Internet Engineering Task Force (IETF)
Request for Comments: 5905
Obsoletes: 1305, 4330
Category: Standards Track
ISSN: 2070-1721

D. Mills
U. Delaware
J. Martin, Ed.
ISC
J. Burbank
W. Kasch
JHU/APL

June 2010

Network Time Protocol Version 4: Protocol and Algorithms Specification

Abstract

The Network Time Protocol (NTP) is widely used to synchronize computer clocks in the Internet. This document describes NTP version 4 (NTPv4), which is backwards compatible with NTP version 3 (NTPv3), described in RFC 1305, as well as previous versions of the protocol. NTPv4 includes a modified protocol header to accommodate the Internet Protocol version 6 address family. NTPv4 includes fundamental improvements in the mitigation and discipline algorithms that extend the potential accuracy to the tens of microseconds with modern workstations and fast LANs. It includes a dynamic server discovery scheme, so that in many cases, specific server configuration is not required. It corrects certain errors in the NTPv3 design and implementation and includes an optional extension mechanism.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc5905.

Table of Contents

1. Introduction 4
   1.1. Requirements Notation 5
2. Modes of Operation 6
3. Protocol Modes 6
   3.1. Dynamic Server Discovery 7
4. Definitions 8
5. Implementation Model 10
6. Data Types 12
7. Data Structures 16
   7.1. Structure Conventions 16
   7.2. Global Parameters 16
   7.3. Packet Header Variables 17
   7.4. The Kiss-o'-Death Packet 24
   7.5. NTP Extension Field Format 25
8. On-Wire Protocol 26
9. Peer Process 30
   9.1. Peer Process Variables 31
   9.2. Peer Process Operations 33
10. Clock Filter Algorithm 37

11. System Process 39
11.1. System Process Variables 40
11.2. System Process Operations 41
11.2.1. Selection Algorithm 43
11.2.2. Cluster Algorithm 44
11.2.3. Combine Algorithm 45
11.3. Clock Discipline Algorithm 47
12. Clock-Adjust Process 51
13. Poll Process 51
13.1. Poll Process Variables 51
13.2. Poll Process Operations 52
14. Simple Network Time Protocol (SNTP) 54
15. Security Considerations 55
16. IANA Considerations 58
17. Acknowledgements 59
18. References 59
18.1. Normative References 59
18.2. Informative References 59
Appendix A. Code Skeleton 61
A.1. Global Definitions 61
A.1.1. Definitions, Constants, Parameters 61
A.1.2. Packet Data Structures 65
A.1.3. Association Data Structures 66
A.1.4. System Data Structures 68
A.1.5. Local Clock Data Structures 69
A.1.6. Function Prototypes 69
A.2. Main Program and Utility Routines 70
A.3. Kernel Input/Output Interface 73
A.4. Kernel System Clock Interface 74
A.5. Peer Process 76
A.5.1. receive() 77
A.5.2. clock_filter() 85
A.5.3. fast_xmit() 88
A.5.4. access() 89
A.5.5. System Process 90
A.5.6. Clock Adjust Process 103
A.5.7. Poll Process 104

1. Introduction

This document defines the Network Time Protocol version 4 (NTPv4), which is widely used to synchronize system clocks among a set of distributed time servers and clients. It describes the core architecture, protocol, state machines, data structures, and algorithms. NTPv4 introduces new functionality to NTPv3, as described in [RFC1305], and functionality expanded from Simple NTP version 4 (SNTPv4) as described in [RFC4330] (SNTPv4 is a subset of NTPv4). This document obsoletes [RFC1305] and [RFC4330]. While certain minor changes have been made in some protocol header fields, these do not affect the interoperability between NTPv4 and previous versions of NTP and SNTP.

The NTP subnet model includes a number of widely accessible primary time servers synchronized by wire or radio to national standards. The purpose of the NTP protocol is to convey timekeeping information from these primary servers to secondary time servers and clients via both private networks and the public Internet. Precisely tuned algorithms mitigate errors that may result from network disruptions, server failures, and possible hostile actions. Servers and clients are configured such that values flow towards clients from the primary servers at the root via branching secondary servers.

The NTPv4 design overcomes significant shortcomings in the NTPv3 design, corrects certain bugs, and incorporates new features. In particular, expanded NTP timestamp definitions encourage the use of the floating double data type throughout the implementation. As a result, the time resolution is better than one nanosecond, and frequency resolution is less than one nanosecond per second. Additional improvements include a new clock discipline algorithm that is more responsive to system clock hardware frequency fluctuations. Typical primary servers using modern machines are precise within a few tens of microseconds. Typical secondary servers and clients on fast LANs are within a few hundred microseconds with poll intervals up to 1024 seconds, which was the maximum with NTPv3. With NTPv4, servers and clients are precise within a few tens of milliseconds with poll intervals up to 36 hours.

The main body of this document describes the core protocol and data structures necessary to interoperate between conforming implementations. Appendix A contains a full-featured example in the form of a skeleton program, including data structures and code segments for the core algorithms as well as the mitigation algorithms used to enhance reliability and accuracy. While the skeleton program and other descriptions in this document apply to a particular implementation, they are not intended as the only way the required functions can be implemented. The contents of Appendix A are non-

normative examples designed to illustrate the protocol's operation and are not a requirement for a conforming implementation. While the NTPv3 symmetric key authentication scheme described in this document has been carried over from NTPv3, the Autokey public key authentication scheme new to NTPv4 is described in [RFC5906].

The NTP protocol includes modes of operation described in Section 2 using data types described in Section 6 and data structures described in Section 7. The implementation model described in Section 5 is based on a threaded, multi-process architecture, although other architectures could be used as well. The on-wire protocol described in Section 8 is based on a returnable-time design that depends only on measured clock offsets, but does not require reliable message delivery. Reliable message delivery such as TCP [RFC0793] can actually make the delivered NTP packet less reliable since retries would increase the delay value and other errors. The synchronization subnet is a self-organizing, hierarchical, master-slave network with synchronization paths determined by a shortest-path spanning tree and defined metric. While multiple masters (primary servers) may exist, there is no requirement for an election protocol.

This document includes material from [ref9], which contains flow charts and equations unsuited for RFC format. There is much additional information in [ref7], including an extensive technical analysis and performance assessment of the protocol and algorithms in this document. The reference implementation is available at www.ntp.org.

The remainder of this document contains numerous variables and mathematical expressions. Some variables take the form of Greek characters, which are spelled out by their full case-sensitive name. For example, DELTA refers to the uppercase Greek character, while delta refers to the lowercase character. Furthermore, subscripts are denoted with '_'; for example, theta_i refers to the lowercase Greek character theta with subscript i, or phonetically theta sub i. In this document, all time values are in seconds (s), and all frequencies will be specified as fractional frequency offsets (FFOs) (pure number). It is often convenient to express these FFOs in parts per million (ppm).

1.1. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. Modes of Operation

An NTP implementation operates as a primary server, secondary server, or client. A primary server is synchronized to a reference clock directly traceable to UTC (e.g., GPS, Galileo, etc.). A client synchronizes to one or more upstream servers, but does not provide synchronization to dependent clients. A secondary server has one or more upstream servers and one or more downstream servers or clients. All servers and clients who are fully NTPv4-compliant MUST implement the entire suite of algorithms described in this document. In order to maintain stability in large NTP subnets, secondary servers SHOULD be fully NTPv4-compliant. Alternative algorithms MAY be used, but their output MUST be identical to the algorithms described in this specification.

3. Protocol Modes

There are three NTP protocol variants: symmetric, client/server, and broadcast. Each is associated with an association mode (a description of the relationship between two NTP speakers) as shown in Figure 1. In addition, persistent associations are mobilized upon startup and are never demobilized. Ephemeral associations are mobilized upon the arrival of a packet and are demobilized upon error or timeout.

Association Mode	Assoc. Mode Value	Packet Mode Value
Symmetric Active	1	1 or 2
Symmetric Passive	2	1
Client	3	4
Server	4	3
Broadcast Server	5	5
Broadcast Client	6	N/A

Figure 1: Association and Packet Modes

In the client/server variant, a persistent client sends packet mode 4 packets to a server, which returns packet mode 3 packets. Servers provide synchronization to one or more clients, but do not accept synchronization from them. A server can also be a reference clock driver that obtains time directly from a standard source such as a GPS receiver or telephone modem service. In this variant, clients pull synchronization from servers.

In the symmetric variant, a peer operates as both a server and client using either a symmetric active or symmetric passive association. A persistent symmetric active association sends symmetric active (mode 1) packets to a symmetric active peer association. Alternatively, an ephemeral symmetric passive association can be mobilized upon the arrival of a symmetric active packet with no matching association. That association sends symmetric passive (mode 2) packets and persists until error or timeout. Peers both push and pull synchronization to and from each other. For the purposes of this document, a peer operates like a client, so references to client imply peer as well.

In the broadcast variant, a persistent broadcast server association sends periodic broadcast server (mode 5) packets that can be received by multiple clients. Upon reception of a broadcast server packet without a matching association, an ephemeral broadcast client (mode 6) association is mobilized and persists until error or timeout. It is useful to provide an initial volley where the client operating in client mode exchanges several packets with the server, so as to calibrate the propagation delay and to run the Autokey security protocol, after which the client reverts to broadcast client mode. A broadcast server pushes synchronization to clients and other servers.

Loosely following the conventions established by the telephone industry, the level of each server in the hierarchy is defined by a stratum number. Primary servers are assigned stratum one; secondary servers at each lower level are assigned stratum numbers one greater than the preceding level. As the stratum number increases, its accuracy degrades depending on the particular network path and system clock stability. Mean errors, measured by synchronization distances, increase approximately in proportion to stratum numbers and measured round-trip delay.

As a standard practice, timing network topology should be organized to avoid timing loops and minimize the synchronization distance. In NTP, the subnet topology is determined using a variant of the Bellman-Ford distributed routing algorithm, which computes the shortest-path spanning tree rooted on the primary servers. As a result of this design, the algorithm automatically reorganizes the subnet, so as to produce the most accurate and reliable time, even when there are failures in the timing network.

3.1. Dynamic Server Discovery

There are two special associations, manycast client and manycast server, which provide a dynamic server discovery function. There are two types of manycast client associations: persistent and ephemeral. The persistent manycast client sends client (mode 3) packets to a

designated IPv4 or IPv6 broadcast or multicast group address. Designated manycast servers within range of the time-to-live (TTL) field in the packet header listen for packets with that address. If a server is suitable for synchronization, it returns an ordinary server (mode 4) packet using the client's unicast address. Upon receiving this packet, the client mobilizes an ephemeral client (mode 3) association. The ephemeral client association persists until error or timeout.

A manycast client continues sending packets to search for a minimum number of associations. It starts with a TTL equal to one and continuously adding one to it until the minimum number of associations is made or when the TTL reaches a maximum value. If the TTL reaches its maximum value and yet not enough associations are mobilized, the client stops transmission for a time-out period to clear all associations, and then repeats the search cycle. If a minimum number of associations has been mobilized, then the client starts transmitting one packet per time-out period to maintain the associations. Field constraints limit the minimum value to 1 and the maximum to 255. These limits may be tuned for individual application needs.

The ephemeral associations compete among themselves. As new ephemeral associations are mobilized, the client runs the mitigation algorithms described in Sections 10 and 11.2 for the best candidates out of the population, the remaining ephemeral associations are timed out and demobilized. In this way, the population includes only the best candidates that have most recently responded with an NTP packet to discipline the system clock.

4. Definitions

A number of technical terms are defined in this section. A timescale is a frame of reference where time is expressed as the value of a monotonically increasing binary counter with an indefinite number of bits. It counts in seconds and fractions of a second, when a decimal point is employed. The Coordinated Universal Time (UTC) timescale is defined by ITU-R TF.460 [ITU-R_TF.460]. Under the auspices of the Metre Convention of 1865, in 1975 the CGPM [CGPM] strongly endorsed the use of UTC as the basis for civil time.

The Coordinated Universal Time (UTC) timescale represents mean solar time as disseminated by national standards laboratories. The system time is represented by the system clock maintained by the hardware and operating system. The goal of the NTP algorithms is to minimize both the time difference and frequency difference between UTC and the system clock. When these differences have been reduced below nominal tolerances, the system clock is said to be synchronized to UTC.

