Charting the Temporal Topology: Enumerating the Global NTP Server Network

Rein Fernhout
Design and Analysis of Communication Systems
University of Twente
Enschede, Netherlands
r.p.j.fernhout@student.utwente.nl

***Abstract*—Network Time Protocol (NTP) servers play a critical role in maintaining synchronized time across the internet, enabling accurate timekeeping for a wide range of applications and services. In the early 2010s, NTP servers were abused in DDoS attacks through the use of diagnostic commands, such as monlist [1]. Much effort has been put into eliminating this kind of abuse, so that the usage of NTP in DDoS has declined [2]. However, some abuse remains prevalent today [3], while research into the network has stagnated. In this work, we do a current, global analysis of NTP servers on the IPv4 address space. We do so by developing a two-phase scanning technique, utilizing both ZMap and a custom tool called *ntpscan*. In our analysis we use reported characteristics of NTP servers like the reference ID and stratum, as well as a novel NTP daemon fingerprinting technique based on the version field in response packets. Using these methods we've detected over 5 million active NTP servers. We show that these servers report a wide range of operating systems, architectures, daemon versions and clock sources. We found 6617 servers with the *monlist* command enabled and measured their response sizes.**

***Index Terms*—Network Time Protocol, Network Enumeration, DDOS Amplification, Network Survey**

I. Introduction

The Network Time Protocol (NTP) is a networking protocol designed to synchronize the clocks of computers over a network.

The protocol attracted significant research attention in the 2010s, following its exploitation in large-scale DDoS amplification attacks.

However, while academic research into the network has stagnated in recent years, the NTP ecosystem itself has remained dynamic, with growing interest in the development and adoption of new server implementations like *chronyd*, *ntpsec*, and *ntpd-rs*.

We want to provide insight into the current state of the network. We are interested in the amount of NTP servers that are publicly accessible, their software versions and their behavioral characteristics.

Additionally, we are interested in the ability of the current network to still be used for DDoS amplification. In 2021, NTP was reportedly still commonly used in DDoS attacks [3]. The prevalence of these attacks suggests there may still be some vulnerable servers on the internet today.

In this paper, we show our methods for doing a global survey of the NTP network. This methodology combines the existing tool ZMap with the development of a custom NTP analysis tool called *ntpscan*. We additionally experiment with novel ways of classifying NTP servers based on their behavior.

To tackle this problem, we propose a number of research questions (RQ).

A. Research Question 1

How can we reliably discover an NTP server?

B. Research Question 2

How can we gather descriptive information from an NTP server?

C. Research Question 3

Using the methods from RQ1 and RQ2, what are the characteristics of discoverable NTP servers?

II. Background

A. Network Time Protocol

The Network Time Protocol is a protocol operating on UDP for synchronizing time across a network. It allows for a distributed network of reference servers, layered in strata, to serve accurate time from source clocks, like radio sources and GPS satellites, to clients over the internet. The following section will briefly explain some parts of the protocol that are relevant to our work.

In NTP, packets have a so-called *Association Mode*. In typical client-server interaction, the client will send mode 3 (client) packets, while the servers sends mode 4 (server) packets. The available modes are shown in Table I.

TABLE I: Association Modes

value	meaning
0	reserved
1	symmetric active
2	symmetric passive
3	client
4	server
5	broadcast
6	NTP control message
7	reserved for private use

Mode 6 (NTP control message) allows reading and potentially setting certain variables on a server. Mode 7 (private use) may be used by a daemon to extend the protocol. A typical client/server interaction uses mode 3 and 4 and the packet header shown in Fig. 1.

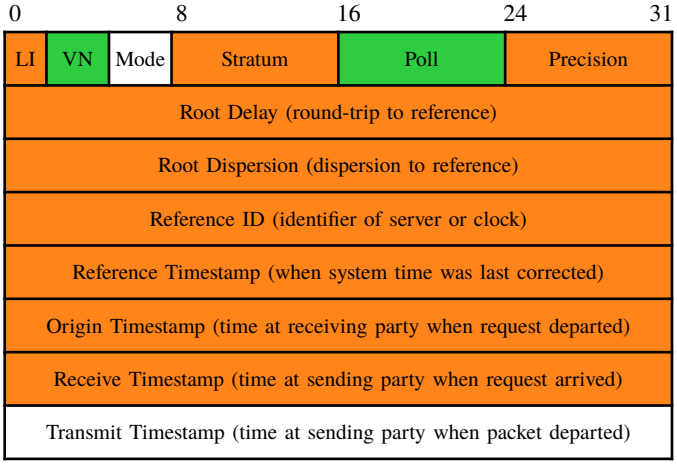

Fig. 1: NTP Packet Header Format

The NTP Project has a reference implementation for NTP called *ntpd*. Though it has also been called *xntpd*, we will refer to it as *ntpd*. We will use the term *daemon* to refer to arbitrary NTP server software, and we will use the term *server* to refer to a host that is serving NTP.

B. DDoS Amplification using NTP

UDP DDoS amplification is a technique where an attacker uses a public server running a UDP-based protocol to amplify a DDoS attack. This is done by finding a service (the *amplifier*) that can be made to respond with more data than it receives. Once such an amplifier is identified, the attacker sends UDP packets to the service with a spoofed source-IP of the victim.

NTP specifies a mode 6 command called *read variables*, which returns a large list of values about the server. Additionally, the reference implementation *ntpd* used to support a monitoring command called *monlist*, which returns the IPs of the last 600 peers [4]. *monlist* is not referenced in any RFC because it was implemented using mode 7 (private use). Both of these commands, though particularly the latter, can be used for DDoS amplification.

