Network Working Group
Internet-Draft
Intended status: Best Current Practice
Expires: 15 February 2022

S. Cheshire
Apple Inc.

14 August 2021

The Internet is a Shared Network
draft-cheshire-internet-is-shared-00b

Abstract

(Unpublished Internet Draft) In the 1980s the designers of the Internet succeeded in creating a fast, efficient, inexpensive, shared network, that provided satisfactory fair sharing of capacity in a lightweight, decentralized fashion. One area that remains to be improved is to provide this fast, inexpensive, shared service with lower end-to-end round-trip delays for all traffic.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 15 February 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction

[This document is an unpublished initial version of an upcoming Internet Draft. This is currently just the personal opinions of the author. This document may contain errors which should be corrected. Feedback is welcomed.]

In the 1980s there was much debate between proponents of circuit switching and proponents of connectionless packet switching.

The circuit-switching proponents argued the benefits of reservations and guarantees. Providing these capabilities made the network slow and expensive (remember when international telephone calls cost several dollars per minute, and for most people there were no international video calls available at all) but it was argued that these capabilities were necessary.

In contrast, connectionless packet-switched networks like Ethernet provided much higher rates at lower costs, by declining any attempt to provide reservations and guarantees. Instead, software running over Ethernet had to be smart enough to adapt itself to the available rate dynamically, rather than expecting a guaranteed rate and then simply using that rate.

In summary, a circuit-switched network was a smart, expensive network, into which you could connect simple devices like a $10 telephone handset. A packet-switched network was a simple, cheap network, which relied on the end devices to be smart, which at the time meant high-end computer workstations costing tens of thousands

of dollars. Instead of building a smart network with simple devices, packet switching moved the intelligence to the edges, making a simple network with smart end-devices.

One often-overlooked aspect of reservations and guarantees is the fact that, necessarily, a reservation request may sometimes be denied. If the network has such extremely abundant capacity that it always has the ability to serve all traffic that wants to use it, then no reservation mechanism is needed, because the network will always work. The very existence of a reservation mechanism admits that sometimes there will not be enough capacity to serve every request, in which case some requests will be denied -- a telephone "busy signal".

When I first arrived at Stanford at the start of my Ph.D., I heard a story from Professor David Cheriton. Professor Cheriton had been out of California during the 1989 California earthquake, and when he learned of the earthquake he tried to call to see if his family was alive or dead. The California telephone system was so grossly overloaded that it was unable to handle any incoming calls at all, so all attempts to call California were met with a busy signal. For some time Professor Cheriton was unable to find out the fate of his family. As he commented later, the telephone system had only two modes of operation: It could give him a fixed 64kb/s, or nothing at all. Since it could not guarantee 64kb/s, it gave him nothing at all. Professor Cheriton wasn't wanting 64,000 bits per second of information, he was wanting one single bit of information, a service the telephone system was utterly unable to provide.

The strong lesson Professor Cheriton learned from this experience, and passed on to his students, was that a flexible network is better than a rigid one. Not every application needs a fixed 64kb/s data rate, and a given moment a network may not be able to provide that rate. A rigid network offers reservations and guarantees, but the Faustian bargain made there is that reservations and guarantees come with a dark side -- the inevitable refusals. A flexible best-effort network like the Internet doesn't offer reservations and guarantees, but it also doesn't give refusals. If a flexible best-effort network which never gives refusals sounds like a panacea, remember that it also comes with a kind of Faustian bargain of its own: by moving the intelligence out of the network and into the end devices, those devices take on the solemn responsibility to dutifully carry out the burden of being the intelligence of the network, taking care to adapt their sending rates to what the network is able to carry at that moment in time. When the end devices fail to perform those duties correctly the result is congestion collapse of the entire Internet, as happened several times during the 1980s until the TCP congestion control algorithms were refined sufficiently to work correctly.

By the 1990s it was clear that circuit-switching technologies like ISDN, ATM (Asynchronous Transfer Mode) to the desktop, and Wireless ATM were failures, and packet-switching technologies like the Internet, Ethernet, and Wi-Fi were succeeding in a big way.