The date of an event is the UTC time at which the event takes place. Dates are ephemeral values designated with uppercase T. Running time is another timescale that is coincident to the synchronization function of the NTP program.

A timestamp T(t) represents either the UTC date or time offset from UTC at running time t. Which meaning is intended should be clear from the context. Let T(t) be the time offset, R(t) the frequency offset, and D(t) the aging rate (first derivative of R(t) with respect to t). Then, if T(t_0) is the UTC time offset determined at t = t_0, the UTC time offset at time t is

T(t) = T(t_0) + R(t_0)(t-t_0) + 1/2 * D(t_0)(t-t_0)^2 + e,

where e is a stochastic error term discussed later in this document. While the D(t) term is important when characterizing precision oscillators, it is ordinarily neglected for computer oscillators. In this document, all time values are in seconds (s) and all frequency values are in seconds-per-second (s/s). It is sometimes convenient to express frequency offsets in parts-per-million (ppm), where 1 ppm is equal to 10^(-6) s/s.

It is important in computer timekeeping applications to assess the performance of the timekeeping function. The NTP performance model includes four statistics that are updated each time a client makes a measurement with a server. The offset (theta) represents the maximum-likelihood time offset of the server clock relative to the system clock. The delay (delta) represents the round-trip delay between the client and server. The dispersion (epsilon) represents the maximum error inherent in the measurement. It increases at a rate equal to the maximum disciplined system clock frequency tolerance (PHI), typically 15 ppm. The jitter (psi) is defined as the root-mean-square (RMS) average of the most recent offset differences, and it represents the nominal error in estimating the offset.

While the theta, delta, epsilon, and psi statistics represent measurements of the system clock relative to each server clock separately, the NTP protocol includes mechanisms to combine the statistics of several servers to more accurately discipline and calibrate the system clock. The system offset (THETA) represents the maximum-likelihood offset estimate for the server population. The system jitter (PSI) represents the nominal error in estimating the system offset. The delta and epsilon statistics are accumulated at each stratum level from the reference clock to produce the root delay (DELTA) and root dispersion (EPSILON) statistics. The synchronization distance (LAMBDA) equal to EPSILON + DELTA / 2 represents the maximum error due to all causes. The detailed

formulations of these statistics are given in Section 11.2. They are available to the dependent applications in order to assess the performance of the synchronization function.

5. Implementation Model

Figure 2 shows the architecture of a typical, multi-threaded implementation. It includes two processes dedicated to each server, a peer process to receive messages from the server or reference clock, and a poll process to transmit messages to the server or reference clock.

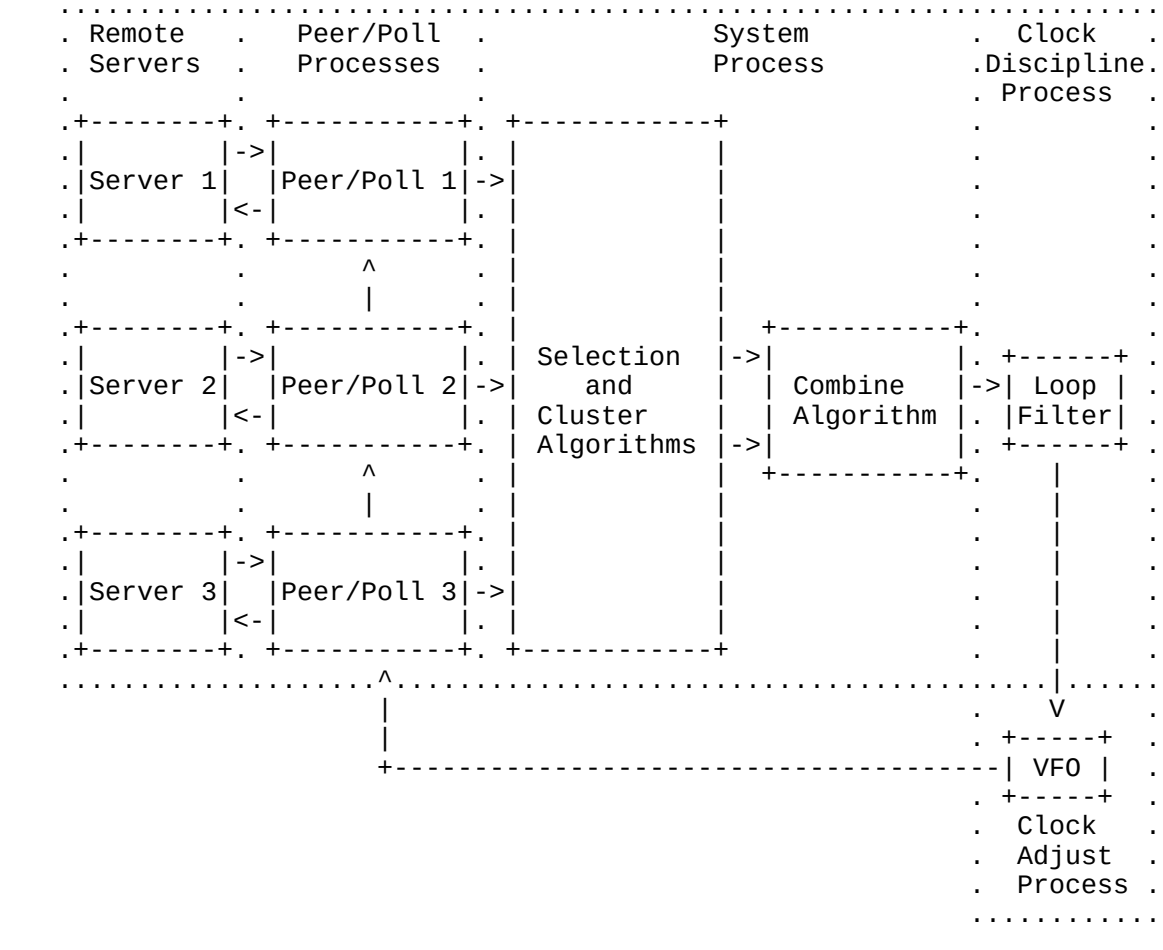


Figure 2: Implementation Model

These processes operate on a common data structure, called an association, which contains the statistics described above along with various other data described in Section 9. A client sends packets to one or more servers and then processes returned packets when they are received. The server interchanges source and destination addresses and ports, overwrites certain fields in the packet and returns it immediately (in the client/server mode) or at some time later (in the symmetric modes). As each NTP message is received, the offset theta between the peer clock and the system clock is computed along with the associated statistics delta, epsilon, and psi.

The system process includes the selection, cluster, and combine algorithms that mitigate among the various servers and reference clocks to determine the most accurate and reliable candidates to synchronize the system clock. The selection algorithm uses Byzantine fault detection principles to discard the presumably incorrect candidates called "falsetickers" from the incident population, leaving only good candidates called "truechimers". A truechimer is a clock that maintains timekeeping accuracy to a previously published and trusted standard, while a falseticker is a clock that shows misleading or inconsistent time. The cluster algorithm uses statistical principles to find the most accurate set of truechimers. The combine algorithm computes the final clock offset by statistically averaging the surviving truechimers.

The clock discipline process is a system process that controls the time and frequency of the system clock, here represented as a variable frequency oscillator (VFO). Timestamps struck from the VFO close the feedback loop that maintains the system clock time. Associated with the clock discipline process is the clock-adjust process, which runs once each second to inject a computed time offset and maintain constant frequency. The RMS average of past time offset differences represents the nominal error or system clock jitter. The RMS average of past frequency offset differences represents the oscillator frequency stability or frequency wander. These terms are given precise interpretation in Section 11.3.

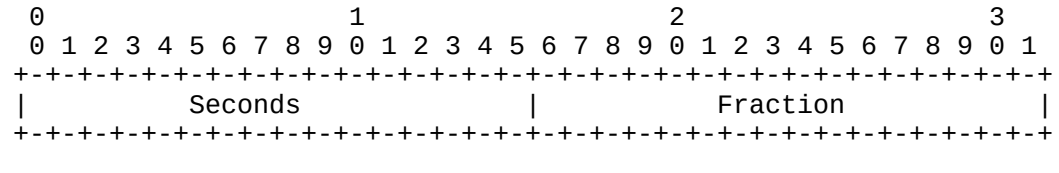
A client sends messages to each server with a poll interval of 2^tau seconds, as determined by the poll exponent tau. In NTPv4, tau ranges from 4 (16 s) to 17 (36 h). The value of tau is determined by the clock discipline algorithm to match the loop-time constant T_c = 2^tau. In client/server mode, the server responds immediately; however, in symmetric modes, each of two peers manages tau as a function of current system offset and system jitter, so they may not agree with the same value. It is important that the dynamic behavior of the clock discipline algorithm be carefully controlled in order to maintain stability in the NTP subnet at large. This requires that

the peers agree on a common tau equal to the minimum poll exponent of both peers. The NTP protocol includes provisions to properly negotiate this value.

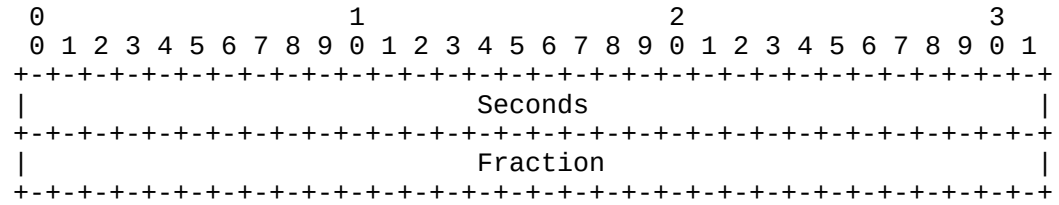
The implementation model includes some means to set and adjust the system clock. The operating system is assumed to provide two functions: one to set the time directly, for example, the Unix settimeofday() function, and another to adjust the time in small increments advancing or retarding the time by a designated amount, for example, the Unix adjtime() function. In this and following references, parentheses following a name indicate reference to a function rather than a simple variable. In the intended design the clock discipline process uses the adjtime() function if the adjustment is less than a designated threshold, and the settimeofday() function if above the threshold. The manner in which this is done and the value of the threshold as described in Section 10.

6. Data Types

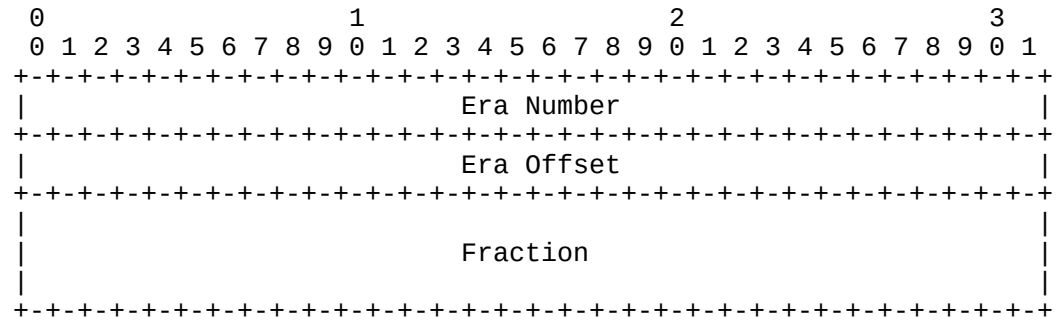
All NTP time values are represented in twos-complement format, with bits numbered in big-endian (as described in Appendix A of [RFC0791]) fashion from zero starting at the left, or high-order, position. There are three NTP time formats, a 128-bit date format, a 64-bit timestamp format, and a 32-bit short format, as shown in Figure 3. The 128-bit date format is used where sufficient storage and word size are available. It includes a 64-bit signed seconds field spanning 584 billion years and a 64-bit fraction field resolving .05 attosecond (i.e., 0.5e-18). For convenience in mapping between formats, the seconds field is divided into a 32-bit Era Number field and a 32-bit Era Offset field. Eras cannot be produced by NTP directly, nor is there need to do so. When necessary, they can be derived from external means, such as the filesystem or dedicated hardware.