At the end of 2013, attackers started using these two NTP commands (especially *monlist*) as DDoS amplifiers. This quickly escalated, and the first quarter of 2014 would see 85% of DDoS attacks over 100Gbps use NTP [1].

Part of those attacks was a record-breaking 400Gbps attack on a Cloudflare (a DDOS mitigation service) client. This attack was big enough to cause network congestion in parts of Europe [5].

During this period, multiple researchers scanned the internet continuously for NTP amplifying servers [1], [2], [6], [7]. A campaign was started to reduce NTP amplifiers on the internet, which resulted in nearly 92% of NTP monlist amplifiers to be shut down within 13 weeks [2].

As a result of these efforts, many servers stopped responding to *monlist* request packets. However, even to this day, DDoS amplification attacks through NTP are still common [3], [8]. This suggests that there are still servers on the internet with the *monlist* or *read variables* commands enabled.

III. Related works

In 1999, an attempt was made to survey the entire NTP network [9]. However, the method used was walking the network graph, instead of scanning the entire address range. This was done by asking hosts for their peer list and monitor list, using the private mode *peers* and *monlist* commands, and then visiting the resulting hosts with a spider program.

Kührer et al (2014) tracked NTP amplifiers from the end of 2013 to the end of 2016, as part of a large-scale campaign to notify administrators of the amplification problems in NTP [2]. Their data shows that the campaign caused nearly 95% of NTP amplifiers to be shut down in just six months.

In the same period, openntpproject.org (*ONP*) also scanned for NTP servers that respond to *monlist*. The scan was performed using the *ntpdc* tool shipped with the reference implementation.

Czyz et al (2014) also scanned NTP servers around this time. They drew comparisons between their data set and those of Kührer and ONP [1].

Rudman (2015) scanned NTP DDoS related traffic in South-Africa in the same period. They specifically focused on the TTL values of NTP DDoS traffic [6].

Because Kührer et al (2014) only sent mode 6 *readvars* and mode 7 *monlist* commands, Rytilahti et al (2018) later complemented the data with two additional scans, which also included a generic mode 3 client request [7]. In the same paper, they also constantly requested time at public time services via DNS. They used the NTP pool, the ntp.org server list and servers from major vendors, like Google or Apple. They then tracked what IP addresses the domains resolved to.

IV. Methods

We decided to split our survey into two scans. First, we do a preliminary scan, with the sole purpose of listing all public NTP servers on the IPv4 address space. We then perform a second scan, which attempts to extract information from the servers found in the preliminary scan.

A. For reliably identifying NTP servers (RQ1)

Because NTP operates on UDP, discovery is trickier than with TCP services. The only way to verify that a server offers NTP is to send a valid NTP request and receive a response. A few existing tools can be used to probe UDP services. We decided to utilize zmap [10].

For the payload, we looked at another scanning tool called nmap, which is shipped with two NTP related probes found in its `nmap-service-probes` file, which nmap uses during its service/version scan. These probes are a mode 3 client request and a mode 1 symmetric active request. We extracted

these probes for use with zmap. zmap was locally compiled from the master branch, commit `5ee08f4`.

We chose zmap for its efficiency and ability to maintain a constant send rate. Our scan was performed at a rate of 100Mbps for a duration of 8 hours.

We used the default zmap blocklist, along with an additional blocklist provided by the university. This scan provided us with a long list of IPs that returned a response to one of the probes.

B. For gathering useful information from an NTP server (RQ2)

We developed a scanning routine that contains three parts. The type of packets we sent are mode 6 *readvars*, mode 7 *monlist* and mode 3 client packets.

a) *Mode 6 readvars:* The most obvious method of getting information from an NTP server is via the mode 6 *read variables* (also referred to as *readvars*) command. This command can be used to retrieve a set of variables from the daemon, including information like the daemon name and version, reference ID as an ASCII string, operating system and architecture.

However, not all daemons allow public access of this command.

b) *Mode 7 monlist:* The NTP specification has a mode reserved for private use. Ever since the first versions of *xntpd*, this was used as a way to add additional monitoring capabilities. The functionality of this command is now embedded in the mode 6 *mrulist* command. The mode 7 implementation in *ntpd* has slightly changed during its lifetime. Specifically, there is a pre-ipv6 and a post-ipv6 version, which use different implementation codes, namely `IMPL_XNTPD` and `IMPL_XNTP_OLD`. Additionally, there are two opcodes for *monlist* commands, `REQ_MONLIST` and `REQ_MONLIST_1`. During our scan, we tried all four combinations of implementation codes and opcodes.

c) *Mode 3 version responses:* Because not all servers respond to mode 6, we attempted to draw information from just the standard packet header. These packets contain very little information, as seen in Fig. 1. The second field in the header is the version of the NTP protocol in use. The RFC does not accurately describe how this field should be set, and we found slight differences in the behavior among daemons. We developed a novel fingerprinting technique for NTP daemons, which uses these versions. This technique consists of sending requests with all possible versions, 0 to 7. For each outgoing request, the transmit timestamp was randomized. The response packet will have this timestamp set as the origin timestamp, which allows us to map incoming responses to our requests.