There were moves to add reservations and guarantees to IP, the Internet Protocol, but none succeeded, and probably for good reason -- the lack of guarantees (and the corresponding refusals that necessarily go hand-in-hand with providing guarantees) are actually a benefit of the Internet, not a drawback.

A consequence of using a network where there are no reservations, guarantees, or service refusals is that all apps on such a connectionless packet-switched network like the Internet need to play their part in keeping the network operating smoothly by adapting their usage to the capacity they find available. When you click "send" on an email, the email program should transmit your email as fast as the network is able to carry it, but no faster. When you talk on a voice or video call, the app should scale its audio and video quality to match the available capacity. Sending at a rate lower than the available capacity results in an unnecessarily degraded audio or video experience. Sending at a rate higher than the available capacity results in the excess packets all being lost due to queue overflow at the bottleneck, which also generally results in an unnecessarily degraded audio or video experience, and also degrades other traffic sharing the network by causing excessive packet loss for those flows too. Of course there are limits to the scaling range of a typical voice or video app. If the available network capacity is multiple gigabits per second then a typical voice or video app will not use that because such an app is usually unable to make use of more than a few megabits per second. If the available network capacity is under one kilobit per second then the app is unlikely to be able to sustain any usable audio or video at all at that rate, and it will either have to terminate the call entirely or downgrade to text-based messaging.

2. Queueing Delays in the Internet

We have built this shared network, and the adaptive apps that make use of it. We achieved the satisfactory fair sharing of capacity by having large, simple FIFO (First-In First-Out) buffers (queues) at each hop in the network. When an app sends data at a rate faster than the bottleneck link can carry it, packets arrive in the bottleneck queue faster than they depart the queue, and the queue fills up. When a packet arrives when the queue is already full to capacity the packet is unavoidably lost, because there is no room to store it. The sender reacts to this loss by realizing that it is sending too fast, and reduces its rate. By having every device on the Internet work according to these rules, the Internet remains stable and adapts to varying traffic demands. This rate adaptation in response to congestion loss is generally not done directly by most network apps -- if they use a mature transport protocol like TCP [RFC7414] or QUIC [RFC9000], then this rate adaptation is done for them automatically. However, if an app embeds its own home-grown transport protocol running directly over UDP [RFC8085] with no other transport layer, then it is responsible for doing the necessary rate adaptation itself (see "UDP is not a transport protocol" below).

There are two consequences of this simple network buffering design.

The first consequence is that the buffers tend to be fairly large in order to smooth out changes in traffic rate without the output link going idle because the queue drained down to empty. As a result of these large buffers the time a packet spends waiting can be several times longer than the actual end-to-end signal transmission time. The speed of light in fibre, and the speed of electrical signals in copper wire, are both about 200 million metres per second. This means that an IP packet can travel 200 km (125 miles) in one millisecond, or 20,000 km (12,500 miles) in 100 ms. Were it not for oversized network buffers, an IP packet should be able to go coast-to-coast across the United States, and back again, in under 100 ms. But in reality today, in many cases a packet may spend several hundred milliseconds sitting in network buffers waiting to complete its journey. These oversized buffers significantly increase the time it takes to deliver a packet beyond what the speed-of-light limit would indicate.

The second consequence is that if the only way the network indicates that a device is sending too fast for the bottleneck link is by losing a packet, then the lost packet needs to be retransmitted. Typically that is done by the receiver sending acknowledgements (or negative acknowledgements) back to the sender, indicating the missing packet, which causes the sender to retransmit the lost packet. This adds an extra round-trip time before the desired data finally arrives at the receiver, thereby doubling the already inflated packet delivery time.

To reduce end-to-end round-trip delays on our packet-switched connectionless Internet, three things need to improved. Firstly, network bottlenecks should not allow large queues to become excessively full before finally deciding to signal the sender to slow down a little. Secondly, if the congestion signal comes sooner, the end-system rate reduction should be less drastic. And finally, the signal to slow down a little should not be resorting to drastic means like completely destroying a packet, but should be something a little less destructive.