NTP Short Format



NTP Timestamp Format



NTP Date Format

Figure 3: NTP Time Formats

The 64-bit timestamp format is used in packet headers and other places with limited word size. It includes a 32-bit unsigned seconds field spanning 136 years and a 32-bit fraction field resolving 232 picoseconds. The 32-bit short format is used in delay and dispersion header fields where the full resolution and range of the other formats are not justified. It includes a 16-bit unsigned seconds field and a 16-bit fraction field.

In the date and timestamp formats, the prime epoch, or base date of era 0, is 0 h 1 January 1900 UTC, when all bits are zero. It should be noted that strictly speaking, UTC did not exist prior to 1 January 1972, but it is convenient to assume it has existed for all eternity, even if all knowledge of historic leap seconds has been lost. Dates are relative to the prime epoch; values greater than zero represent

times after that date; values less than zero represent times before it. Note that the Era Offset field of the date format and the Seconds field of the timestamp format have the same interpretation.

Timestamps are unsigned values, and operations on them produce a result in the same or adjacent eras. Era 0 includes dates from the prime epoch to some time in 2036, when the timestamp field wraps around and the base date for era 1 is established. In either format, a value of zero is a special case representing unknown or unsynchronized time. Figure 4 shows a number of historic NTP dates together with their corresponding Modified Julian Day (MJD), NTP era, and NTP timestamp.

Date	MJD	NTP Era	NTP Timestamp Era Offset	Epoch
1 Jan -4712	-2,400,001	-49	1,795,583,104	1st day Julian
1 Jan -1	-679,306	-14	139,775,744	2 BCE
1 Jan 0	-678,491	-14	171,311,744	1 BCE
1 Jan 1	-678,575	-14	202,939,144	1 CE
4 Oct 1582	-100,851	-3	2,873,647,488	Last day Julian
15 Oct 1582	-100,840	-3	2,874,597,888	First day Gregorian
31 Dec 1899	15019	-1	4,294,880,896	Last day NTP Era -1
1 Jan 1900	15020	0	0	First day NTP Era 0
1 Jan 1970	40,587	0	2,208,988,800	First day UNIX
1 Jan 1972	41,317	0	2,272,060,800	First day UTC
31 Dec 1999	51,543	0	3,155,587,200	Last day 20th Century
8 Feb 2036	64,731	1	63,104	First day NTP Era 1

Figure 4: Interesting Historic NTP Dates

Let p be the number of significant bits in the second fraction. The clock resolution is defined as 2^(-p), in seconds. In order to minimize bias and help make timestamps unpredictable to an intruder, the non-significant bits should be set to an unbiased random bit string. The clock precision is defined as the running time to read the system clock, in seconds. Note that the precision defined in this way can be larger or smaller than the resolution. The term rho, representing the precision used in the protocol, is the larger of the two.

The only arithmetic operation permitted on dates and timestamps is twos-complement subtraction, yielding a 127-bit or 63-bit signed result. It is critical that the first-order differences between two dates preserve the full 128-bit precision and the first-order differences between two timestamps preserve the full 64-bit precision. However, the differences are ordinarily small compared to the seconds span, so they can be converted to floating double format for further processing and without compromising the precision.

It is important to note that twos-complement arithmetic does not distinguish between signed and unsigned values (although comparisons can take sign into account); only the conditional branch instructions do. Thus, although the distinction is made between signed dates and unsigned timestamps, they are processed the same way. A perceived hazard with 64-bit timestamp calculations spanning an era, such as is possible in 2036, might result in over-run. In point of fact, if the client is set within 68 years of the server before the protocol is started, correct values are obtained even if the client and server are in adjacent eras.

Some time values are represented in exponent format, including the precision, time constant, and poll interval. These are in 8-bit signed integer format in log2 (log base 2) seconds. The only arithmetic operations permitted on them are increment and decrement. For the purpose of this document and to simplify the presentation, a reference to one of these variables by name means the exponentiated value, e.g., the poll interval is 1024 s, while reference by name and exponent means the actual value, e.g., the poll exponent is 10.

To convert system time in any format to NTP date and timestamp formats requires that the number of seconds s from the prime epoch to the system time be determined. To determine the integer era and timestamp given s,

era = s / 2^(32) and timestamp = s - era * 2^(32),

which works for positive and negative dates. To determine s given the era and timestamp,

s = era * 2^(32) + timestamp.

Converting between NTP and system time can be a little messy, and is beyond the scope of this document. Note that the number of days in era 0 is one more than the number of days in most other eras, and this won’t happen again until the year 2400 in era 3.

In the description of state variables to follow, explicit reference to integer type implies a 32-bit unsigned integer. This simplifies bounds checks, since only the upper limit needs to be defined. Without explicit reference, the default type is 64-bit floating double. Exceptions will be noted as necessary.

7. Data Structures

The NTP state machines are defined in the following sections. State variables are separated into classes according to their function in packet headers, peer and poll processes, the system process, and the clock discipline process. Packet variables represent the NTP header values in transmitted and received packets. Peer and poll variables represent the contents of the association for each server separately. System variables represent the state of the server as seen by its dependent clients. Clock discipline variables represent the internal workings of the clock discipline algorithm. An example is described in Appendix A.

7.1. Structure Conventions

In order to distinguish between different variables of the same name but used in different processes, the naming convention summarized in Figure 5 is adopted. A receive packet variable v is a member of the packet structure r with fully qualified name r.v. In a similar manner, x.v is a transmit packet variable, p.v is a peer variable, s.v is a system variable, and c.v is a clock discipline variable. There is a set of peer variables for each association; there is only one set of system and clock variables.

Name	Description
r.	receive packet header variable
x.	transmit packet header variable
p.	peer/poll variable
s.	system variable
c.	clock discipline variable

Figure 5: Prefix Conventions

7.2. Global Parameters

In addition to the variable classes, a number of global parameters are defined in this document, including those shown with values in Figure 6.

Name	Value	Description
PORT	123	NTP port number
VERSION	4	NTP version number
TOLERANCE	15e-6	frequency tolerance PHI (s/s)
MINPOLL	4	minimum poll exponent (16 s)
MAXPOLL	17	maximum poll exponent (36 h)
MAXDISP	16	maximum dispersion (16 s)
MINDISP	.005	minimum dispersion increment (s)
MAXDIST	1	distance threshold (1 s)
MAXSTRAT	16	maximum stratum number

Figure 6: Global Parameters

While these are the only global parameters needed for interoperability, a larger collection is necessary in any implementation. Appendix A.1.1 contains those used by the skeleton for the mitigation algorithms, clock discipline algorithm, and related implementation-dependent functions. Some of these parameter values are cast in stone, like the NTP port number assigned by the IANA and the version number assigned NTPv4 itself. Others, like the frequency tolerance (also called PHI), involve an assumption about the worst-case behavior of a system clock once synchronized and then allowed to drift when its sources have become unreachable. The minimum and maximum parameters define the limits of state variables as described in later sections of this document.

While shown with fixed values in this document, some implementations may make them variables adjustable by configuration commands. For instance, the reference implementation computes the value of PRECISION as log2 of the minimum time in several iterations to read the system clock.

7.3. Packet Header Variables

The most important state variables from an external point of view are the packet header variables described in Figure 7 and below. The NTP packet header consists of an integral number of 32-bit (4 octet) words in network byte order. The packet format consists of three components: the header itself, one or more optional extension fields, and an optional message authentication code (MAC). The header component is identical to the NTPv3 header and previous versions. The optional extension fields are used by the Autokey public key cryptographic algorithms described in [RFC5906]. The optional MAC is used by both Autokey and the symmetric key cryptographic algorithm described in this RFC.

Name	Formula	Description
leap	leap	leap indicator (LI)
version	version	version number (VN)
mode	mode	mode
stratum	stratum	stratum
poll	poll	poll exponent
precision	rho	precision exponent
rootdelay	delta_r	root delay
rootdisp	epsilon_r	root dispersion
refid	refid	reference ID
reftime	reftime	reference timestamp
org	T1	origin timestamp
rec	T2	receive timestamp
xmt	T3	transmit timestamp
dst	T4	destination timestamp
keyid	keyid	key ID
dgst	dgst	message digest

Figure 7: Packet Header Variables

The NTP packet is a UDP datagram [RFC0768]. Some fields use multiple words and others are packed in smaller fields within a word. The NTP packet header shown in Figure 8 has 12 words followed by optional extension fields and finally an optional message authentication code (MAC) consisting of the Key Identifier field and Message Digest field.

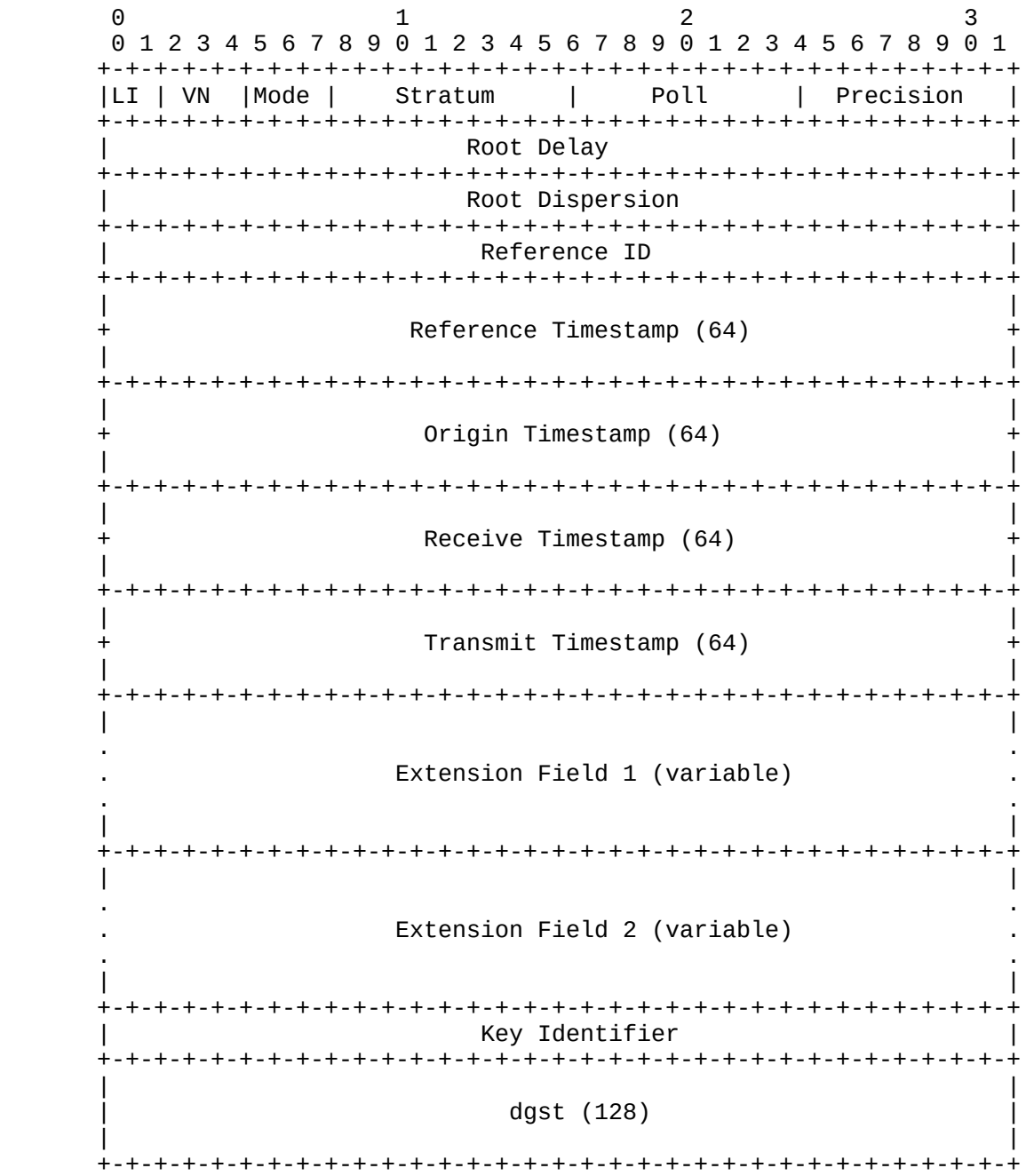


Figure 8: Packet Header Format

The extension fields are used to add optional capabilities, for example, the Autokey security protocol [RFC5906]. The extension field format is presented in order for the packet to be parsed without the knowledge of the extension field functions. The MAC is used by both Autokey and the symmetric key authentication scheme.