During this scan, we also collect the reference IDs of incoming packets.

d) *On rate-limits:* The NTP project recommends enabling a rate-limiting feature in ntpd [11], and these recommendations are often the default in distributions of ntpd. The rate is not configurable and not clearly defined in the sourcecode. When a rate-limit is reached, the daemon may send a so-called Kiss-o'-Death packet and drop packets until the rate is lowered, or it may immediately start dropping packets. From our testing, we found that a burst of 8 mode 3 requests was enough to hit the rate-limit of *ntpd*.

e) *Choice of tooling: ntpd* is shipped with two CLI tools, which can be used for sending mode 6 and 7 packets. *ntpq* is used to send mode 6 packets and *ntpdc* is used to send mode 7 packets. However, because our methodology requires fine control the send rate and packet contents, we developed our own scanning tool, called *ntpscan* [12].

f) *ntpscan:* Our *ntpscan* [12] tool performs our three different scanning methods per peer in sequence and produces various output files, which can be used for analysis.

To account for rate-limiting, it maintains two rate related variables per target, the duration between sent packets (spread) and the timeout duration for when a RATE Kiss-o'-Death is received. Whenever a Kiss-o'-Death packet is received, these variables are doubled to slow the rate down for that peer.

For our final scan, we configured *ntpscan* to use an initial spread of 2 seconds. We found that with the latest FreeBSD release of *ntpd* with recommended security settings a two-second spread spread did not cause a Kiss-o'-Death packet to be sent. Because some servers may not send a Kiss-o'-Death packet before dropping packets, hitting rate-limits is prevented as much as possible.

We configured *ntpscan* to retry every out-going packet at least once. *ntpscan* has complicated logic surrounding rate-limits, and might choose to send more packets at a slower rate if it believes to have hit a rate-limit.

C. For performing the over-all scan and analysis (RQ3)

After we perform a preliminary enumeration scan, using the methods of RQ1, and a follow-up scan, using the methods of RQ2, we can perform analysis on the data. To aid in this analysis, we used *tcpdump* to capture all traffic associated with UDP port `123` while performing the second scan with *ntpscan* [12]. We then used *tshark* and a variety of Ruby scripts to extract additional information from the capture file.

V. Results

A. Preliminary scans

Fig. 2 shows the results of our different scans.

In total, we discovered 5,905,934 unique IPv4 addresses responding to the NTP packets. The `mode 1` packets got much fewer responses than the `mode 3` packets.

48,841 IPs were only discovered using `mode 1` requests, suggesting some servers may not respond to `mode 3` packets at all.

Due to the nature of this scan, it is impossible to tell if a response is from an actual NTP server. For instance, we found some servers which simply echo incoming port 123 UDP packets. Of the 5,905,934 IPs discovered in the preliminary scan, our second tool *ntpscan* wasn't able to get a valid response from 802,256.

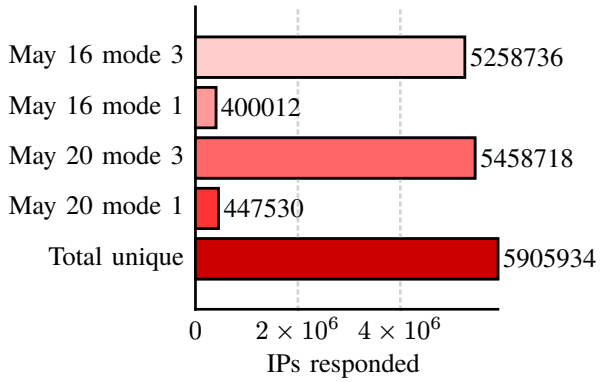

Fig. 2: Preliminary scan results.

B. Version responses

In text, we make use of a presentation like `1->3` to mean that a client (mode 3) packet with version 1 was responded to with a server (mode 4) packet with version 3.

In tables, we use a column per version sent, and the cell value is the version that was received. An empty cell indicates that the server did not respond to that version.

In Table II, we tested our tool with 5 different daemons.

In Table III, we tested `ntpscan` against various servers from known big tech companies.

It should be noted that this scan type is very sensitive to rate-limits. For instance, a chronyd server might be misrepresented as fingerprint `1->1 3->3 4->4` if all version 2 requests and/or responses were dropped or lost. We discuss this further in Section VI.

Table IV shows the most common fingerprints.

By far the most common fingerprint is the one shared by chronyd and ntpsec.

TABLE II: Version responses of popular daemons

	version response fingerprint							
daemon	**v0**	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
ntpd		1	2	3	4	5	6	7
chronyd		1	2	3	4			
ntpsec		1	2	3	4			
ntpd-rs				3	4			
openntpd		1	2	3	4	5	6	7

TABLE III: Version responses of servers by companies

	version response fingerprint							
company	**v0**	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
Microsoft		3	3	3	3			
Apple		1	2	3	4	5	6	7
Google		1	2	3	4	5	6	7
Cloudflare		1	2	3	4			
Amazon		1	2	3	4			
Meta		1	2	3	4			

TABLE IV: Top mode 3 version fingerprints

version response fingerprint								
v0	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**frequency**
	1	2	3	4				2777895
	1	2	3	4	5	6	7	1502870
0	1	2	3	4	5	6	7	158995
0	1	2	3	4				58003
	1	2	3					56050
3	3	3	3	3	3	3	3	39241
	3	3	3	3				33154
			3	4				32907

Aside from the common few fingerprints, we also discovered a few less-common but distinct ones, like always responding with 4: `0->4 1->4 2->4 3->4 4->4 5->4 6->4 7->4`, which was seen 1289 times. Or mirroring the client version except for versions beyond 4: `0->0 1->1 2->2 3->3 4->4 5->4 6->4 7->4`, which was seen 1770 times.

We also discovered servers that only responded to one version.

C. Reference IDs

Reference IDs in the NTP specification may be interpreted in three different ways depending on the stratum field. Stratum 1 servers use a 4-byte ASCII string identifying the reference clock. Servers with a higher stratum set the IPv4 address of the reference server as the reference ID, when using IPv6, they use the first 4 bytes of the MD5 digest of the IPv6 address. Stratum 0 packets are Kiss-o'-Death packets, and the reference ID is an ASCII string denoting the error.

