Chiesa, M. et al. A survey of fast-recovery mechanisms in packet-switched networks. *IEEE Communications Surveys Tutorials 23*, 2 (2021), 1253–1301; https://doi.org/10.1109/COMST.2021.3063980.
6. Clark, D. The design philosophy of the DARPA Internet protocols. In *ACM SIGCOMM Computer Communication Review 18*, 4 (August 1988), 106–114; https://doi.org/10.1145/52324.52336.
7. Clark, D.D. *Designing an Internet* (1st ed.), The MIT Press (2018).
8. Labovitz, C. Pandemic impact on global Internet traffic. North American Network Operators Group (2019); http://bit.ly/3YCXy5c.
9. Gao, L. and Rexford, J. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking 9*, 6 (2001), 681–692; https://doi.org/10.1109/90.974523.
10. Ghemawat, S., Gobioff, H., and Leung, S-T. The Google file system. In *Proceedings of the 19th ACM Symp. on Operating Systems Principles*, Association for Computing Machinery (2003), 29–43; https://doi.org/10.1145/945445.945450.
11. Ghodsi, A. et al. Naming in content-oriented architectures. In *Proceedings of the ACM SIGCOMM Workshop on Information-Centric Networking*, Association for Computing Machinery, (2011), 1–6; https://doi.org/10.1145/2018584.2018586.
12. Handley, M. Why the Internet only just works. *BT Technology Journal 24* (2006), 119–129.
13. Huston, G. The death of transit? *APNIC.net* (2016); https://blog.apnic.net/2016/10/28/the-death-of-transit/.
14. Leiner, B.M. et al. A brief history of the Internet. *Computer Communication Review 39*, 5 (2009), 22–31; https://doi.org/10.1145/1629607.1629613
15. Saltzer, J.H., Reed, D.P., and Clark, D.D. End-to-end arguments in system design. *ACM Transactions on Computer Systems 2*, 4 (November 1984), 277–288; https://doi.org/10.1145/357401.357402.
16. Sherry, J. et al. Making middleboxes someone else's problem: Network processing as a cloud service. *ACM SIGCOMM 2012 Conf.*, L. Eggert, J. Ott, V.N. Padmanabhan, and G. Varghese (Eds.), 13–24; https://doi.org/10.1145/2342356.2342359.
17. Varadhan, K., Govindan, R., and Estrin, D. Persistent route oscillations in inter-domain routing. *Computer Networks 32*, 1 (2000), 1–16; http://dblp.uni-trier.de/db/journals/cn/cn32.html#VaradhanGE00.
18. Wikipedia contributors. Software defined networking; https://en.wikipedia.org/wiki/Software-defined_networking.
19. Wohlfart, F. et al. Leveraging interconnections for performance: The serving infrastructure of a large CDN. In *Proceedings of the 2018 Conf. of the ACM Special Interest Group on Data Communication*. Association for Computing Machinery, 206–220; https://doi.org/10.1145/3230543.3230576.

James McCauley is an assistant professor at Mount Holyoke College, Computer Science, South Hadley, Massachusetts, USA.

Scott Shenker (shenker@icsi.berkeley.edu) is a professor at the University of California, Berkeley, EECS and a researcher at the International Computer Science Institute, Berkeley, California, USA.

George Varghese is the Jonathan B. Postel Professor of Computer Science at the University of California Los Angeles, USA.

Watch the authors discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/simplicity-of-the-internet