A list of the packet header variables is shown in Figure 7 and described in detail below. Except for a minor variation when using the IPv6 address family, these fields are backwards compatible with NTPv3. The packet header fields apply to both transmitted packets (x prefix) and received packets (r prefix). In Figure 8, the size of some multiple-word fields is shown in bits if not the default 32 bits. The basic header extends from the beginning of the packet to the end of the Transmit Timestamp field.

The fields and associated packet variables (in parentheses) are interpreted as follows:

LI Leap Indicator (leap): 2-bit integer warning of an impending leap second to be inserted or deleted in the last minute of the current month with values defined in Figure 9.

Value	Meaning
0	no warning
1	last minute of the day has 61 seconds
2	last minute of the day has 59 seconds
3	unknown (clock unsynchronized)

Figure 9: Leap Indicator

VN Version Number (version): 3-bit integer representing the NTP version number, currently 4.

Mode (mode): 3-bit integer representing the mode, with values defined in Figure 10.

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

Figure 10: Association Modes

Stratum (stratum): 8-bit integer representing the stratum, with values defined in Figure 11.

Value	Meaning
0	unspecified or invalid
1	primary server (e.g., equipped with a GPS receiver)
2-15	secondary server (via NTP)
16	unsynchronized
17-255	reserved

Figure 11: Packet Stratum

It is customary to map the stratum value 0 in received packets to MAXSTRAT (16) in the peer variable p.stratum and to map p.stratum values of MAXSTRAT or greater to 0 in transmitted packets. This allows reference clocks, which normally appear at stratum 0, to be conveniently mitigated using the same clock selection algorithms used for external sources (see Appendix A.5.5.1 for an example).

Poll: 8-bit signed integer representing the maximum interval between successive messages, in log2 seconds. Suggested default limits for minimum and maximum poll intervals are 6 and 10, respectively.

Precision: 8-bit signed integer representing the precision of the system clock, in log2 seconds. For instance, a value of -18 corresponds to a precision of about one microsecond. The precision can be determined when the service first starts up as the minimum time of several iterations to read the system clock.

Root Delay (rootdelay): Total round-trip delay to the reference clock, in NTP short format.

Root Dispersion (rootdisp): Total dispersion to the reference clock, in NTP short format.

Reference ID (refid): 32-bit code identifying the particular server or reference clock. The interpretation depends on the value in the stratum field. For packet stratum 0 (unspecified or invalid), this is a four-character ASCII [RFC1345] string, called the "kiss code", used for debugging and monitoring purposes. For stratum 1 (reference clock), this is a four-octet, left-justified, zero-padded ASCII string assigned to the reference clock. The authoritative list of Reference Identifiers is maintained by IANA; however, any string beginning with the ASCII character "X" is reserved for unregistered experimentation and development. The identifiers in Figure 12 have been used as ASCII identifiers:

ID	Clock Source
GOES	Geosynchronous Orbit Environment Satellite
GPS	Global Position System
GAL	Galileo Positioning System
PPS	Generic pulse-per-second
IRIG	Inter-Range Instrumentation Group
WWVB	LF Radio WWVB Ft. Collins, CO 60 kHz
DCF	LF Radio DCF77 Mainflingen, DE 77.5 kHz
HBG	LF Radio HBG Prangins, HB 75 kHz
MSF	LF Radio MSF Anthorn, UK 60 kHz
JJY	LF Radio JJY Fukushima, JP 40 kHz, Saga, JP 60 kHz
LORC	MF Radio LORAN C station, 100 kHz
TDF	MF Radio Allouis, FR 162 kHz
CHU	HF Radio CHU Ottawa, Ontario
WWV	HF Radio WWV Ft. Collins, CO
WWVH	HF Radio WWVH Kauai, HI
NIST	NIST telephone modem
ACTS	NIST telephone modem
USNO	USNO telephone modem
PTB	European telephone modem

Figure 12: Reference Identifiers

Above stratum 1 (secondary servers and clients): this is the reference identifier of the server and can be used to detect timing loops. If using the IPv4 address family, the identifier is the four-octet IPv4 address. If using the IPv6 address family, it is the

first four octets of the MD5 hash of the IPv6 address. Note that, when using the IPv6 address family on an NTPv4 server with a NTPv3 client, the Reference Identifier field appears to be a random value and a timing loop might not be detected.

Reference Timestamp: Time when the system clock was last set or corrected, in NTP timestamp format.

Origin Timestamp (org): Time at the client when the request departed for the server, in NTP timestamp format.

Receive Timestamp (rec): Time at the server when the request arrived from the client, in NTP timestamp format.

Transmit Timestamp (xmt): Time at the server when the response left for the client, in NTP timestamp format.

Destination Timestamp (dst): Time at the client when the reply arrived from the server, in NTP timestamp format.

Note: The Destination Timestamp field is not included as a header field; it is determined upon arrival of the packet and made available in the packet buffer data structure.

If the NTP has access to the physical layer, then the timestamps are associated with the beginning of the symbol after the start of frame. Otherwise, implementations should attempt to associate the timestamp to the earliest accessible point in the frame.

The MAC consists of the Key Identifier followed by the Message Digest. The message digest, or cryptosum, is calculated as in [RFC1321] over all NTP-header and optional extension fields, but not the MAC itself.

Extension Field n: See Section 7.5 for a description of the format of this field.

Key Identifier (keyid): 32-bit unsigned integer used by the client and server to designate a secret 128-bit MD5 key.

Message Digest (digest): 128-bit MD5 hash computed over the key followed by the NTP packet header and extensions fields (but not the Key Identifier or Message Digest fields).

It should be noted that the MAC computation used here differs from those defined in [RFC1305] and [RFC4330] but is consistent with how existing implementations generate a MAC.

7.4. The Kiss-o'-Death Packet

If the Stratum field is 0, which implies unspecified or invalid, the Reference Identifier field can be used to convey messages useful for status reporting and access control. These are called Kiss-o'-Death (KoD) packets and the ASCII messages they convey are called kiss codes. The KoD packets got their name because an early use was to tell clients to stop sending packets that violate server access controls. The kiss codes can provide useful information for an intelligent client, either NTPv4 or SNTPv4. Kiss codes are encoded in four-character ASCII strings that are left justified and zero filled. The strings are designed for character displays and log files. A list of the currently defined kiss codes is given in Figure 13. Recipients of kiss codes MUST inspect them and, in the following cases, take these actions:

a. For kiss codes DENY and RSTR, the client MUST demobilize any associations to that server and stop sending packets to that server;

b. For kiss code RATE, the client MUST immediately reduce its polling interval to that server and continue to reduce it each time it receives a RATE kiss code.

c. Kiss codes beginning with the ASCII character "X" are for unregistered experimentation and development and MUST be ignored if not recognized.

d. Other than the above conditions, KoD packets have no protocol significance and are discarded after inspection.

Code	Meaning
ACST	The association belongs to a unicast server.
AUTH	Server authentication failed.
AUTO	Autokey sequence failed.
BCST	The association belongs to a broadcast server.
CRYP	Cryptographic authentication or identification failed.
DENY	Access denied by remote server.
DROP	Lost peer in symmetric mode.
RSTR	Access denied due to local policy.
INIT	The association has not yet synchronized for the first time.
MCST	The association belongs to a dynamically discovered server.
NKEY	No key found. Either the key was never installed or is not trusted.
RATE	Rate exceeded. The server has temporarily denied access because the client exceeded the rate threshold.
RMOT	Alteration of association from a remote host running ntpdc.
STEP	A step change in system time has occurred, but the association has not yet resynchronized.

Figure 13: Kiss Codes

The Receive Timestamp and the Transmit Timestamp (set by the server) are undefined when in a KoD packet and MUST NOT be relied upon to have valid values and MUST be discarded.

7.5. NTP Extension Field Format

In NTPv4, one or more extension fields can be inserted after the header and before the MAC, which is always present when an extension field is present. Other than defining the field format, this document makes no use of the field contents. An extension field contains a request or response message in the format shown in Figure 14.

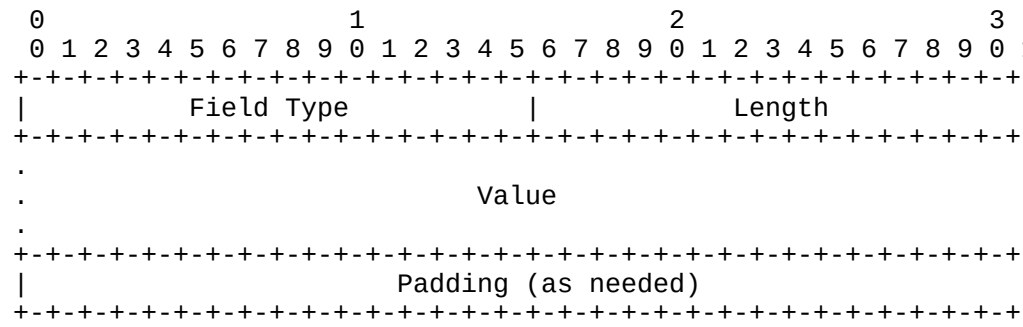


Figure 14: Extension Field Format

All extension fields are zero-padded to a word (four octets) boundary. The Field Type field is specific to the defined function and is not elaborated here. While the minimum field length containing required fields is four words (16 octets), a maximum field length remains to be established.

The Length field is a 16-bit unsigned integer that indicates the length of the entire extension field in octets, including the Padding field.

8. On-Wire Protocol

The heart of the NTP on-wire protocol is the core mechanism that exchanges time values between servers, peers, and clients. It is inherently resistant to lost or duplicate packets. Data integrity is provided by the IP and UDP checksums. No flow control or retransmission facilities are provided or necessary. The protocol uses timestamps, which are either extracted from packet headers or struck from the system clock upon the arrival or departure of a packet. Timestamps are precision data and should be restruck in the case of link-level retransmission and corrected for the time to compute a MAC in transmit.

NTP messages make use of two different communication modes, one-to-one and one-to-many, commonly referred to as unicast and broadcast. For the purposes of this document, the term broadcast is interpreted as any available one-to-many mechanism. For IPv4, this equates to either IPv4 broadcast or IPv4 multicast. For IPv6, this equates to IPv6 multicast. For this purpose, IANA has allocated the IPv4 multicast address 224.0.1.1 and the IPv6 multicast address ending :101, with the prefix determined by scoping rules. Any other non-allocated multicast address may also be used in addition to these allocated multicast addresses.

The on-wire protocol uses four timestamps numbered t1 through t4 and three state variables org, rec, and xmt, as shown in Figure 15. This figure shows the most general case where each of two peers, A and B, independently measure the offset and delay relative to the other. For purposes of illustration, the packet timestamps are shown in lowercase, while the state variables are shown in uppercase. The state variables are copied from the packet timestamps upon arrival or departure of a packet.