We will first cover the reference IDs found on stratum 1 servers. The most common ID was empty with 4 null-bytes. The second was the ASCII string `LOCL`. See Fig. 3.

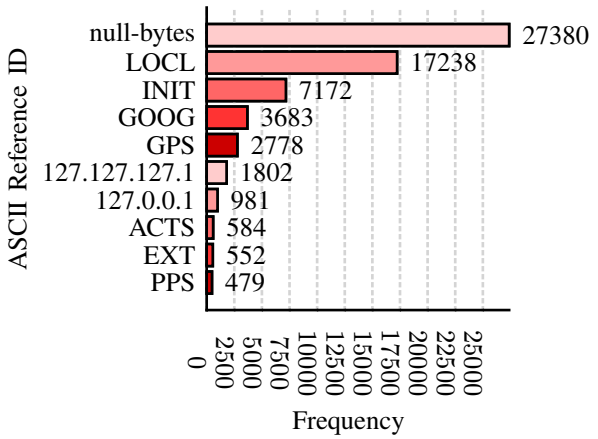

Fig. 3: Top few ASCII identifiers of stratum 1 servers

Internally, NTP uses addresses starting with octets `127.127` to refer to different local reference clocks.

> Drivers have addresses in the form 127.127.t.u, where t is the driver type and u is a unit number in the range 0-3 to distinguish multiple instances of the same driver.
>
> — NTP Reference Clock Support [13]

However, `127.127.127.1` isn't known to correspond to any specific driver.

Because so many stratum 0 servers responded with `GOOG` as the identifier, we decided to investigate these further, and found that 3504 of the 3683 discovered servers are from just 35 prefixes, nearly all of which start with octet `103`. These prefixes and additional comments are listed in Appendix B.

For servers that are not stratum 1, the reference ID is either the IPv4 address of the reference server, or the first four octets of the MD5 hash of the IPv6 address. We do not have a method for differentiating between the two, so we interpreted all as IPv4 addresses. Fig. 4 shows the most common reference IDs for servers beyond stratum 1.

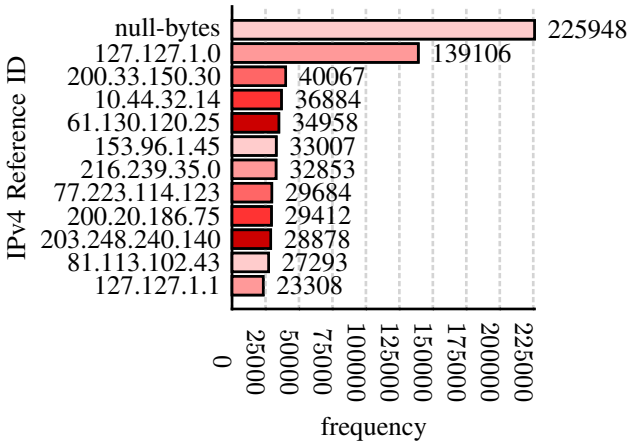

Fig. 4: Top few IPv4 reference identifiers for non-stratum-1 servers

Omitted from these results are servers which responded with stratum 0. These packets should be Kiss-o'-Death packets, but we have found some servers sending stratum 0 packets with common identifiers like `127.0.0.1` and `INIT`. This may indicate that these servers are wrongly setting the stratum field to 0.

The reference ID that is default on an uninitialized ntpd-rs daemon, `XNON`, was not discovered once.

A more complete list of reference IDs of stratum 1 servers can be found in Appendix A.

D. Strata

The most common stratum used was 3. Fig. 5 shows the frequencies of the ten most common strata.

E. Variables responses

We discovered 1,111,566 servers that responded to the `mode 6` *read variables* command. The scanner omitted responses from 1280 servers because they could not be converted to ASCII.

a) *Variable system:* The `system` variable contains the name of the operating system. We slightly altered the data by removing version and architecture information as well as the common `UNIX/` prefix. Table V shows the most common operating systems, a complete list is in Appendix D.

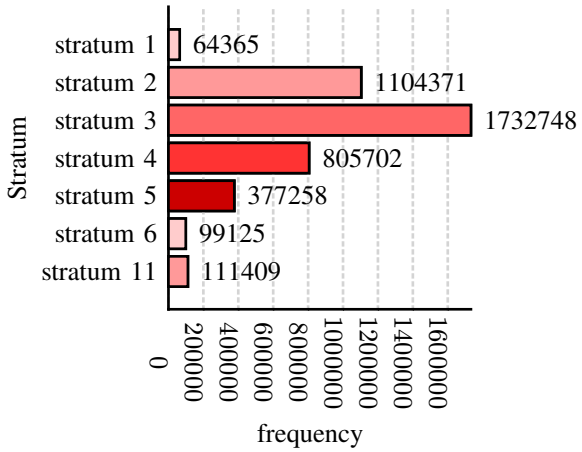

Fig. 5: Frequency of top 10 strata

TABLE V: MOST COMMON OPERATING SYSTEMS

operating system	frequency
UNIX	798692
cisco	95595
Linux	65676
/	62234
FreeBSD	50807
JUNOS	21925

b) *Variable version:* The `version` variable contains the version of the daemon. The most common versions are shown in Table VI. In the case of `ntpd` and `ntpsec` versions, we truncated them to just the first 3 digits (`x.y.z`). The complete list is in Appendix E.