In the original Internet design the only way to signal congestion was by having the bottleneck link lose packets due to queue overflow. There was a certain elegance in this: Since, for reliability, end systems had to deal with the occasional unexplained packet loss anyway, losing additional packets due to congestion was not fatal, because the end systems would retransmit the lost data and recover. Also, since congestion occurs when the network is busy, expecting routers to do more work in this situation might not be ideal, and simply losing the extra packets is arguably the "least work" solution to handle an overload situation. Since random unexplained packet loss was, and still is, relatively rare, and the dominant reason for loss is congestion and queue overflow, it is prudent for end systems to assume that any packet loss is probably an indicator of congestion. Thus end systems respond to packet loss both by reducing their sending rate and by retransmitting the lost data.

Thus packet loss is both a vital congestion signal (to keep the shared network stable), and an impairment (lost data needs to be retransmitted). A newer development, Explicit Congestion Notification [RFC3168] allows the network to communicate the vital congestion signal without the impairment caused by data loss.

3. Goals for Low-Delay Networking

A familiar problem that many people have experienced is that using the Internet sometimes feels "slow", despite Internet "speed tests" reporting that the connection can carry hundreds of megabits per second, or even gigabits per second. This slowness is obvious when it impacts real-time gaming and video conferencing, but it affect all applications. We are used to seeing simple network operations, like getting weather forecasts, stock quotes, or driving directions, all showing spinning animations while they wait for the network, even on multi-megabit or even gigabit connections.

Some efforts to improve responsiveness for real-time gaming and video conferencing have focused on prioritizing some traffic over other traffic, but these efforts are unlikely to be fruitful. One problem with traffic prioritization is that it implicitly assumes that traffic management is a zero-sum game. For some traffic to get more of the scarce bandwidth, some other traffic must get less. That requires making value judgements about which traffic is more deserving than other traffic. It is natural for engineers to assume that whatever they work on is the most important traffic. It is common to assume that voice and video traffic is automatically more important than any other traffic. But when the children in the house are having an eight-hour casual group video chat with their friends, and the parents are trying to access files to get their work done, is the video traffic really more important and more deserving than the file transfer traffic? The notion of prioritizing traffic is grounded in an assumption that scarcity of bandwidth is the root cause of the problem. This may have been a valid analysis in the era of 1Mb/s DSL connections, but when home users have gigabit connections and the problems remain, something else must be causing the problem.

If a network has abundant capacity, sufficient to reasonably serve all the traffic sharing it, and network operations still feel sluggish, then the problem is not too little capacity; it's too much queueing delay. One network connection could have 100Mb/s throughput with 5ms queueing delay, while another network connection could have 100Mb/s throughput with 500ms queueing delay. An Internet speed test would report the same throughput for both, but the user-experience while using those network connections -- the responsiveness of network applications -- would be very different.

Therefore, the solution we need to work on is not capacity allocation of a presumed scarce resource, it is reducing unnecessary queueing delays.

The main goal of this work is to minimize the end-to-end round-trip delay for all network traffic. The goal is to minimize the time between when a client issues a request to a server and when the client receives the corresponding reply in response, to as little as possible above the unavoidable speed-of-light-in-fibre delay.

A consequence of this goal is that per-flow queueing, while useful, is not itself an entire solution. Flow Queueing (FQ) seeks to isolate one flow from the potential bad behaviours of other flows sharing the same network link. The reason FQ alone is insufficient is that FQ assumes the "good" flow that deserves low delay is, just by chance, a source-limited flow with a very low-rate compared to the actual capacity on this network at this moment in time, so it will never build up queue, and the "bad" flow that is building up a queue is a capacity-seeking flow, so it doesn't deserve low delay.

Our goal should be to have an Internet where all flows can be capacity-seeking (to adapt to the network conditions, to provide the best user experience that the network can support at that time) and at the same time all flows also get low round-trip delays. An example of this is video streaming. A viewer usually wants their video streaming to provide the best visual quality that the video source and/or the network can provide, so the flow should be capacity-seeking. At the same time, if the viewer decides to skip to a different point in the video, they also want to spend the least time possible watching a "buffering... buffering... buffering..." indicator waiting for the new video segment to load, so we want the flow to experience low delay. This means that when a video streaming client issues a new "GET" request for a different segment of video, there should be the minimum amount of stale, now unwanted, old data sitting in its network queue. Thus the goal is not just to provide inter-flow protection (to isolate light flows with no queue from other flows that have filled their queue) but also to keep each individual queue short too.