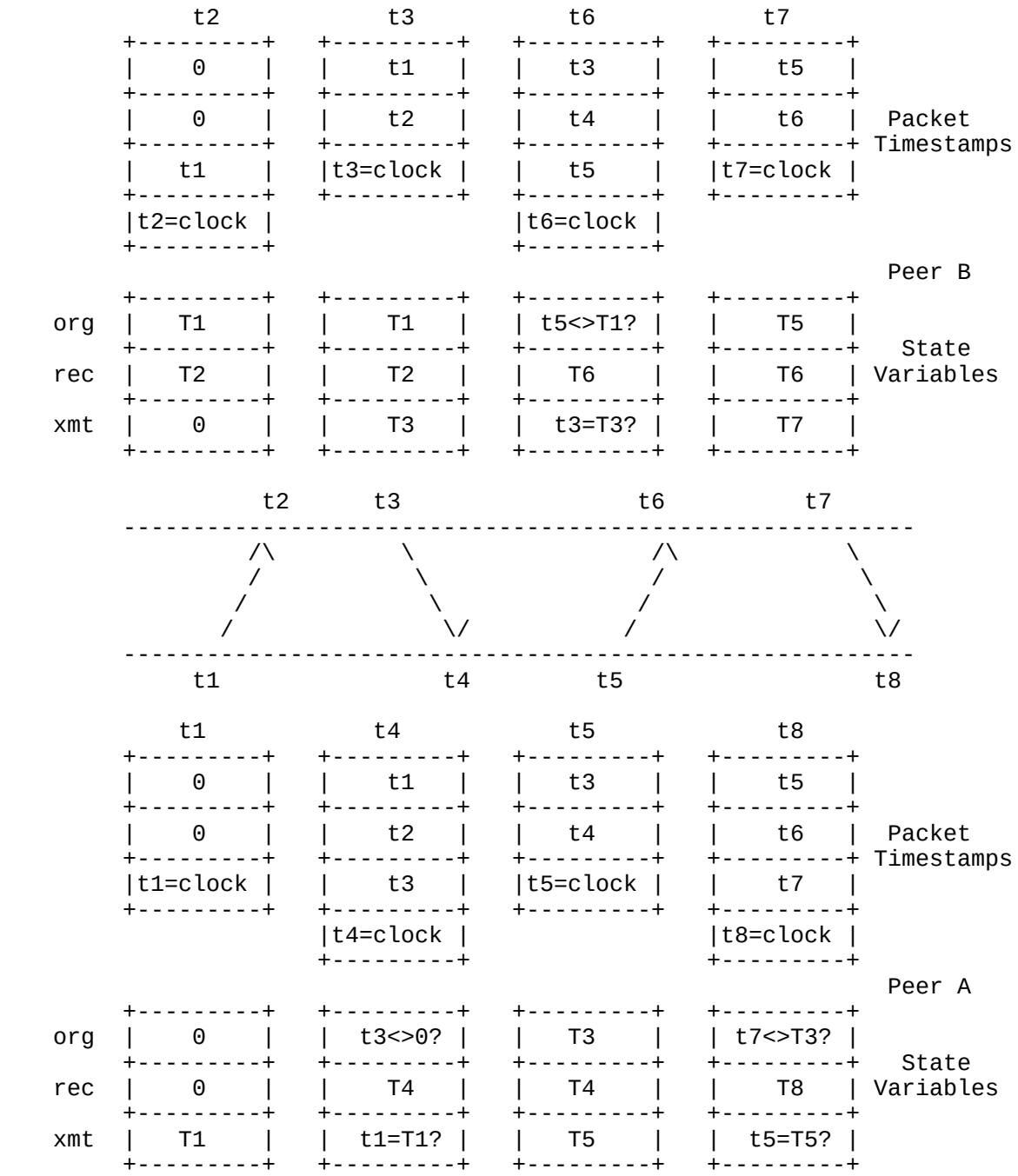


Figure 15: On-Wire Protocol

In the figure, the first packet transmitted by A contains only the origin timestamp t1, which is then copied to T1. B receives the packet at t2 and copies t1 to T1 and the receive timestamp t2 to T2. At this time or some time later at t3, B sends a packet to A containing t1 and t2 and the transmit timestamp t3. All three timestamps are copied to the corresponding state variables. A receives the packet at t4 containing the three timestamps t1, t2, and t3 and the destination timestamp t4. These four timestamps are used to compute the offset and delay of B relative to A, as described below.

Before the xmt and org state variables are updated, two sanity checks are performed in order to protect against duplicate, bogus, or replayed packets. In the exchange above, a packet is duplicate or replay if the transmit timestamp t3 in the packet matches the org state variable T3. A packet is bogus if the origin timestamp t1 in the packet does not match the xmt state variable T1. In either of these cases, the state variables are updated, then the packet is discarded. To protect against replay of the last transmitted packet, the xmt state variable is set to zero immediately after a successful bogus check.

The four most recent timestamps, T1 through T4, are used to compute the offset of B relative to A

theta = T(B) - T(A) = 1/2 * [(T2-T1) + (T3-T4)]

and the round-trip delay

delta = T(ABA) = (T4-T1) - (T3-T2).

Note that the quantities within parentheses are computed from 64-bit unsigned timestamps and result in signed values with 63 significant bits plus sign. These values can represent dates from 68 years in the past to 68 years in the future. However, the offset and delay are computed as sums and differences of these values, which contain 62 significant bits and two sign bits, so they can represent unambiguous values from 34 years in the past to 34 years in the future. In other words, the time of the client must be set within 34 years of the server before the service is started. This is a fundamental limitation with 64-bit integer arithmetic.

In implementations where floating double arithmetic is available, the first-order differences can be converted to floating double and the second-order sums and differences computed in that arithmetic. Since

the second-order terms are typically very small relative to the timestamp magnitudes, there is no loss in significance, yet the unambiguous range is restored from 34 years to 68 years.

In some scenarios where the initial frequency offset of the client is relatively large and the actual propagation time small, it is possible for the delay computation to become negative. For instance, if the frequency difference is 100 ppm and the interval T4-T1 is 64 s, the apparent delay is -6.4 ms. Since negative values are misleading in subsequent computations, the value of delta should be clamped not less than s.rho, where s.rho is the system precision described in Section 11.1, expressed in seconds.

The discussion above assumes the most general case where two symmetric peers independently measure the offsets and delays between them. In the case of a stateless server, the protocol can be simplified. A stateless server copies T3 and T4 from the client packet to T1 and T2 of the server packet and tacks on the transmit timestamp T3 before sending it to the client. Additional details for filling in the remaining protocol fields are given in a Section 9 and following sections and in the appendix.

Note that the on-wire protocol as described resists replay of a server response packet. However, it does not resist replay of the client request packet, which would result in a server reply packet with new values of T2 and T3 and result in incorrect offset and delay. This vulnerability can be avoided by setting the xmt state variable to zero after computing the offset and delay.

9. Peer Process

The process descriptions to follow include a listing of the important state variables followed by an overview of the process operations implemented as routines. Frequent reference is made to the skeleton in the appendix. The skeleton includes C-language fragments that describe the functions in more detail. It includes the parameters, variables, and declarations necessary for a conforming NTPv4 implementation. However, many additional variables and routines may be necessary in a working implementation.

The peer process is called upon arrival of a server or peer packet. It runs the on-wire protocol to determine the clock offset and round-trip delay and computes statistics used by the system and poll processes. Peer variables are instantiated in the association data structure when the structure is initialized and updated by arriving packets. There is a peer process, poll process, and association process for each server.

9.1. Peer Process Variables

Figures 16, 17, 18, and 19 summarize the common names, formula names, and a short description of the peer variables. The common names and formula names are interchangeable; formula names are intended to increase readability of equations in this specification. Unless noted otherwise, all peer variables have assumed prefix p.

Name	Formula	Description
srcaddr	srcaddr	source address
srcport	srcport	source port
dstaddr	dstaddr	destination address
dstport	destport	destination port
keyid	keyid	key identifier key ID

Figure 16: Peer Process Configuration Variables

Name	Formula	Description
leap	leap	leap indicator
version	version	version number
mode	mode	mode
stratum	stratum	stratum
ppoll	ppoll	peer poll exponent
rootdelay	delta_r	root delay
rootdisp	epsilon_r	root dispersion
refid	refid	reference ID
reftime	reftime	reference timestamp

Figure 17: Peer Process Packet Variables

Name	Formula	Description
org	T1	origin timestamp
rec	T2	receive timestamp
xmt	T3	transmit timestamp
t	t	packet time

Figure 18: Peer Process Timestamp Variables

Name	Formula	Description
offset	theta	clock offset
delay	delta	round-trip delay
disp	epsilon	dispersion
jitter	psi	jitter
filter	filter	clock filter
tp	t_p	filter time

Figure 19: Peer Process Statistics Variables

The following configuration variables are normally initialized when the association is mobilized, either from a configuration file or upon the arrival of the first packet for an unknown association.

srcaddr: IP address of the remote server or reference clock. This becomes the destination IP address in packets sent from this association.

srcport: UDP port number of the server or reference clock. This becomes the destination port number in packets sent from this association. When operating in symmetric modes (1 and 2), this field must contain the NTP port number PORT (123) assigned by the IANA. In other modes, it can contain any number consistent with local policy.

dstaddr: IP address of the client. This becomes the source IP address in packets sent from this association.

dstport: UDP port number of the client, ordinarily the NTP port number PORT (123) assigned by the IANA. This becomes the source port number in packets sent from this association.

keyid: Symmetric key ID for the 128-bit MD5 key used to generate and verify the MAC. The client and server or peer can use different values, but they must map to the same key.

The variables defined in Figure 17 are updated from the packet header as each packet arrives. They are interpreted in the same way as the packet variables of the same names. It is convenient for later processing to convert the NTP short format packet values r.rootdelay and r.rootdisp to floating doubles as peer variables.

The variables defined in Figure 18 include the timestamps exchanged by the on-wire protocol in Section 8. The t variable is the seconds counter c.t associated with these values. The c.t variable is maintained by the clock-adjust process described in Section 12. It

counts the seconds since the service was started. The variables defined in Figure 19 include the statistics computed by the clock_filter() routine described in Section 10. The tp variable is the seconds counter associated with these values.

9.2. Peer Process Operations

The receive routine defines the process flow upon the arrival of a packet. An example is described by the receive() routine in Appendix A.5.1. There is no specific method required for access control, although it is recommended that implementations include such a scheme, which is similar to many others now in widespread use. The access() routine in Appendix A.5.4 describes a method of implementing access restrictions using an access control list (ACL). Format checks require correct field length and alignment, acceptable version number (1-4), and correct extension field syntax, if present.

There is no specific requirement for authentication; however, if authentication is implemented, then the MD5-keyed hash algorithm described in [RFC1321] must be supported.

Next, the association table is searched for matching source address and source port, for example, using the find_assoc() routine in Appendix A.5.1. Figure 20 is a dispatch table where the columns correspond to the packet mode and rows correspond to the association mode. The intersection of the association and packet modes dispatches processing to one of the following steps.

	Packet Mode				
Association Mode	1	2	3	4	5
No Association 0	NEWPS	DSCRD	FXMIT	MANY	NEWBC
Symm. Active 1	PROC	PROC	DSCRD	DSCRD	DSCRD
Symm. Passive 2	PROC	ERR	DSCRD	DSCRD	DSCRD
Client 3	DSCRD	DSCRD	DSCRD	PROC	DSCRD
Server 4	DSCRD	DSCRD	DSCRD	DSCRD	DSCRD
Broadcast 5	DSCRD	DSCRD	DSCRD	DSCRD	DSCRD
Bcast Client 6	DSCRD	DSCRD	DSCRD	DSCRD	PROC

Figure 20: Peer Dispatch Table

DSCRD. This indicates a non-fatal violation of protocol as the result of a programming error, long-delayed packet, or replayed packet. The peer process discards the packet and exits.

ERR. This indicates a fatal violation of protocol as the result of a programming error, long-delayed packet, or replayed packet. The peer process discards the packet, demobilizes the symmetric passive association, and exits.

FXMIT. This indicates a client (mode 3) packet matching no association (mode 0). If the destination address is not a broadcast address, the server constructs a server (mode 4) packet and returns it to the client without retaining state. The server packet header is constructed. An example is described by the fast_xmit() routine in Appendix A.5.3. The packet header is assembled from the receive packet and system variables as shown in Figure 21. If the s.rootdelay and s.rootdisp system variables are stored in floating double, they must be converted to NTP short format first.

Packet Variable -->	Variable
r.leap -->	p.leap
r.mode -->	p.mode
r.stratum -->	p.stratum
r.poll -->	p.ppoll
r.rootdelay -->	p.rootdelay
r.rootdisp -->	p.rootdisp
r.refid -->	p.refid
r.reftime -->	p.reftime
r.keyid -->	p.keyid

Figure 21: Receive Packet Header

Note that, if authentication fails, the server returns a special message called a crypto-NAK. This message includes the normal NTP header data shown in Figure 8, but with a MAC consisting of four octets of zeros. The client MAY accept or reject the data in the message. After these actions, the peer process exits.