TABLE VI: MOST COMMON DAEMON VERSIONS

daemon version	frequency
4	798586
ntpd 4.2.8	80451
ntpd 4.2.0	44281
ntpd 4.1.1	6287
ntpd 4.2.6	5409
ntpd 4.2.4	2419

c) *Variable processor:* The `processor` variable contains the architecture of the server. All architectures with more than a 1000 occurrences are listed in Table VII, the complete list is in Appendix F.

TABLE VII: MOST COMMON ARCHITECTURES

architecture	frequency
unknown	798731
amd64	47286
armv7l	37251
mips	10302
x86_64	9670
powerpc	8212
octeon	7912
i386	6173
arm	2379
aarch64	2324
ppc	2313
mips64	1417

d) *Other variables:* We found a number of unusual variable identifiers. We found 33 servers with a variable called `LANTIME`, for example `LANTIME=LANTIME/GPS170/M600/V7.08.024/SN030111`. These are likely instances of Meinberg LANTIME servers. We also found 7 servers using the variable `host`.

Furthermore, we found 118 servers with corrupted variable identifiers. This is an example of a corrupted response: `3="4", 3-926154393-926154393="unknown", 926154393="UNIX", leap=0, 4491979-stratum=3, precision=-24,` Additionally, we found 34,723 servers that add a single null-byte after their `refid` value. We found that both the servers with corrupted responses and servers with the null-byte all share the variables `version="4", processor="unknown", system="UNIX"` (when not corrupted) and the version fingerprint `1->1 2->2 3->3 4->4`.

e) *Response size:* We measured the sizes of the *readvars* responses and compiled the histogram in Fig. 6.

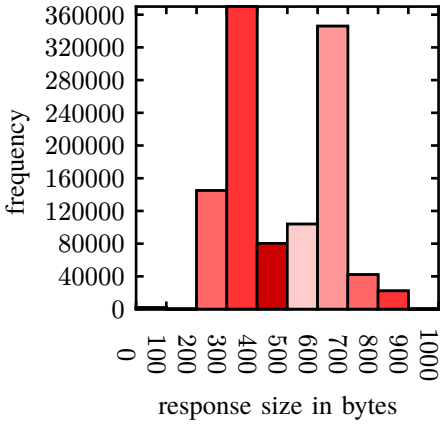

Fig. 6: Histogram of readvars response sizes

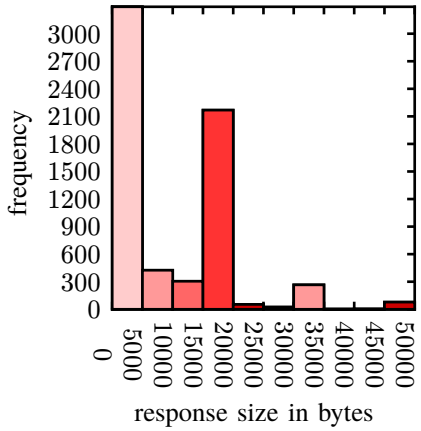

Fig. 7: Histogram of monlist response sizes

F. Monlist responses

We discovered 6617 servers that responded to monlist requests without errors.

Many responses were small, below 500 bytes. However, the most common response size we saw was 16,080 bytes, which we saw 2116 times. Given that the request is only 50 bytes, this is an amplification factor of ~321. We also notably received 252 responses of 32,160 bytes, and 74 responses of 48,800 bytes.

Table VIII shows the implementation code and opcode used by the responses. Some servers respond to both the old implementation code (`IMPL_XNTPD_OLD`) and the new implementation code (`IMPL_XNTPD`), in which case we used to biggest response.

TABLE VIII: IMPLEMENTATION CODE AND OPCODE OF MONLIST RESPONSES

implementation code	opcode	frequency
IMPL_XNTPD_OLD	REQ_MON_GETLIST	4899
IMPL_XNTPD_OLD	REQ_MON_GETLIST_1	149
IMPL_XNTPD	REQ_MON_GETLIST	1345
IMPL_XNTPD	REQ_MON_GETLIST_1	224

We've seen many servers where existing tools like *nmap* with the `ntp-monlist.nse` script and the latest version of *ntpdc* both weren't able to get a monlist response, but our custom tool was. This is because *ntpdc* sends a request with `opcode 42` (`REQ_MON_GETLIST_1` and only attempts other versions of the protocol after an error is received, but does nothing on a timeout. `nmap`'s `ntp-monlist.nse` script simply doesn't try other versions of the protocol. Our tool always sends a request with all possible codes unconditionally, which yields more results when daemon's do not respond to `code 42` at all. Additionally, *ntpdc* has had a bug since at least 2004, which causes it to report a timeout with large responses. [14]

We found 1118 servers that responded to both *monlist* and *readvars*. This allowed us to construct Table IX.

TABLE IX: NTPD VERSIONS OF MONLIST ENABLED SERVERS

version	frequency
ntpd 4.2.6	914
ntpd 4.2.0	90
ntpd 4.2.4	62
ntpd 4.2.2	29
ntpd 4.2.8	11
ntpd 4.2.5	10
ntpd 4.2.1	1

G. Servers acting as a proxy

We found a number of servers that reply with a different source-IP than what they were probed with. Using the randomized transmit timestamps from our fingerprint scan to pair mode 3 requests with mode 4 responses, we discovered 2151 servers that reply under a different source-IP.

H. Use of timestamps

During our analysis, we found a number of interesting behaviors regarding how servers set the timestamps in their mode 4 responses.

We saw 29,892 servers that set the Origin Timestamp to the same value of as their Transmit Timestamp, instead of using the Transmit Timestamp from the mode 3 packet. This behavior corresponds to earlier versions of the SNTP specification (RFC 4330) and interferes with our fingerprinting method because our randomized transmit timestamp is not reflected back.