To summarize: any given flow should be able to be capacity-seeking (to give best possible user experience -- best video quality, best map tile fidelity, etc.) without suffering excessive self-induced queueing delay as a result.

4. Explicit Congestion Notification Strategies

Classic ECN [RFC3168] treats a single packet marked CE (Congestion Experienced) as equivalent to a packet dropped due to queue overflow. This results in a drastic reduction by the sender in the number of packets in flight, historically by half, and more recently a reduction of 30%. This drastic reduction by the sender risks the bottleneck queue draining to empty, resulting in wasted network capacity, so network queues are typically quite large to avoid this.

These large oscillations in the number of packets in flight, and the large queues to accommodate these large oscillations, mean that queueing delays with AQM (Active Queue Management) and Classic ECN, while still lower than a simple tail-drop FIFO, are still higher than they could be.

Newer techniques, like L4S [L4S] and SCE [SCE], signal mild congestion sooner, expecting a more restrained reaction from the sender. This allows smaller oscillations in the number of packets in flight, which requires less queueing, which allows even lower delays.

L4S uses an input signal to the network, indicating that the sender and receiver understand L4S, and will interpret CE (Congestion Experienced) marks in the more moderate manner dictated by L4S.

SCE uses only an output signal from the network. SCE does not know whether the sender and receiver implement SCE or just Classic ECN. It marks packets with the SCE mark when the low queue threshold is reached, and it marks packets with the Classic CE mark when a higher queue threshold is reached. If the sender and receiver understand the SCE mark then they will respond to it; otherwise they will ignore it and respond when they start seeing CE marks.

5. Queue Management Strategy

Flow Queueing (FQ) implements a separate queue for each flow traversing that link. In routers close to the edge of the network, where there are a small number of flows and per-flow queues are feasible, this is useful, because if a particular flow behaves badly it only affects that flow's private queue. Deeper in the the core of the Internet, where there may be thousands or millions of flows sharing a link, it has been argued that perfect per-flow queueing is infeasible, so multiple flows will have to share a queue. When multiple flows share a queue it is more important that they all behave well.

Technologies like VPN also make flow segregation difficult, because many independent flows may share the same VPN tunnel, and part of the purpose of VPN is to obscure traffic details from outside observers. Similarly, newer transport protocols like QUIC [RFC9000] provide multiple independent streams on single QUIC connection, and use encryption to obscure the flow details from outside observers.

A system that offers perfect flow segregation into a separate queue per flow, can use SCE (Some Congestion Experienced -- an early indicator of light queue buildup beginning to occur). The per-flow private queue marks packets with SCE when light queue buildup begins to occur. If the receiver and/or sender don't implement SCE then the SCE marks are ignored. When a larger queue builds up, conventional CE is generated, and the sender responds by slowing down. Older senders and receivers that only support Classic ECN get reasonably low queueing delay with little-to-no packet loss. SCE-aware senders and receivers can get ultra-low queueing delay with little-to-no packet loss. The queue management algorithm doesn't need to know in advance whether the sender and receiver support just Classic ECN or SCE -- the queue management algorithm does both and the sender and receiver interpret the marks they understand. This makes SCE an output signal from the network. The sender and receiver use ECT to indicate they have some kind of ECN support, but they don't say which. The SCE algorithm provides both SCE and CE marks, and lets the sender and receiver decide which marks they want to respond to. If the sender and receiver only support Classic ECN they get a longer queue, but that doesn't matter because the bottleneck implements perfect flow segregation so the longer queue doesn't impact any other traffic.

In the core of the Internet where perfect per-flow queueing is infeasible, different flows share a queue. Putting all traffic in a single shared FIFO queue requires all traffic sharing that queue to play by the same rules. If a thousand flows occupying space in the same shared FIFO queue implement SCE, and one flow implements only

Classic ECN, then all the SCE flows will slow down, leaving the Classic ECN to fill the queue up to the Classic ECN CE marking threshold, resulting in low throughput and high delay for all the other flows. Consequently, when engineering constraints dictate that perfect flow segregation into a separate queue per flow is not feasible, and traffic needs to share a queue, the traffic needs to signal its willingness to "play by the rules" of ultra-low-delay queueing. L4S dual-queue assumes a signal that a flow is willing to "play by the rules" of ultra-low-delay queueing, and other traffic goes into the Classic ECN queue.

