computers are bad

You are receiving this facsimile because you signed up for fax delivery of this newsletter. To stop delivery, contact Computers Are Bad by email or fax.

https://computer.rip - me@computer.rip - fax: +1 (505) 926-5492

2022-01-16 peer to peer but mostly the main peer

I have admitted on HN to sometimes writing computer.rip posts which are extensions of my HN comments, and I will make that admission here as well. A discussion recently came up that relates to a topic I am extremely interested in: the fundamental loss of peer-to-peer capability on the internet and various efforts to implement peer-to-peer distributed systems on top of the internet.

Of course, as is usual, someone questioned my contention that there really is no such thing as a distributed system on the internet with the example of Bitcoin. Having once made the mistake of a graduate thesis on Bitcoin implementation details it is one of the things I feel most confident in complaining about, and I do so frequently. Rest assured that Bitcoin's technical implementation is 1) a heaping pile of trash which ought to immediately dispel stories about Nakamoto being some kind of engineering savant, and 2) Bitcoin is no exception to the fundamental truth that the internet does not permit distributed systems.

But before we get there we need to start with history... fortunate since history is the part I really care to write about.

Some time ago I mentioned that I had a half-written blog post that I would eventually finish. I still have it, although it's now more like 3/4 written. The topic of that post tangentially involves early computer networks such as ARPANET, BITNET, ALOHAnet, etc. that grew up in academic institutions and incubated the basic concepts around which computer networks are built today. One of my claims there is that ARPANET is overrated and had a smaller impact on the internet of today than everyone thinks (PLATO and BITNET were probably both more significant), but it is no exception to a general quality most of these early networks had that has been, well, part of the internet story: they were fundamentally peer to peer.

This differentiation isn't a minor one. Many early commercial computer networks were extensions of timeshare multiple access systems. That is, they had a strictly client-server (or we could actually call it terminal-computer) architecture in which service points and service clients were completely separated. Two clients were never expected to communicate with each other directly, and the architecture of the network often did not facilitate this kind of use.

Another major type of network which predated computer networks and had influence on them were telegraph networks. Telegraph systems had long employed some type of "routing," and you can make a strong argument that the very concept of packet switching originated in telegraph systems. We tend to think of telegraph systems as being manual, morse-code based networks where routing decisions and the actual message transfer were conducted by men wearing green visors. By the 1960s, though, telegraph

networks were gaining full automation. Messages were sent in baudot (a 5-bit alphabetical encoding) with standardized headers and trailers that allowed electromechanical equipment and, later, computers to route them from link to link automatically. The resemblance to packet switched computer networks is very strong, and by most reasonable definitions you could say that the first wide-scale packet network in the US was that of Western Union [1].

Still, these telegraph networks continued to have a major structural difference from what we now consider computer networks. Architecturally they were "inverted" from how we think of hardware distribution on computer networks: they relied on simple, low-cost, low-capability hardware at the customer site, and large, complex routing equipment within the network. In other words, the "brains of the operation" were located in the routing points, not in the users equipment. This was useful for operators like Western Union in that it reduced the cost of onboarding users and let them perform most of the maintenance, configuration, upgrades etc. at their central locations. It was not so great for computer systems, where it was desirable for the computers to have a high degree of control over network behavior and to minimize the cost of interconnecting computers given that the computers were typically already there.

So there are two significant ways in which early computer networks proper were differentiated from time-sharing and telegraph networks, and I put both of them under the label of "network of equals," a business term that is loosely equivalent to "peer to peer" but more to our point. First, a computer network allows any node to communicate with any other node. There is no strict definition of a "client" or "server" and the operation of the network does not make any such assumptions as to the role of a given node. Second, a computer network places complexity at the edge. Each node is expected to have the ability to make its own decisions about routing, prioritization, etc. In exchange, the "interior" equipment of the network is relatively simple and does not restrict or dictate the behavior of nodes.

A major manifestation of this latter idea is distributed routing. Most earlier networks had their routes managed centrally. In the phone and telegraph networks, the maintenance of routing tables was considered part of "traffic engineering," an active process performed by humans in a network operations center. In order to increase flexibility, computer networks often found it more desirable to generate routing tables automatically based on exchange of information between peers. This helped to solidify the "network of equals" concept by eliminating the need for a central NOC with significant control of network operations, instead deferring routing issues to the prudence of the individual system operators.