We saw 3013 servers that round the Origin Timestamp to the nearest second, which also interferes with our fingerprinting method, but could be accounted for in future scans.

We saw 1,184,304 servers that set the Receive Timestamp to that of the Origin Timestamp.

I. Kiss-o'-Death RATE packets

We found only 38 servers that responded with Kiss-o'-Death RATE packets (signalling a rate-limit has been hit). This means that our default spread of 2 seconds between packets is likely sufficient.

VI. Discussion

Scanning for UDP services is difficult, as it is hard to detect when a packet is dropped. During our scan, we monitored for dropped packets on the host machine, but did not do so for upstream devices like firewalls and routers. We have found that intensive scans like these can easily reach a machine's max tracked connection count. Because of this, we think it may be a good idea to perform a slower preliminary scan, as well as having more monitoring set-up. A slower scan might also produce better scan results because there is less chance of triggering a remote firewall.

Aside from hitting firewalls, the importance of not hitting the daemon's rate-limit should also be stressed. We have found that many daemons are configured to not send Kiss-o'-Death packets. Without these packets, it is impossible to determine if a packet was dropped on purpose. When an undetected rate-limit has occurred, it will skew the version scan results. This is often recognizable by an impossible version fingerprint. We tried our best to tune the scanner for a proper balance of speed and accuracy.

We saw that the scan results were fairly volatile, which means it is important to minimize the time between the preliminary scan and the second in-depth scan. Our time difference of over a month between the scans may have degraded our results. Our tool and methodology also didn't allow us to check for false positives from the preliminary scan, as *ntpscan* just reports servers as offline when no valid packet is received. The preliminary scan did not differentiate between actual NTP servers responses and other UDP responses. For instance, we found a number of servers which just echo incoming packets.

We also think the monlist scanning ability of our custom scanner could be improved. We only received empty responses to the pre-ipv6 protocol implementation (`IMPL_XNTPD_OLD`), which could mean that our tool does not send proper requests for that protocol. As that version has no support for error codes, it is harder to work with. There's also no documentation for the either mode 7 implementation, so we worked solely by reviewing the *ntpdc* and *ntpd* sourecode and reverse engineering packets. We believe it might be possible to find more monlist enabled servers by properly trying all 4 combinations of implementation and request codes.

Our fingerprinting method relies on sending a randomized transmit timestamp to pair responses packets to their requests. However, we found cases where a server does not directly copy the transmit timestamp to the origin timestamp of the response packet, as shown in Section V.H.

There are a number of things we did not test for, but for which we think data might be interesting. For instance, we did not try modes other than 3, 6 and 7. We also did not try commands other than *readvars* and *monlist* in modes 6 and 7. We believe there might be cases of broken authentication for mode 6, which could allow an attacker to set variables or use the *mrulist* command, which replaced *monlist*. For fingerprinting, it might also be interesting to send types of invalid packets, like using mode 0 or sending the wrong packet size. We did manually analyze the behavior of how timestamps are set by servers in Section V.H and in the future we would like to incorporate this analysis into *ntpscan* and draw comparisons with the version fingerprint.

We would also like to see more research done into servers that appear to be proxy-ing other servers.

VII. Conclusion

To answer RQ1, we believe to have found a reliable way of discovering NTP servers using zmap. We nearly had 6 million results, although this did include some false positives. Other methods may result in less false positives, like using nmap, of which the service/version scan has the ability to parse incoming packets.

For RQ2, we believe that especially the *readvars* command is of value when gathering information from an NTP server, as it is an immediate means of gathering valuable information while only sending a single packet, and we were surprised to see nearly 22% of daemons respond to it. For servers that do not respond to this command, we believe our fingerprinting method using the version in mode 3 packets to be useful. However, when using fingerprinting methods that rely on the presence of certain responses, extreme care should be taken not to invoke the rate-limits of the daemon.

For finding monlist enabled servers, we already found our custom tool to yield more results than *ntpdc*, in at least some areas. Yet *ntpscan*'s ability to scan for monlist enabled servers is still incomplete, as it does not properly implement the pre-ipv6 implementation of mode 7. We think there might be even more monlist servers that have gone undetected.

To answer RQ3, we found the vast majority of servers to have very similar characteristics. However, overall, we found a great variety different operating systems, daemon versions and architectures. Many of the characteristics we found correspond to routers, firewalls and other network infrastructure.

Due to the number of NTP servers with monlist support and their response sizes, we believe DDoS amplification through NTP is likely still viable. We found many monlist enabled servers to be from just a few AS numbers, and we hope that by contacting these hosts we could further reduce the number of public monlist enabled servers.

References

[1] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir, "Taming the 800 Pound Gorilla: The Rise and Decline of NTP DDoS Attacks," in *Proceedings of the 2014 Conference on Internet Measurement Conference*, in IMC '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 435–448. doi: 10.1145/2663716.2663717.

[2] M. Kührer, T. Hupperich, C. Rossow, and T. Holz, "Exit from hell? reducing the impact of amplification DDoS attacks," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, in SEC'14. San Diego, CA: USENIX Association, 2014, pp. 111–125.

[3] D. Kopp, C. Dietzel, and O. Hohlfeld, "DDoS Never Dies? An IXP Perspective on DDoS Amplification Attacks," in *Passive and Active Measurement*, O. Hohlfeld, A. Lutu, and D. Levin, Eds., Cham: Springer International Publishing, 2021, pp. 284–301.