If the destination address is a multicast address, the sender is operating in manycast client mode. If the packet is valid and the server stratum is less than the client stratum, the server sends an ordinary server (mode 4) packet, but one which uses its unicast destination address. A crypto-NAK is not sent if authentication fails. After these actions, the peer process exits.

MANY: This indicates a server (mode 4) packet matching no association. Ordinarily, this can happen only as the result of a manycast server reply to a previously sent multicast client packet.

If the packet is valid, an ordinary client (mode 3) association is mobilized and operation continues as if the association was mobilized by the configuration file.

NEWBC. This indicates a broadcast (mode 5) packet matching no association. The client mobilizes either a client (mode 3) or broadcast client (mode 6) association. Examples are shown in the mobilize() and clear() routines in Appendix A.2. Then, the packet is validated and the peer variables initialized. An example is provided by the packet() routine in Appendix A.5.1.1.

If the implementation supports no additional security or calibration functions, the association mode is set to broadcast client (mode 6) and the peer process exits. Implementations supporting public key authentication MAY run the Autokey or equivalent security protocol. Implementations SHOULD set the association mode to 3 and run a short client/server exchange to determine the propagation delay. Following the exchange, the association mode is set to 6 and the peer process continues in listen-only mode. Note the distinction between a mode-6 packet, which is reserved for the NTP monitor and control functions, and a mode-6 association.

NEWPS. This indicates a symmetric active (mode 1) packet matching no association. The client mobilizes a symmetric passive (mode 2) association. An example is shown in the mobilize() and clear() routines in Appendix A.2. Processing continues in the PROC section below.

PROC. This indicates a packet matching an existing association. The packet timestamps are carefully checked to avoid invalid, duplicate, or bogus packets. Additional checks are summarized in Figure 22. Note that all packets, including a crypto-NAK, are considered valid only if they survive these tests.

Packet Type	Description
1 duplicate packet	The packet is at best an old duplicate or at worst a replay by a hacker. This can happen in symmetric modes if the poll intervals are uneven.
2 bogus packet	
3 invalid	One or more timestamp fields are invalid. This normally happens in symmetric modes when one peer sends the first packet to the other and before the other has received its first reply.
4 access denied	The access controls have blacklisted the source.
5 authentication failure	The cryptographic message digest does not match the MAC.
6 unsynchronized	The server is not synchronized to a valid source.
7 bad header data	One or more header fields are invalid.

Figure 22: Packet Error Checks

Processing continues by copying the packet variables to the peer variables as shown in Figure 21. An example is described by the packet() routine in Appendix A.5.1.1. The receive() routine implements tests 1-5 in Figure 22; the packet() routine implements tests 6-7. If errors are found, the packet is discarded and the peer process exits.

The on-wire protocol calculates the clock offset theta and round-trip delay delta from the four most recent timestamps as described in Section 8. While it is, in principle, possible to do all calculations except the first-order timestamp differences in fixed-point arithmetic, it is much easier to convert the first-order differences to floating doubles and do the remaining calculations in that arithmetic, and this will be assumed in the following description.

Next, the 8-bit p.reach shift register in the poll process described in Section 13 is used to determine whether the server is reachable and the data are fresh. The register is shifted left by one bit when a packet is sent and the rightmost bit is set to zero. As valid packets arrive, the rightmost bit is set to one. If the register contains any nonzero bits, the server is considered reachable; otherwise, it is unreachable. Since the peer poll interval might

have changed since the last packet, the host poll interval is reviewed. An example is provided by the poll_update() routine in Appendix A.5.7.2.

The dispersion statistic epsilon(t) represents the maximum error due to the frequency tolerance and time since the last packet was sent. It is initialized

epsilon(t_0) = r.rho + s.rho + PHI * (T4-T1)

when the measurement is made at t_0 according to the seconds counter. Here, r.rho is the packet precision described in Section 7.3 and s.rho is the system precision described in Section 11.1, both expressed in seconds. These terms are necessary to account for the uncertainty in reading the system clock in both the server and the client.

The dispersion then grows at constant rate PHI; in other words, at time t, epsilon(t) = epsilon(t_0) + PHI * (t-t_0). With the default value PHI = 15 ppm, this amounts to about 1.3 s per day. With this understanding, the argument t will be dropped and the dispersion represented simply as epsilon. The remaining statistics are computed by the clock filter algorithm described in the next section.

10. Clock Filter Algorithm

The clock filter algorithm is part of the peer process. It grooms the stream of on-wire data to select the samples most likely to represent accurate time. The algorithm produces the variables shown in Figure 19, including the offset (theta), delay (delta), dispersion (epsilon), jitter (psi), and time of arrival (t). These data are used by the mitigation algorithms to determine the best and final offset used to discipline the system clock. They are also used to determine the server health and whether it is suitable for synchronization.

The clock filter algorithm saves the most recent sample tuples (theta, delta, epsilon, t) in the filter structure, which functions as an 8-stage shift register. The tuples are saved in the order that packets arrive. Here, t is the packet time of arrival according to the seconds counter and should not be confused with the peer variable tp.

The following scheme is used to ensure sufficient samples are in the filter and that old stale data are discarded. Initially, the tuples of all stages are set to the dummy tuple (0, MAXDISP, MAXDISP, 0). As valid packets arrive, tuples are shifted into the filter causing old tuples to be discarded, so eventually only valid tuples remain.

If the three low-order bits of the reach register are zero, indicating three poll intervals have expired with no valid packets received, the poll process calls the clock filter algorithm with a dummy tuple just as if the tuple had arrived from the network. If this persists for eight poll intervals, the register returns to the initial condition.

In the next step, the shift register stages are copied to a temporary list and the list sorted by increasing delta. Let i index the stages starting with the lowest delta. If the first tuple epoch t_0 is not later than the last valid sample epoch tp, the routine exits without affecting the current peer variables. Otherwise, let epsilon_i be the dispersion of the ith entry, then

$$\text{epsilon} = \sum_{i=0}^{i=n-1} \frac{\text{epsilon\_i}}{2^{(i+1)}}$$

is the peer dispersion p.disp. Note the overload of epsilon, whether input to the clock filter or output, the meaning should be clear from context.

The observer should note (a) if all stages contain the dummy tuple with dispersion MAXDISP, the computed dispersion is a little less than 16 s, (b) each time a valid tuple is shifted into the register, the dispersion drops by a little less than half, depending on the valid tuples dispersion, and (c) after the fourth valid packet the dispersion is usually a little less than 1 s, which is the assumed value of the MAXDIST parameter used by the selection algorithm to determine whether or not the peer variables are acceptable.

Let the first stage offset in the sorted list be theta_0; then, for the other stages in any order, the jitter is the RMS average

$$\text{psi} = \frac{1}{(n-1)} * \left(\sum_{j=1}^{n-1} (\text{theta\_0} - \text{theta\_j})^2 \right)^{1/2}$$

where n is the number of valid tuples in the filter (n > 1). In order to ensure consistency and avoid divide exceptions in other

computations, the psi is bounded from below by the system precision s.rho expressed in seconds. While not in general considered a major factor in ranking server quality, jitter is a valuable indicator of fundamental timekeeping performance and network congestion state. Of particular importance to the mitigation algorithms is the peer synchronization distance, which is computed from the delay and dispersion.

lambda = (delta / 2) + epsilon.

Note that epsilon and therefore lambda increase at rate PHI. The lambda is not a state variable, since lambda is recalculated at each use. It is a component of the root synchronization distance used by the mitigation algorithms as a metric to evaluate the quality of time available from each server.

It is important to note that, unlike NTPv3, NTPv4 associations do not show a timeout condition by setting the stratum to 16 and leap indicator to 3. The association variables retain the values determined upon arrival of the last packet. In NTPv4, lambda increases with time, so eventually the synchronization distance exceeds the distance threshold MAXDIST, in which case the association is considered unfit for synchronization.

An example implementation of the clock filter algorithm is shown in the clock_filter() routine of Appendix A.5.2.

11. System Process

As each new sample (theta, delta, epsilon, jitter, t) is produced by the clock filter algorithm, all peer processes are scanned by the mitigation algorithms consisting of the selection, cluster, combine, and clock discipline algorithms in the system process. The selection algorithm scans all associations and casts off the falsetickers, which have demonstrably incorrect time, leaving the truechimers as result. In a series of rounds, the cluster algorithm discards the association statistically furthest from the centroid until a specified minimum number of survivors remain. The combine algorithm produces the best and final statistics on a weighted average basis. The final offset is passed to the clock discipline algorithm to steer the system clock to the correct time.

The cluster algorithm selects one of the survivors as the system peer. The associated statistics (theta, delta, epsilon, jitter, t) are used to construct the system variables inherited by dependent servers and clients and made available to other applications running on the same machine.

11.1. System Process Variables

Figure 23 summarizes the common names, formula names, and a short description of each system variable. Unless noted otherwise, all variables have assumed prefix s.

Name	Formula	Description
t	t	update time
p	p	system peer identifier
leap	leap	leap indicator
stratum	stratum	stratum
precision	rho	precision
offset	THETA	combined offset
jitter	PSI	combined jitter
rootdelay	DELTA	root delay
rootdisp	EPSILON	root dispersion
v	v	survivor list
refid	refid	reference ID
reftime	reftime	reference time
NMIN	3	minimum survivors
CMIN	1	minimum candidates

Figure 23: System Process Variables

Except for the t, p, offset, and jitter variables and the NMIN and CMIN constants, the variables have the same format and interpretation as the peer variables of the same name. The NMIN and CMIN parameters are used by the selection and cluster algorithms described in the next section.

The t variable is the seconds counter at the time of the last update. An example is shown by the clock_update() routine in Appendix A.5.5.4. The p variable is the system peer identifier determined by the cluster() routine in Section 11.2.2. The precision variable has the same format as the packet variable of the same name. The precision is defined as the larger of the resolution and time to read the clock, in log2 units. For instance, the precision of a mains-frequency clock incrementing at 60 Hz is 16 ms, even when the system clock hardware representation is to the nanosecond.

The offset and jitter variables are determined by the combine algorithm in Section 11.2.3. These values represent the best and final offset and jitter used to discipline the system clock.

Initially, all variables are cleared to zero, then the leap is set to 3 (unsynchronized) and stratum is set to MAXSTRAT (16). Remember that MAXSTRAT is mapped to zero in the transmitted packet.

11.2. System Process Operations

Figure 24 summarizes the system process operations performed by the clock select routine. The selection algorithm described in Section 11.2.1 produces a majority clique of presumed correct candidates (truechimers) based on agreement principles. The cluster algorithm described in Section 11.2.2 discards outliers to produce the most accurate survivors. The combine algorithm described in Section 11.2.3 provides the best and final offset for the clock discipline algorithm. An example is described in Appendix A.5.5.6. If the selection algorithm cannot produce a majority clique, or if it cannot produce at least CMIN survivors, the system process exits without disciplining the system clock. If successful, the cluster algorithm selects the statistically best candidate as the system peer and its variables are inherited as the system variables.