6. Queue Protection

Because of the vulnerability of an L4S-style shared ultra-low-delay queue to being disrupted by a single misbehaving flow, a queue protection function is required. If it were possible to perfectly track the behaviour of every individual flow separately then presumably it would be equally possible to queue every individual flow separately, so we have to assume that in this scenario that is not possible. Therefore queue protection, like road traffic policing, is a statistical operation. Some small percentage of flows are selected for monitoring, and if detected to be violating the rules of the ultra-low-delay queue, they are penalized. Road traffic police catch some drivers for infractions such as exceeding the speed limit or illegally parking in a handicapped parking space, but not every time. Thus the penalty for drivers who are caught violating the rules has to be a sufficient disincentive to discourage drivers from doing that. Similarly here, the penalty for being caught abusing the ultra-low-delay queue has to be a sufficient disincentive to discourage application developers from thinking they can get away with using the ultra-low-delay queue without responding when it generates congestion signals.

7. Network Equipment Vendor Choices

Network equipment vendors have a choice of implementing Classic ECN or something newer like L4S or SCE. Given that there are clients today that implement Classic ECN but there are no widely used clients that implement L4S or SCE, Classic ECN is attractive. Implementing Classic ECN offers an immediate benefit today, whereas L4S or SCE offer the promise of some benefit at an uncertain time in the future.

8. End-System Choices

Writers of apps on end systems have a choice of implementing Classic ECN or something newer like L4S or SCE. Given that there are network devices today that implement Classic ECN but there are no widely used network devices that implement L4S or SCE, implementing Classic ECN is attractive because if offers the potential of an immediate benefit.

Furthermore, there are almost always multiple hops on an Internet path, any of which could be the bottleneck link. In addition, the bottleneck link on a given Internet path can change repeatedly during the lifetime of a flow. Suppose a user has 100Mb/s cable modem Internet service. When they are close to their Wi-Fi access point, their Wi-Fi rate may be above 100Mb/s, so their bottleneck link is from the cable CMTS to their cable modem. The CMTS may implement L4S. If the user walks a little further from their Wi-Fi access point, so that their Wi-Fi rate drops below 100Mb/s, then their Wi-Fi access point becomes their bottleneck link. Their Wi-Fi access point may only implement Classic ECN. Therefore an application writer desiring low-delay networking would be wise to support both L4S and Classic ECN. When the bottleneck link is the CMTS, which supports L4S, that's great for ultra-low delay. When the user walks a few feet and the bottleneck link becomes the Wi-Fi access point, which only supports Classic ECN, that's still better than a simple tail-drop FIFO.

Thus an application writer would be wise to support both Classic ECN and one of the newer, even better, techniques, and use whichever is available at any given moment. One difficulty is how an end system knows which style of ECN is implemented on the current bottleneck link for its path, especially when the bottleneck can change from second to second during the lifetime of a communication.

9. Non-Queue-Building

A recent development is the concept of Non-Queue-Building flows, which can also benefit from ultra-low delay. A common example is DNS [RFC1034][RFC1035] traffic, which typically generates only a single query packet, and receives only a single response packet in reply, and getting that response as quickly as possible results in an improved user experience. When only one packet is in flight, it is not clear today how to apply meaningful congestion control to that single packet (though such technology could be developed in the future). Consequently, it has been proposed that such traffic should allowed to occupy the ultra-low delay queue without the responsibility of responding to congestion signals.

It is recommended that this concept by applied with extreme caution, and limited to traffic sources that generate at most one packet per round-trip time. Otherwise, there is a risk that many application developers will declare that their traffic fits this category, and therefore is entitled to occupy the ultra-low delay queue without the responsibility of responding to congestion signals.