[4] Accessed: Jun. 19, 2025. [Online]. Available: https://www.ntp.org/documentation/4.2.8-series/ntpdc/

[5] Matthew Prince, Accessed: Apr. 25, 2025. [Online]. Available: https://blog.cloudflare.com/technical-details-behind-a-400gbps-ntp-amplification-ddos-attack/

[6] L. Rudman and B. Irwin, "Characterization and analysis of NTP amplification based DDoS attacks," in *2015 Information Security for South Africa (ISSA)*, 2015, pp. 1–5. doi: 10.1109/ISSA.2015.7335069.

[7] T. Rytilahti, D. Tatang, J. Köpper, and T. Holz, "Masters of Time: An Overview of the NTP Ecosystem," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018, pp. 122–136. doi: 10.1109/EuroSP.2018.00017.

[8] J. Krupp, M. Karami, C. Rossow, D. McCoy, and M. Backes, "Linking Amplification DDoS Attacks to Booter Services," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds., Cham: Springer International Publishing, 2017, pp. 427–449.

[9] N. Minar, "A Survey of the NTP Network," 1999. [Online]. Available: https://api.semanticscholar.org/CorpusID:17133789

[10] Z. Durumeric, E. Wustrow, and J. A. Halderman, "ZMap: Fast Internet-wide scanning and its security applications," in *22nd USENIX Security Symposium*, 2013.

[11] NTP Project, [Online]. Available: https://support.ntp.org/Support/AccessRestrictions

[12] Rein Fernhout, [Online]. Available: https://github.com/LevitatingBusinessMan/ntpscan

[13] Dave Mills, [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/refclock.html

[14] NTP Project, [Online]. Available: https://bugs.ntp.org/show_bug.cgi?id=286

Appendix A: ASCII Reference IDs

The following table consists of all reference IDs report by stratum 1 servers which occurred more than once.

In cases where the string is not valid ASCII, we either converted the string to an IP in range 127.0.0.1/8 or displayed the hexadecimal value instead.

Reference ID	frequency
0x00000000	27380
LOCL	17238
INIT	7172
GOOG	3683
GPS	2778
127.127.127.1	1802
127.0.0.1	981
ACTS	584
EXT	552
PPS	479
PTP	211
VMTP	207
GNSS	171
NICT	125
MRS	92
PTP0	51
PHC0	47
NIST	38
GPSs	33
DCF	28
MBGh	23
PPS0	22
MBS	21
IRIG	20
SPG	18
MX7	17
GSL	17
ATOM	15
SEL_	14
PZF	14
CTD	14
0xd05b703c	13
BD	13
DCFa	12
127.127.1.1	12
WAAS	11
127.127.1.0	11
oa	9
kPPS	9
PPS	8
FREQ	7
GLN	7
GPGL	7
LOCA	7
GPPS	7
pps	7
LCL	7
NMEA	6
BDS	6
SHM	6
TSTR	6
IRG0	6
0x00001251	5
DCFs	5
TC	5
GPS	5
OJJY	4
BBgp	4
MBG	4
PSM0	4
OLEG	4
0xa625da5a	4
PTP	4
0xa9fea97a	4
DTS	4
CABL	4
0x00000001	3
TMNL	3
PHC	3
LCOL	3
0xc057ad48	3
GPS0	3
C	3

Reference ID	frequency
GMR	3
0xa6258257	3
ONBR	3
KPPS	2
PZFs	2
HARD	2
T.C.	2
FREE	2
SOCK	2
ROA	2
GPS1	2
PTPs	2
ANT2	2
MSF	2
127.127.7.1	2
XFAK	2
NOVL	2
ANT1	2
SH0	2
GAL	2
GPSD	2
G+BD	2
GNSs	2
GPS<space>	2

Appendix B: GOOG responses

The following list is all `\24` prefixes of which 10 or more IPs responded with a stratum 1 `GOOG` response. According to whois records, all of them are registered in Bangladesh.

```
103.112.204.0\24 (119)
103.112.206.0\24 (17)
103.112.207.0\24 (51)
103.122.74.0\24 (16)
103.126.149.0\24 (17)
103.12.74.0\24 (32)
103.13.193.0\24 (12)
103.137.66.0\24 (32)
103.137.67.0\24 (29)
103.142.184.0\24 (23)
103.143.182.0\24 (256)
103.143.183.0\24 (256)
103.146.16.0\24 (84)
103.146.17.0\24 (158)
103.150.255.0\24 (16)
103.155.52.0\24 (167)
103.155.53.0\24 (112)
103.159.254.0\24 (16)
103.167.190.0\24 (256)
103.167.191.0\24 (256)
103.174.23.0\24 (40)
103.178.188.0\24 (256)
103.229.255.0\24 (42)
103.55.242.0\24 (256)
103.55.243.0\24 (256)
103.58.74.0\24 (24)
103.58.75.0\24 (18)
103.75.138.0\24 (256)
180.94.28.0\24 (28)
180.94.29.0\24 (256)
203.76.220.0\24 (53)
203.76.221.0\24 (28)
203.76.222.0\24 (40)
203.76.223.0\24 (12)
```

It should be noted that we did not only receive stratum 1 `GOOG` packets from these prefixes. They were also found to contain many servers configured as stratum 3 or higher.