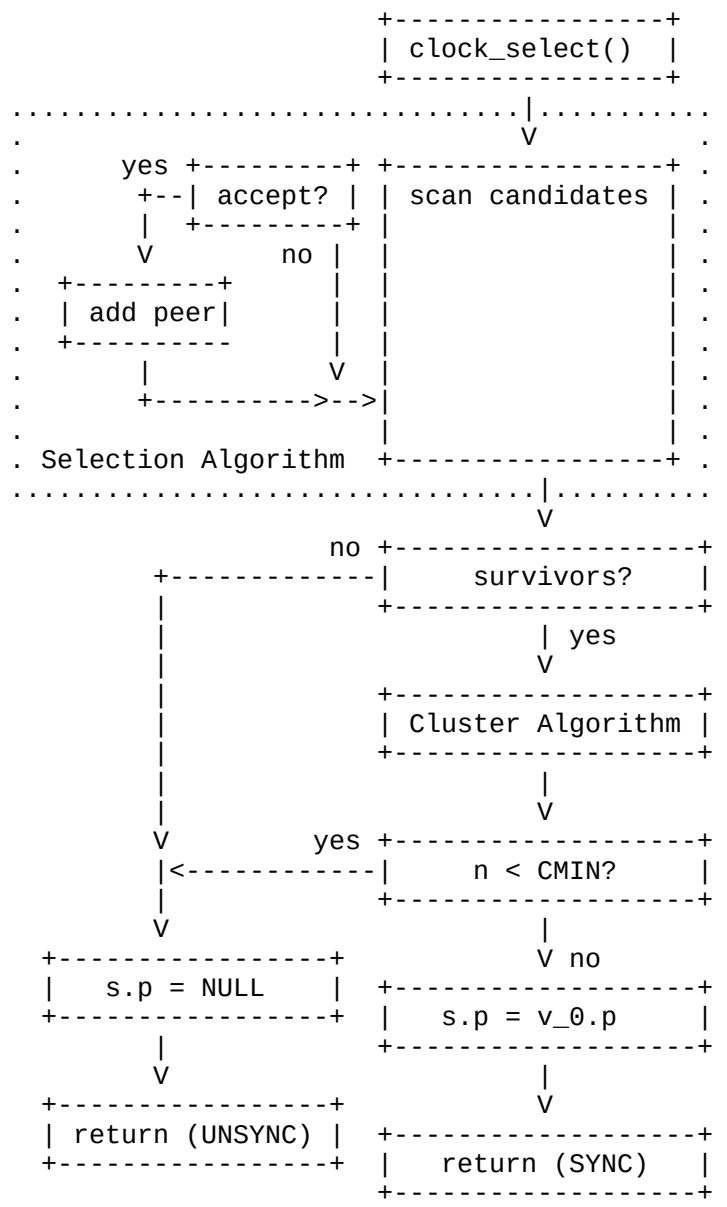


Figure 24: Clock Select Routine

11.2.1. Selection Algorithm

Note that the selection and cluster algorithms are described separately, but combined in the code skeleton. The selection algorithm operates to find an intersection interval containing a majority clique of truechimers using Byzantine agreement principles originally proposed by Marzullo [ref6], but modified to improve accuracy. An overview of the algorithm is given below and described in the first half of the clock_select() routine in Appendix A.5.5.1.

First, those servers that are unusable according to the rules of the protocol are detected and discarded as shown by the accept() routine in Appendix A.5.5.3. Next, a set of tuples (p, type, edge) is generated for the remaining candidates. Here, p is the association identifier and type identifies the upper (+1), middle (0), and lower (-1) endpoints of a correctness interval centered on theta for that candidate. This results in three tuples, lowpoint (p, -1, theta - lambda), midpoint (p, 0, theta), and highpoint (p, +1, theta + lambda), where lambda is the root synchronization distance. An example of this calculation is shown by the rootdist() routine in Appendix A.5.1.1. The steps of the algorithm are:

1. For each of m associations, place three tuples as defined above on the candidate list.

2. Sort the tuples on the list by the edge component. Order the lowpoint, midpoint, and highpoint of these intervals from lowest to highest. Set the number of falsetickers f = 0.

3. Set the number of midpoints d = 0. Set c = 0. Scan from lowest endpoint to highest. Add one to c for every lowpoint, subtract one for every highpoint, add one to d for every midpoint. If c >= m - f, stop; set l = current lowpoint.

4. Set c = 0. Scan from highest endpoint to lowest. Add one to c for every highpoint, subtract one for every lowpoint, add one to d for every midpoint. If c >= m - f, stop; set u = current highpoint.

5. Is d = f and l < u? If yes, then follow step 5A; else, follow step 5B.

5A. Success: the intersection interval is [l, u].

5B. Add one to f. Is f < (m / 2)? If yes, then go to step 3 again. If no, then go to step 6.

6. Failure; a majority clique could not be found. There are no suitable candidates to discipline the system clock.

The algorithm is described in detail in Appendix A.5.5.1. Note that it starts with the assumption that there are no falsetickers (f = 0) and attempts to find a non-empty intersection interval containing the midpoints of all correct servers, i.e., truechimers. If a non-empty interval cannot be found, it increases the number of assumed falsetickers by one and tries again. If a non-empty interval is found and the number of falsetickers is less than the number of truechimers, a majority clique has been found and the midpoint of each truechimer (theta) represents the candidates available to the cluster algorithm.

If a majority clique is not found, or if the number of truechimers is less than CMIN, there are insufficient candidates to discipline the system clock. CMIN defines the minimum number of servers consistent with the correctness requirements. Suspicious operators would set CMIN to ensure multiple redundant servers are available for the algorithms to mitigate properly. However, for historic reasons the default value for CMIN is one.

11.2.2. Cluster Algorithm

The candidates of the majority clique are placed on the survivor list v in the form of tuples (p, theta_p, psi_p, lambda_p), where p is an association identifier, theta_p, psi_p, and stratum_p the current offset, jitter and stratum of association p, respectively, and lambda_p is a merit factor equal to stratum_p * MAXDIST + lambda, where lambda is the root synchronization distance for association p. The list is processed by the cluster algorithm below. An example is shown by the second half of the clock_select() algorithm in Appendix A.5.5.1.

1. Let (p, theta_p, psi_p, lambda_p) represent a survivor candidate.

2. Sort the candidates by increasing lambda_p. Let n be the number of candidates and NMIN the minimum required number of survivors.

3. For each candidate, compute the selection jitter psi_s:

$$psi_s = \left(\frac{1}{n-1} \sum_{j=1}^{n-1} (theta_s - theta_j)^2 \right)^{1/2}$$

4. Select psi_max as the candidate with maximum psi_s.

5. Select psi_min as the candidate with minimum psi_p.

6. Is psi_max < psi_min or n <= NMIN? If yes, follow step 6A; otherwise, follow step 6B.

6A. Done. The remaining candidates on the survivor list are ranked in the order of preference. The first entry on the list represents the system peer; its variables are used later to update the system variables.

6B. Delete the outlier candidate with psi_max; reduce n by one and go back to step 3.

The algorithm operates in a series of rounds where each round discards the statistical outlier with maximum selection jitter psi_s. However, if psi_s is less than the minimum peer jitter psi_p, no improvement is possible by discarding outliers. This and the minimum number of survivors represent the terminating conditions of the algorithm. Upon termination, the final value of psi_max is saved as the system selection jitter PSI_s for use later.

11.2.3. Combine Algorithm

The clock combine route processes the remaining survivors to produce the best and final data for the clock discipline algorithm. The routine processes peer offset and jitter statistics to produce the combined system offset THETA and system peer jitter PSI_p, where each server statistic is weighted by the reciprocal of the root synchronization distance and the result normalized. An example is shown by the clock_combine() routine in Appendix A.5.5.5

The combined THETA is passed to the clock update routine. The first candidate on the survivor list is nominated as the system peer with identifier p. The system peer jitter PSI_p is a component of the system jitter PSI. It is used along with the selection jitter PSI_s to produce the system jitter:

PSI = [(PSI_s)^2 + (PSI_p)^2]^1/2

Each time an update is received from the system peer, the clock update routine is called. By rule, an update is discarded if its time of arrival p.t is not strictly later than the last update used s.t. The labels IGNOR, PANIC, ADJ, and STEP refer to return codes from the local clock routine described in the next section.

IGNORE means the update has been ignored as an outlier. PANIC means the offset is greater than the panic threshold PANICT (1000 s) and SHOULD cause the program to exit with a diagnostic message to the

system log. STEP means the offset is less than the panic threshold, but greater than the step threshold STEPT (125 ms). In this case, the clock is stepped to the correct offset, but since this means all peer data have been invalidated, all associations MUST be reset and the client begins as at initial start.

ADJ means the offset is less than the step threshold and thus a valid update. In this case, the system variables are updated from the peer variables as shown in Figure 25.

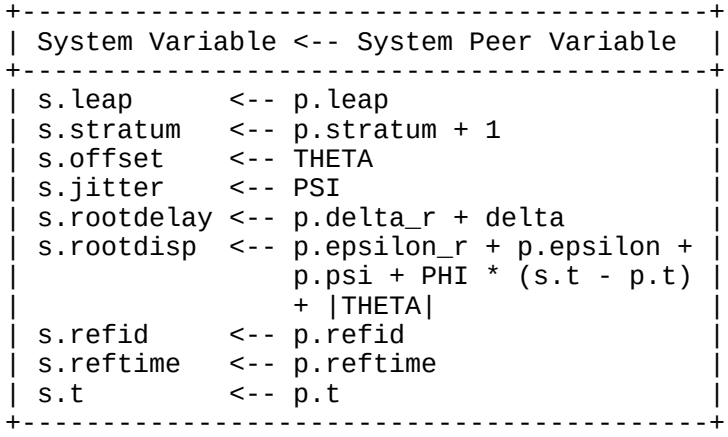


Figure 25: System Variables Update

There is an important detail not shown. The dispersion increment (p.epsilon + p.psi + PHI * (s.t - p.t) + |THETA|) is bounded from below by MINDISP. In subnets with very fast processors and networks and very small delay and dispersion this forces a monotone-definite increase in s.rootdisp (EPSILON), which avoids loops between peers operating at the same stratum.

The system variables are available to dependent application programs as nominal performance statistics. The system offset THETA is the clock offset relative to the available synchronization sources. The system jitter PSI is an estimate of the error in determining this value, elsewhere called the expected error. The root delay DELTA is the total round-trip delay relative to the primary server. The root dispersion EPSILON is the dispersion accumulated over the network from the primary server. Finally, the root synchronization distance is defined as:

LAMBDA = EPSILON + DELTA / 2,

which represents the maximum error due all causes and is designated the root synchronization distance.

An example of the clock update routine is provided in Appendix A.5.5.4.

11.3. Clock Discipline Algorithm

The NTPv4 clock discipline algorithm, shortened to discipline in the following, functions as a combination of two quite philosophically different feedback control systems. In a phase-locked loop (PLL) design, periodic phase updates at update intervals mu seconds are used directly to minimize the time error and indirectly the frequency error. In a frequency-locked loop (FLL) design, periodic frequency updates at intervals mu are used directly to minimize the frequency error and indirectly the time error. As shown in [ref7], a PLL usually works better when network jitter dominates, while an FLL works better when oscillator wander dominates. This section contains an outline of how the NTPv4 design works. An in-depth discussion of the design principles is provided in [ref7], which also includes a performance analysis.

The discipline is implemented as the feedback control system shown in Figure 26. The variable theta_r represents the combine algorithm offset (reference phase) and theta_c the VFO offset (control phase). Each update produces a signal V_d representing the instantaneous phase difference theta_r - theta_c. The clock filter for each server functions as a tapped delay line, with the output taken at the tap selected by the clock filter algorithm. The selection, cluster, and combine algorithms combine the data from multiple filters to produce the signal V_s. The loop filter, with impulse response F(t), produces the signal V_c, which controls the VFO frequency omega_c and thus the integral of the phase theta_c which closes the loop. The V_c signal is generated by the clock-adjust process in Section 12. The detailed equations that implement these functions are best presented in the routines of Appendices A.5.5.6 and A.5.6.1.

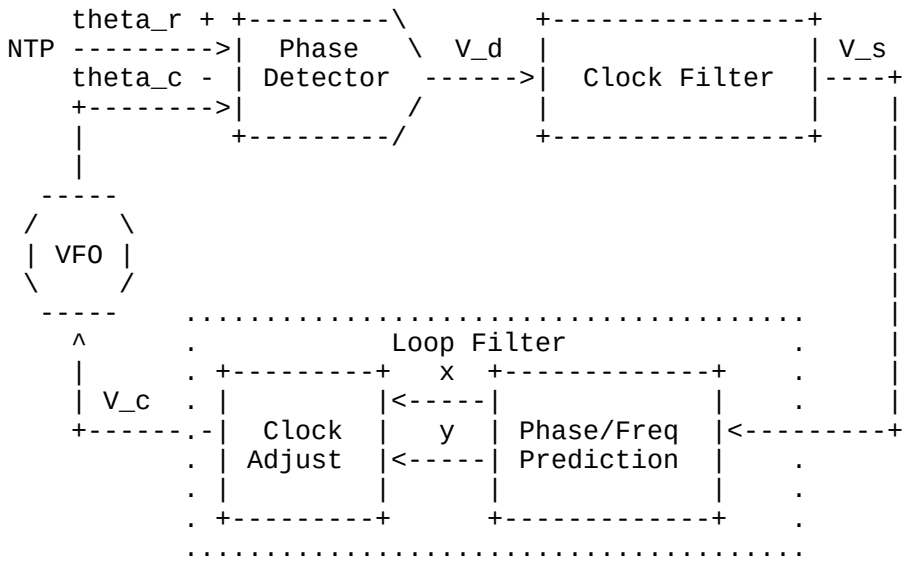


Figure 26: Clock Discipline Feedback Loop

Ordinarily, the pseudo-linear feedback loop described above operates to discipline the system clock. However, there are cases where a non-linear algorithm offers considerable improvement. One case is when the discipline starts without knowledge of the intrinsic clock frequency. The pseudo-linear loop takes several hours to develop an accurate measurement and during most of that time the poll interval cannot be increased. The non-linear loop described below does this in 15 minutes. Another case is when occasional bursts of large jitter are present due to congested network links. The state machine described below resists error bursts lasting less than 15 minutes.