For example, a DNS client that performs only one request at a time may be a legitimate case of Non-Queue-Building traffic, but a DNS client that is able to perform multiple queries concurrently would not qualify for the exemption from responding to congestion signals.

Similarly, the designer of a home automation controller using CoAP [RFC7252] to control devices may believe they can claim their software to be Non-Queue-Building and therefore they can ignore congestion control, because CoAP sends only a single packet and waits for a single reply packet. However, when their home automation controller executes an "all lights on" command to turn on 50 light bulbs, and then sends 50 concurrent CoAP requests into the network, that can induce significant queue overflow and packet loss in that network.

Consequently, we need to be very careful when deciding what traffic can legitimately be declared as being Non-Queue-Building in all possible scenarios, and therefore exempt from the responsibility of implementing competent congestion control to protect the shared packet-switched network.

10. UDP is not a Transport Protocol

UDP is not a Transport Protocol. It is just a datagram header format. Merely having "P" in the acronym is not enough to qualify as a protocol.

We use the term "protocol" in computer networking by analogy with diplomatic protocol. A protocol is a set of rules for behaviour. The rules state what should happen and when it should happen. Such rules may say something like, "If I do A then you should do B within time period C. If you fail to do that, then after time D has elapsed I will do E."

UDP just says where a packet came from, where it is going to, and nothing else. The source and destination ports in the UDP header are just like the source and destination addresses in the IP header. In fact, the source and destination ports should have been in the IP header from the start, and then UDP wouldn't have even been needed.

As described earlier, the insight of the Internet was to build a fast, cheap, simple network, instead of a slow, expensive, smart network. A consequence of this is that the smarts of the Internet lie in the end systems, not the network itself. To operate on the Internet, end systems have to meet that responsibility to be smart. If you use TCP or QUIC, that hard work is done for you. If you invent your own protocol running directly over UDP with no other transport layer, you have the responsibility to make it smart. UDP will not do any of the important work for you.

It is common for people to talk about packet loss as something that just happens on the Internet, and their home-grown protocol has to send more packets to "power through" the packet loss. The reality is the opposite. Packet loss is not something the Internet does to a transport protocol. Excessive packet loss is something a home-grown transport protocol inflicts on the Internet by sending too many packets and not paying attention to congestion signals (packet loss and/or explicit congestion marking) and implementing appropriate rate reductions (congestion control) in response.

SUN Microsystems used to have the advertising slogan, "The network is the computer". A more accurate slogan for the Internet might be, "The computer is the network". The crucial intelligence that keeps the Internet working smoothly exists in the transport protocols of the end systems that connect to it, not in the Internet itself. This is why it is so crucial that transport protocols are designed well.

A transport protocol handles:

1. Corruption (via checksums)

2. Reordering / Duplication (via sequence numbers)

3. Loss (via acknowledgements and retransmission)

4. Security: SYN flood and DDoS protections, encryption (separate layer, or combined like QUIC)

5. Flow control (for when receiver-limited rather than sender-limited) (receive window)

6. Most important: Congestion control (driven by loss or ECN) Congestion control has to cover a vast range of throughputs, from kb/s to Gb/s. A Transport Protocol has to infer the correct bottleneck rate, and transmit at that rate. A Transport Protocol has to respect overrun signals from network by slowing down.

Transport protocols like TCP and QUIC do all of the above. From that list, all UDP does is a modest corruption check (via a weak checksum) and none of the rest.

UDP also does little extra demultiplexing (the 16-bit ports) that arguably is a historical design mistake and should have been left in the IP header when TCP was split off from IP.

If the IP design had left the source and destination ports in the IP header along with the source and destination addresses, then there would be no reason for UDP to exist just as a separate shim to carry endpoint port identifiers. The Internet's "system datagram protocol" (i.e., IP) has an 8-bit protocol field -- but that is not big enough to fully demultiplex incoming packets to an individual software endpoint on the hardware device. Thus every layer running on top of IP has to reinvent its own 16-bit port space, just to identify which software endpoint is receiving this traffic. When you see duplication of functionality (16-bit ports) replicated across a whole set of modules at the same layer in a protocol stack (the transport layer), that's a pretty clear sign that this should be common functionality implemented in the layer below (the IP datagram layer).