Appendix C: Common version fingerprints

The following table contains all fingerprints that appeared more than a 1000 times.

version response fingerprint								
v0	v1	v2	v3	v4	v5	v6	v7	frequency
	1	2	3	4				2777895
	1	2	3	4	5	6	7	1502870
0	1	2	3	4	5	6	7	158995
0	1	2	3	4				58003
	1	2	3					56050
3	3	3	3	3	3	3	3	39241
	3	3	3	3				33154
			3	4				32907
		2	3	4				31945
	1	2		4				31791
	1		3	4				30825
	1	2	3	4	5	6		16551
	1	2	3	4	5		7	14836
	1	2	3	3	3	3	3	14729
	1	2	3	4		6	7	14391
		2	3	4	5	6	7	14194
	1	2	3		5	6	7	14188
	1	2		4	5	6	7	13897
	1		3	4	5	6	7	13539
	1	2						8595
	1		3					7729
		2	3					7453

version response fingerprint								
v0	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**frequency**
	1			4				7050
		2		4				6473
		3	3	3				6144
	1							4466
			3					4164
				4				4063
		2						4022
	1	2	3	4	5			3543
	1	2	3	4		6		2822
	1	2	3	4			7	2730
	1	2	3			6	7	2586
	1	2	3		5	6		2559
0	1	2	3	4	5	6		2493
	1	2	3		5		7	2479
	1	2			5	6	7	2428
0	1	2	3	4	5		7	2381
	1			4	5	6	7	2353
	1	2		4		6	7	2345
	1	2		4	5	6		2344
	1	2		4	5		7	2339
0	1	2	3	4		6	7	2322
	1		3	4	5	6		2268
	1		3		5	6	7	2264
0	1	2		4	5	6	7	2240
3	3	3	3	3	3		3	2187
	1		3	4		6	7	2185
0	1	2	3		5	6	7	2184
		2	3	4	5	6		2182
			3	4	5	6	7	2177
		2	3		5	6	7	2176
		2		4	5	6	7	2173
0	1		3	4	5	6	7	2159
3	3	3	3	3	3	3	3	2143
	1		3	4	5		7	2138
3	3	3	3	3	3	3		2131
0		2	3	4	5	6	7	2106
		2	3	4		6	7	2066
		2	3	4	5		7	2065
3	3	3	3	3		3	3	2050
3	3	3		3	3	3	3	1919
3	3	3	3		3	3	3	1915

version response fingerprint								
v0	**v1**	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**frequency**
3	3		3	3	3	3	3	1891
3		3	3	3	3	3	3	1829
0	1	2	3	4	4	4	4	1770
4	4	4	4	4	4	4	4	1289
	1	2	3			6		1031
	1	2	3		5			1016

Appendix D: System variable responses

operating system	**frequency**
UNIX	798692
cisco	95595
Linux	65676
/	62234
FreeBSD	25415
FreeBSDJNPR	25392
JUNOS	21925
SunOS	351
vxworks	302
QNX	179
VMkernel	120
AIX	100
Windows	87
NetBSD	79
Isilon OneFS	32
eCos	56
HPUX	33
Data ONTAP	17
Darwin	12
unknown	11
Chiaros	9
SecureOS	9
BRIX	9
LeoNTP	7
OpenVMS	5
WINDOWS/NT	5
EqualLogic (R) storage array	2
HongmengOS	2
BSD/OS	2
SCO	2
DECOSF1	2
NOS	1

operating system	frequency
IRIX	1
Win2003	1
Unixware2	1
Solaris	1
GBOS	1
Moscad ACE	1
IPSO	1
NetWare	1
uClinux	1

Appendix E: Daemon versions

version	frequency
4	798586
ntpd 4.2.8	80451
ntpd 4.2.0	44281
ntpd 4.1.1	6287
ntpd 4.2.6	5409
ntpd 4.2.4	2419
ntpd 4.1.0	527
ntpd 4.2.7	373
ntpd 4.3.99	343
ntpd 4.3.105	272
ntpd 4.2.2	154
ntpd 4.1.2	26
ntpd 4.2.5	17
ntpd 4.2.1	12
unknown	11
2	9
ntpd 4.0.99	6
Wangjing NTP 1.0	5
ntpd 4.1.72	4
ntpd ntpsec-1.1.3	8
ntpd ntpsec-1.2.2	3
ntpd ntpsec-1.1.0	3
ntpd 4.3.93	2
Domain	1

Appendix F: Architectures

architecture	frequency
unknown	798731
amd64	47286
armv7l	37251
mips	10302
x86_64	9670
powerpc	8212
octeon	7912
i386	6173
arm	2379
aarch64	2324
ppc	2313
mips64	1417
arm64	786
i686	673
armv5tejl	579
armv3b	412
UltraSparc-IIe	300
armv7b	254
armv6l	223
armv5tel	189
Titan-AM335X	123
aarch64_be	86
x86	80
xlr	65
i586	46
sun4v	41
Tridium_NPM-6xx_Board	29
i86pc	27
kppc	24
sbmips	22
sh4	17
Power Macintosh	12
PowerPC	10
i486	9
armv4tl	9
armv4l	9
blackfin	8
se100	8
sun4u	7
ARMv7E-M	7
x86-SSE2	6
AM335X	5
OHSYS3	5
Tridium_NPM3xx_Board	5

architecture	frequency
Jace_PPC_405	5
Tridium_Jace7xx_Board	4
seil4	2
x64	2
Working	2
00FBFA5F4B00	2
armv5teb	2
m68k	1
Tankvision_NXA8x	1
s390x	1
AT91SAM9260	1
00F9C1964C00	1
seil3	1
00FA74164C00	1
Sabre_SDB-Freescale_i.MX6_SoloX_Sabre_SDB	1
armv3l	1
Honeywell_IPC-QNX7_on_i.MX6_SoloX	1
x86pc	1
8540ADS	1
Edge10	1
Infinera-AMM	1
m68knommu	1
evbarm	1