Both of these qualities make a great deal of sense in the context of computer networking having been pioneered by the military during the Cold War. Throughout the early days of both AUTODIN (the military automated telegraph network) and then ARPANET which was in some ways directly based on it, there was a general atmosphere that survivability was an important characteristic of networks. This could be presented specifically as survivability in nuclear war (which we know was a key goal of basically all military communications projects at the time), but it has had enduring value outside of the Cold War context as we now view a distributed, self-healing architecture as being one of the great innovations of packet-switched computer networks. The fact that this is also precisely a military goal for survival of nuclear C2 may have more or less directly influenced ARPANET depending on who you ask, but I think it's clear that it was at least some of the background that informed many ARPANET design decisions.

It might help illustrate these ideas to briefly consider the technical architecture of ARPANET, which while a somewhat direct precursor to the modern internet is both very similar and very different. Computers did not connect "directly" to ARPANET because at the time it was unclear what such a "direct" connection would even look like. Instead, ARPANET participant computers were connected via serial line to a dedicated computer called an interface message processor, or IMP. IMPs are somewhat tricky to map directly to modern concepts, but you could say that they were network interface controllers, modems, and routers all in one. Each IMP performed the line coding to actually transmit messages over leased telephone lines, but also conducted a simple distributed routing algorithm to determine *which* leased telephone line a message should be sent over. The routing functionality is hard to describe because it evolved rapidly over the lifetime of ARPANET, but it was, by intention, both simple and somewhat hidden. Any computer could send a message to any other computer, using its numeric address, by transferring that message to an IMP. The IMPs performed some internal work to route the message but this was of little interest to the computers. The IMPs only performed enough work to route the message and ensure reliable delivery, they did not do anything further and certainly nothing related to application-level logic.

Later, ARPANET equipment contractor BBN, along with the greater ARPANET project, would begin to openly specify "internal" protocols such as for routing in order to allow the use of non-BBN IMPs. Consequentially, much of the functionality of the IMP would be moved into the host computers themselves, while the routing functionality would be moved into dedicated routing appliances. This remains more or less the general idea of the internet today.

Let's take these observations about ARPANET (and many, but not all, other early computer networks) and apply them to the internet today.

Peer-to-Peer Connectivity

The basic assumption that any computer could connect to any other computer at will proved problematic by, let's say, 1971, when an employee of BBN created something we might now call a computer worm as a proof of concept. The problem was far greater as minicomputers and reduced connectivity costs significantly increased the number of hosts on the internet, and by the end of the '90s network-based computer worms had become routine. Most software and the protocols it used were simply not designed to operate in an adversarial climate. But, with the internet now so readily accessible, that's what it became. Computers frequently exposed functionality to the network that was hazardous when made readily available to anonymous others, with Windows SMB being a frequent concern [2].

The scourge of internet-replicating computer worms was stopped mostly not by enhancements in host security but instead as an incidental effect of the architecture of home internet service. Residential ISPs had operated on the assumption that they provided connectivity to a single device, typically using PPP over some type of telephone connection. As it became more common for home networks to incorporate multiple devices (especially after the introduction of WiFi) residential ISPs did not keep pace. While it was easier to get a subnet allocation from a residential ISP back then than it is now, it was very rare, and so pretty much all home networks employed NAT as a method to make the entire home network look, to the internet, like a single device. This remains the norm today.

NAT, as a side effect of its operation, prohibits all inbound connections not associated with an existing outbound one.

NAT was not the first introduction of this concept. Already by the time residential internet was converging on the "WiFi router" concept, firewalls had become common in institutional networks. These early firewalls were generally stateless and so relatively limited, and a common configuration paradigm (to this day) was to block all traffic that did not match a restricted set of patterns for expected use.

Somewhere along this series of incremental steps, a major change emerged... not by intention so much as by the simple accretion of additional network controls.

The internet was no longer peer to peer.