In an alternate history where the IP header included not just source and destination addresses but also source and destination ports, there would be no need for UDP to exist at all, and all transport protocols would run directly over this expanded IP layer. With port numbers in IP, both TCP and QUIC would run directly over IP, instead of TCP (typically in the kernel) running over IP and QUIC (typically

in user space) running over UDP. There would be no need for this artificial distinction between the "system datagram protocol" and the "user datagram protocol". (As a historical note, AppleTalk did not have this design mistake. AppleTalk did not have a "system datagram protocol" and a "user datagram protocol" that were different, just a single AppleTalk datagram protocol that included source and destination ports saying where the packet was coming from and going to, and all transport protocols used that common AppleTalk datagram protocol.)

Aside: This lack of source and destination ports in the IP header also hurts IP multicast. IP multicast routing protocols pay attention to the IP multicast destination address, but not the IP multicast destination UDP port. This results in anomalous scenarios where a client is subscribed to a particular IP multicast address and listening on a particular UDP port, and the IP multicast routing infrastructure delivers packets to the host because the IP multicast destination address matches, only to then have the kernel discard all the packets because the UDP destination port in the packet does not match the port the client application is listening on. This problem would also not exist if, instead of being split across layers, the IP addresses and transport-layer ports were combined into a single header where the entire combined software endpoint address (what today we artificially divide into "address" and "port") were visible to the IP multicast routing infrastructure.

11. Conclusions

1. All Internet applications need to be adaptive (i.e., capacity-seeking). They should be able to scale up to give a better user experience when more network capacity is available, and scale down to continue working when less network capacity is available. If that adaptable style doesn't suit your application, then perhaps the Internet is not the right network for your application. You can either update your application to be adaptive, in the spirit of how a reservation-less network like the Internet is supposed to work, or use something like ISDN or ATM that can give you the guarantees you desire.

2. If you use TCP or QUIC, then you get this adaptive behaviour automatically. If you build your own home-grown transport protocol on top of UDP, you take on the responsibility of making it be a good Internet citizen.

3. Capacity-seeking apps deserve low delay too.

4. Classic ECN, and newer technologies that offer even lower delay, like L4S and SCE, will coexist for the foreseeable future, just like IPv4 and IPv6 will coexist for the foreseeable future. Any viable solution needs to accommodate this reality, and not assume a swift "transition" to whatever new technology is being imagined.

12. Security Considerations

Homegrown transport protocols often have security weaknesses. They often lack safeguards against misuse, allowing them to be recruited as an unwitting accomplice to conduct DDoS attacks. They often have unsophisticated congestion control, that can risk destabilizing the Internet if deployed on a large enough scale. Using mature, thoroughly-scrutinized protocols, like TLS 1.3 over TCP, or QUIC, reduces the risk of repeating these common mistakes.

13. IANA Considerations

This document has no IANA actions.

14. References

[RFC1034] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, DOI 10.17487/RFC1034, November 1987, <https://www.rfc-editor.org/info/rfc1034>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <https://www.rfc-editor.org/info/rfc3168>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <https://www.rfc-editor.org/info/rfc7414>.

[RFC8085] Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", BCP 145, RFC 8085, DOI 10.17487/RFC8085, March 2017, <https://www.rfc-editor.org/info/rfc8085>.

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/info/rfc9000>.

[L4S] Briscoe, B., Schepper, K. D., Bagnulo, M., and G. White, "Low Latency, Low Loss, Scalable Throughput (L4S) Internet Service: Architecture", Work in Progress, Internet-Draft, draft-ietf-tsvwg-l4s-arch-10, 1 July 2021, <https://tools.ietf.org/html/draft-ietf-tsvwg-l4s-arch-10>.

[SCE] Morton, J., Heist, P. G., and R. W. Grimes, "The Some Congestion Experienced ECN Codepoint", Work in Progress, Internet-Draft, draft-morton-tsvwg-sce-03, 17 May 2021, <https://tools.ietf.org/html/draft-morton-tsvwg-sce-03>.

Author's Address

Stuart Cheshire
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Phone: +1 (408) 996-1010
Email: cheshire@apple.com