Today, the assumption that a given internet host can connect to another internet host is one where exceptions are more common than not. The majority of "end user" hosts are behind NAT and thus cannot accept inbound connections. Even most servers are behind some form of network policy that prevents them accepting connections that do not match an externally defined list. All of this has benefits, there is an upside, but there is also a very real downside, which is that the internet has effectively degraded to a traditional client-server architecture.

One of the issues that clearly illustrated this to many of my generation was multiplayer video games. Most network multiplayer games of the '90s to the early '00s were built on the assumption that one player would "host" a game and other players would connect to them. Of course as WiFi routers and broadband internet became common, this stopped working without extra manual configuration of the NAT appliance. Similar problems were encountered by youths in peer-to-peer systems like BitTorrent and, perhaps this will date me more than anything else, eD2k.

But the issue was not limited to such frivolous applications. Many early internet protocols were designed with the peer-to-peer architecture of the internet baked into the design. FTP is a major example we encounter today, which was originally designed under the assumption that the server could open a connection to the client. RTMP and its entire family of protocols, including SIP which remains quite relevant today, suffer from the same basic problem.

For any of these use-cases to work today, we have had to clumsily re-invent the ability for arbitrary hosts to connect to each other. WebRTC, for example, is based on RTMP and addresses these problems for the web by relying on STUN and TURN, two separate but related approaches to "NAT negotiation." Essentially every two-way real-time media application must take a similar approach, which is particularly unfortunate since TURN introduces appreciable overhead as well as privacy and security implications.

Complexity at the Edge

This is a far looser argument, but I think one that is nonetheless easy to make confidently: the concept of complexity at the edge has largely been abandoned. This happens more at the application layer than at lower layers, since the lower layers ossified decades ago. But almost all software has converged on the web platform, and the web platform is inherently client-server. Trends such as SPAs have somewhat reduced the magnitude of the situation as even in web browsers some behavior can

happen on the client-side, but a look at the larger ecosystem of commercial software will show you that there is approximately zero interest in doing anything substantial on an end-user device. The modern approach to software architecture is to place all state and business logic "in the cloud."

Like the shift away from P2P, this has benefits but also has decided disadvantages. Moreover, it has been normalized to the extent that traditional desktop development methods that were amenable to significant complexity at the client appear to be atrophying on major platforms.

As the web platform evolves we may regain some of the performance, flexibility, and robustness associated with complexity at the edge, but I'm far from optimistic.

Peer-to-Peer and Distributed Applications Today

My contention that the internet is not P2P might be surprising to many as there is certainly a bevy of P2P applications and protocols. Indeed, one of the wonders of software is that with sufficient effort it is possible to build a real-time media application on top of a best-effort packet-switched network... the collective ingenuity of the software industry is great at overcoming the limitations of the underlying system.

And yet, the fundamentally client-server nature of the modern internet cannot be fully overcome. P2P systems rely on finding some mechanism to create and maintain open connections between participants.

Early P2P systems such as BitTorrent (and most other file sharing systems) relied mostly on partial centralization and user effort. In the case of BitTorrent, for example, trackers are centralized services which maintain a database of available peers. BitTorrent should thus be viewed as a partially centralized system, or perhaps better as a distributed system with centralized metadata (this is an extremely common design in practice, and in fact the entire internet could be described this way if you felt like it). Further BitTorrent assumes that the inbound connection problem will be somehow solved by the user, e.g. by configuring appropriate port forwarding or using a local network that supports automated mechanisms such as UPnP.

Many P2P systems in use today have some sort of centralized directory or metadata service that is used for peer discovery, as well as configuration requirements for the user. But more recent advances in distributed methods have somewhat alleviated the needs for this type of centralization. Methods such as distributed hole punching (both parties initiating connections to each other at the same time to result in appropriate conntrack entries at the NAT devices) allow two arbitrary hosts to connect, but require that there be some type of existing communications channel to facilitate that connection.

Distributed hash tables such as the Kademlia DHT are a well understood method for a distributed system to share peer information, and indeed BitTorrent has had it bolted on as an enhancement while many newer P2P systems rely on a DHT as their primary mechanism of peer discovery. But for all of these there is a bootstrapping problem: once connected to other DHT peers you can use the DHT to obtain additional peers. But, this assumes that you are aware of at least a single DHT peer to begin with. How do we get to that point?

You could conceivably search the entire internet space, but given the size of the internet that's infeasible. The next approach you might reach for is some kind of broadcast or multicast, but for long-standing abuse, security, and scalability reasons broadcast and multicast cannot be routed across the internet. Anycast offers similar potential and is feasible on the internet, but it requires the cooperation of an AS owner which would be both centralized and require a larger up-front investment than most P2P projects are interested in.

Instead, real P2P systems address this issue by relying on a centralized service for initial peer discovery. There are various terms for this that are inconsistent between P2P protocols and include introducer, seed, bootstrap node, peer helper, etc. For consistency, I will use the term introducer, because I think it effectively describes the concept: an introducer is a centralized (or at least semi-centralized) service that introduces new peers to enough other peers that they can proceed from that point via distributed methods.

As a useful case study, let's examine Bitcoin... and we have come full circle back to the introduction, finally. Bitcoin prefers the term "seeding" to refer to this initial introduction process. Dating back to the initial Nakamoto Bitcoin codebase, the Bitcoin introduction mechanism was via IRC. New Bitcoin nodes connected to a hardcoded IRC server and joined a harcoded channel, where they announced their presence and listened for other announcements. Because IRC is a very simple and widely implemented message bus, this use of IRC had been very common. Ironically it was its popularity that lead to its downfall: IRC-based C2 was extremely common in early botnets, at such a level that it has become common for corporate networks to block or at least alert on IRC connections, as the majority of IRC connections out of many corporate networks are actually malware checking for instructions.

As a result, and for better scalability, the Bitcoin project changed the introduction mechanism to DNS, which is more common for modern P2P protocols. The Bitcoin Core codebase includes a hardcoded list of around a dozen DNS names. When a node starts up, it queries a few of these names and receives a list of A records that represent known-good, known-accessible Bitcoin nodes. The method by which these lists are curated is up to the operators of the DNS seeds, and it seems that some are automated while some are hand-curated. The details don't really matter that much, as long as it's a list that contains a few contactable peers so that peer discovery can continue from there using the actual Bitcoin protocol.

Other projects use similar methods. One of the more interesting and sophisticated distributed protocols right now is Hypercore, which is the basis of the Beaker Browser, in my opinion the most promising distributed web project around... at least in that it presents a vision of a distributed web that is so far not conjoined at the hip with Ethereum-driven hypercapitalism. Let's take a look at how Hypercore and its underlying P2P communications protocol Hyperswarm address the problem.

Well, it's basically the exact same way as Bitcoin with one less step of indirection. When new Hypercore nodes start up, they connect to bootstrap1.hyperdht.org through bootstrap3.hyperdht.org, each of which represents one well-established Hypercore node that can be used to get a toehold into the broader DHT.

This pattern is quite general. The majority of modern P2P systems bootstrap by using DNS to look up either a centrally maintained node, or a centrally maintained list of nodes. Depending on the project these introducer DNS entries may be fully centralized and run by the project, or may be "lightly decentralized" in that there is a list of several operated by independent people (as in the case of Bitcoin). While this is

slightly less centralized it is only slightly so, and does not constitute any kind of real distributed system.

Part of the motivation for Bitcoin to have multiple independently operated DNS seeds is that they are somewhat integrity sensitive. Normally the Bitcoin network cannot enter a “split-brain” state (e.g. two independent and equally valid blockchains) because there are a large number of nodes which are strongly interconnected, preventing any substantial number of Bitcoin nodes being unaware of blocks that other nodes are aware of. In actuality Bitcoin enters a “split-brain” state on a regular basis (it’s guaranteed to happen by the stochastic proof of work mechanism), but as long as nodes are aware of all “valid” blockchain heads they have an agreed upon convention to select a single head as valid. This method can sometimes take multiple rounds to converge, which is why Bitcoin transactions (and broadly speaking other blockchain entries) are not considered valid until multiple “confirmations”--this simply provides an artificial delay to minimize the probability of a transaction being taken as valid when the Bitcoin blockchain selection algorithm has not yet converged across the network.

But this is only true of nodes which are already participating. When a new Bitcoin node starts for the first time, it has no way to discover any other nodes besides the DNS seeds. In theory, if the DNS seeds were malicious, they could provide a list of nodes which were complicit in an attack by intentionally not forwarding any information about some blocks or advertisements of nodes which are aware of those blocks. In other words, in practice the cost of a sybil attack is actually reduced to the number of nodes directly advertised by the DNS seeds, but only for new users and only if the DNS seeds are complicit. In practice the former is a massive limitation. The Bitcoin project allows only trusted individuals, but also multiple such individuals, to operate DNS seeds in order to mitigate the latter. In practice the risk is quite low, mostly due to the limited impact of the attack rather than its difficulty level (very few people are confirming Bitcoin transactions using a node which was just recently started for the first time).

Multicast

One of the painful points here is that multicast and IGMP make this problem relatively easy on local networks, and indeed mDNS/Avahi/Bonjour/etc solve this problem on a daily basis, in a reasonably elegant and reliable way, to enable things like automatic discovery of printers. Unfortunately we cannot use these techniques across the internet because, among other reasons, IGMP does not manageably scale to internet levels.

P2P systems can use them across local networks, though, and there are P2P systems (and even non-P2P systems) which use multicast methods to opportunistically discover peers on the same local network. When this works, it can potentially eliminate the need for any centralized introducer. It’s, well, not that likely to work... that would require at least one, preferably more than one, fully established peer on the same local network. Still, it’s worth a shot, and Hypercore for example does implement opportunistic peer discovery via mDNS zeroconf.

Multicast presents an interesting possibility: much like the PGP key parties of yore, a P2P system can be bootstrapped without dependence on any central service if its users join the same local network at some point. For sufficiently widely-used P2P systems, going to a coffee shop with a stable, working node once in order to collect

initial peer information will likely be sufficient to remain a member of the system into the future (as long as there are enough long-running peers with stable addresses that you can still find some and use them to discover new peers weeks and months into the future).

Of course by that point we could just as well say that an alternative method to bootstrapping is to call your friends on the phone and ask them for lists of good IP addresses. Still, I like my idea for its cypherpunk aesthetics, and when I inevitably leave my career to open a dive bar I'll be sure to integrate it.

Hope for the future

We have seen that all P2P systems that operate over the internet have, somewhere deep down inside, a little bit of centralized original sin. It's not a consequence of the architecture of the internet so much as it's a consequence of the fifty years of halting change that has brought the internet to its contemporary shape... changes that were focused around the client-server use cases that drive commercial computing for various reasons, and so had the network shaped in their image to the extent of exclusion of true P2P approaches.

Being who I am it is extremely tempting to blame the whole affair on capitalism, but of course that's not quite fair. There are other reasons as well, namely that the security, abuse, stability, scalability, etc. properties of truly distributed systems are universally more complex than centralized ones. Supporting fully distributed internet use cases is more difficult, so it's received lower priority. The plurality of relatively new P2P/distributed systems around today shows that there is some motivation to change this state of affairs, but that's mostly been by working around internet limitations rather than fixing them.

Fixing those limitations is difficult and expensive, and despite the number of P2P systems the paucity of people who actually use them in a P2P fashion would seem to suggest that the level of effort and cost is not justifiable to the internet industry. The story of P2P networking ends where so many stories about computing end: we got here mostly by accident, but it's hard to change now so we're going to stick with it.

I've got a list of good IP addresses for you though if you need it.

[1] WU's digital automatic routing network was developed in partnership with RCA and made significant use of microwave links as it expanded. Many have discussed the way that the US landscape is littered in AT&T microwave relay stations, fewer know that many of the '60s-'80s era microwave relays not built by AT&T actually belonged to Western Union for what we would now call data networking. The WU network was nowhere near as extensive as AT&T's but was particularly interesting due to the wide variety of use-cases it served, which ranged from competitive long distance phone circuits to a a very modern looking digital computer interconnect service.

[2] We should not get the impression that any of these problems are in any way specific to Windows. Many of the earliest computer worms targeted UNIX systems which were, at the time, more common. UNIX systems were in some ways more vulnerable due to their relatively larger inventory of network services available, basically all of which were designed with no thought towards security. Malware developers tended to follow the market.