Figure 27 contains a summary of the variables and parameters including the variable (lowercase) or parameter (uppercase) name, formula name, and short description. Unless noted otherwise, all variables have assumed prefix c. The variables t, tc, state, hyster, and count are integers; the remaining variables are floating doubles. The function of each will be explained in the algorithm descriptions below.

Name	Formula	Description
t	timer	seconds counter
offset	theta	combined offset
resid	theta_r	residual offset
freq	phi	clock frequency
jitter	psi	clock offset jitter
wander	omega	clock frequency wander
tc	tau	time constant (log2)
state	state	state
adj	adj	frequency adjustment
hyster	hyster	hysteresis counter
STEPT	125	step threshold (.125 s)
WATCH	900	stepout thresh(s)
PANICT	1000	panic threshold (1000 s)
LIMIT	30	hysteresis limit
PGATE	4	hysteresis gate
TC	16	time constant scale
AVG	8	averaging constant

Figure 27: Clock Discipline Variables and Parameters

The process terminates immediately if the offset is greater than the panic threshold PANICT (1000 s). The state transition function is described by the rstclock() function in Appendix A.5.5.7. Figure 28 shows the state transition function used by this routine. It has four columns showing, respectively, the state name, predicate and action if the offset theta is less than the step threshold, the predicate and actions otherwise, and finally some comments.

State	theta < STEP	theta > STEP	Comments
NSET	->FREQ adjust time	->FREQ step time	no frequency file
FSET	->SYNC adjust time	->SYNC step time	frequency file
SPIK	->SYNC adjust freq adjust time	if < 900 s ->SPIK else ->SYNC step freq step time	outlier detected
FREQ	if < 900 s ->FREQ else ->SYNC step freq adjust time	if < 900 s ->FREQ else ->SYNC step freq adjust time	initial frequency
SYNC	->SYNC adjust freq adjust time	if < 900 s ->SPIK else ->SYNC step freq step time	normal operation

Figure 28: State Transition Function

In the table entries, the next state is identified by the arrow -> with the actions listed below. Actions such as adjust time and adjust frequency are implemented by the PLL/FLL feedback loop in the local_clock() routine. A step clock action is implemented by setting the clock directly, but this is done only after the stepout threshold WATCH (900 s) when the offset is more than the step threshold STEPT (.125 s). This resists clock steps under conditions of extreme network congestion.

The jitter (psi) and wander (omega) statistics are computed using an exponential average with weight factor AVG. The time constant exponent (tau) is determined by comparing psi with the magnitude of the current offset theta. If the offset is greater than PGATE (4) times the clock jitter, the hysteresis counter hyster is reduced by two; otherwise, it is increased by one. If hyster increases to the upper limit LIMIT (30), tau is increased by one; if it decreases to the lower limit -LIMIT (-30), tau is decreased by one. Normally, tau hovers near MAXPOLL, but quickly decreases if a temperature spike causes a frequency surge.

12. Clock-Adjust Process

The actual clock-adjust process runs at one-second intervals to add the frequency correction and a fixed percentage of the residual offset theta_r. The theta_r is, in effect, the exponential decay of the theta value produced by the loop filter at each update. The TC parameter scales the time constant to match the poll interval for convenience. Note that the dispersion EPSILON increases by PHI at each second.

The clock-adjust process includes a timer interrupt facility driving the seconds counter c.t. It begins at zero when the service starts and increments once each second. At each interrupt, the clock_adjust() routine is called to incorporate the clock discipline time and frequency adjustments, then the associations are scanned to determine if the seconds counter equals or exceeds the p.next state variable defined in the next section. If so, the poll process is called to send a packet and compute the next p.next value.

An example of the clock-adjust process is shown by the clock_adjust() routine in Appendix A.5.6.1.

13. Poll Process

Each association supports a poll process that runs at regular intervals to construct and send packets in symmetric, client, and broadcast server associations. It runs continuously, whether or not servers are reachable in order to manage the clock filter and reach register.

13.1. Poll Process Variables

Figure 29 summarizes the common names, formula names, and a short description of the poll process variables (lowercase) and parameters (uppercase). Unless noted otherwise, all variables have assumed prefix p.

Name	Formula	Description
hpoll	hpoll	host poll exponent
last	last	last poll time
next	next	next poll time
reach	reach	reach register
unreach	unreach	unreach counter
UNREACH	24	unreach limit
BCOUNT	8	burst count
BURST	flag	burst enable
IBURST	flag	iburst enable

Figure 29: Poll Process Variables and Parameters

The poll process variables are allocated in the association data structure along with the peer process variables. The following is a detailed description of the variables. The parameters will be called out in the following text.

hpoll: signed integer representing the poll exponent, in log2 seconds

last: integer representing the seconds counter when the most recent packet was sent

next: integer representing the seconds counter when the next packet is to be sent

reach: 8-bit integer shift register shared by the peer and poll processes

unreach: integer representing the number of seconds the server has been unreachable

13.2. Poll Process Operations

As described previously, once each second the clock-adjust process is called. This routine calls the poll routine for each association in turn. If the time for the next poll message is greater than the seconds counter, the routine returns immediately. Symmetric (modes 1, 2), client (mode 3), and broadcast server (mode 5) associations routinely send packets. A broadcast client (mode 6) association runs the routine to update the reach and unreach variables, but does not send packets. The poll process calls the transmit process to send a packet. If in a burst (burst > 0), nothing further is done except call the poll update routine to set the next poll interval.

If not in a burst, the reach variable is shifted left by one bit, with zero replacing the rightmost bit. If the server has not been heard for the last three poll intervals, the clock filter routine is called to increase the dispersion. An example is shown in Appendix A.5.7.3.

If the BURST flag is lit and the server is reachable and a valid source of synchronization is available, the client sends a burst of BCOUNT (8) packets at each poll interval. The interval between packets in the burst is two seconds. This is useful to accurately measure jitter with long poll intervals. If the IBURST flag is lit and this is the first packet sent when the server has been unreachable, the client sends a burst. This is useful to quickly reduce the synchronization distance below the distance threshold and synchronize the clock.

If the P_MANY flag is lit in the p.flags word of the association, this is a manycast client association. Manycast client associations send client mode packets to designated multicast group addresses at MINPOLL intervals. The association starts out with a TTL of 1. If by the time of the next poll there are fewer than MINCLOCK servers have been mobilized, the ttl is increased by one. If the ttl reaches the limit TTLMAX, without finding MINCLOCK servers, the poll interval increases until reaching BEACON, when it starts over from the beginning.

The poll() routine includes a feature that backs off the poll interval if the server becomes unreachable. If reach is nonzero, the server is reachable and unreach is set to zero; otherwise, unreach is incremented by one for each poll to the maximum UNREACH. Thereafter for each poll hpoll is increased by one, which doubles the poll interval up to the maximum MAXPOLL determined by the poll_update() routine. When the server again becomes reachable, unreach is set to zero, hpoll is reset to the tc system variable, and operation resumes normally.

A packet is sent by the transmit process. Some header values are copied from the peer variables left by a previous packet and others from the system variables. Figure 30 shows which values are copied to each header field. In those implementations, using floating double data types for root delay and root dispersion, these must be converted to NTP short format. All other fields are either copied intact from peer and system variables or struck as a timestamp from the system clock.

Packet Variable	<--	Variable
x.leap	<--	s.leap
x.version	<--	s.version
x.mode	<--	s.mode
x.stratum	<--	s.stratum
x.poll	<--	s.poll
x.precision	<--	s.precision
x.rootdelay	<--	s.rootdelay
x.rootdisp	<--	s.rootdisp
x.refid	<--	s.refid
x.reftime	<--	s.reftime
x.org	<--	p.xmt
x.rec	<--	p.dst
x.xmt	<--	clock
x.keyid	<--	p.keyid
x.digest	<--	md5 digest

Figure 30: xmit_packet Packet Header

The poll update routine is called when a valid packet is received and immediately after a poll message has been sent. If in a burst, the poll interval is fixed at 2 s; otherwise, the host poll exponent hpoll is set to the minimum of ppoll from the last packet received and hpoll from the poll routine, but not less than MINPOLL or greater than MAXPOLL. Thus, the clock discipline can be oversampled but not undersampled. This is necessary to preserve subnet dynamic behavior and protect against protocol errors.

The poll exponent is converted to an interval, which, when added to the last poll time variable, determines the value of the next poll time variable. Finally, the last poll time variable is set to the current seconds counter.

14. Simple Network Time Protocol (SNTP)

Primary servers and clients complying with a subset of NTP, called the Simple Network Time Protocol (SNTPv4) [RFC4330], do not need to implement the mitigation algorithms described in Section 9 and following sections. SNTP is intended for primary servers equipped with a single reference clock, as well as for clients with a single upstream server and no dependent clients. The fully developed NTPv4 implementation is intended for secondary servers with multiple upstream servers and multiple downstream servers or clients. Other than these considerations, NTP and SNTP servers and clients are completely interoperable and can be intermixed in NTP subnets.

An SNTP primary server implementing the on-wire protocol described in Section 8 has no upstream servers except a single reference clock. In principle, it is indistinguishable from an NTP primary server that has the mitigation algorithms and therefore capable of mitigating between multiple reference clocks.

Upon receiving a client request, an SNTP primary server constructs and sends the reply packet as described in Figure 31. Note that the dispersion field in the packet header must be updated as described in Section 5.

Packet Variable	<--	Variable
x.leap	<--	s.leap
x.version	<--	r.version
x.mode	<--	4
x.stratum	<--	s.stratum
x.poll	<--	r.poll
x.precision	<--	s.precision
x.rootdelay	<--	s.rootdelay
x.rootdisp	<--	s.rootdisp
x.refid	<--	s.refid
x.reftime	<--	s.reftime
x.org	<--	r.xmt
x.rec	<--	r.dst
x.xmt	<--	clock
x.keyid	<--	r.keyid
x.digest	<--	md5 digest

Figure 31: fast_xmit Packet Header

An SNTP client implementing the on-wire protocol has a single server and no dependent clients. It can operate with any subset of the NTP on-wire protocol, the simplest approach using only the transmit timestamp of the server packet and ignoring all other fields. However, the additional complexity to implement the full on-wire protocol is minimal so that a full implementation is encouraged.

15. Security Considerations

NTP security requirements are even more stringent than most other distributed services. First, the operation of the authentication mechanism and the time synchronization mechanism are inextricably intertwined. Reliable time synchronization requires cryptographic keys that are valid only over a designated time interval; but, time intervals can be enforced only when participating servers and clients

are reliably synchronized to UTC. In addition, the NTP subnet is hierarchical by nature, so time and trust flow from the primary servers at the root through secondary servers to the clients at the leaves.

An NTP client can claim to have authentic time to dependent applications only if all servers on the path to the primary servers are authenticated. In NTP each server authenticates the next lower stratum servers and authenticates by induction the lowest stratum (primary) servers. It is important to note that authentication in the context of NTP does not necessarily imply the time is correct. An NTP client mobilizes a number of concurrent associations with different servers and uses a crafted agreement algorithm to pluck truechimers from the population possibly including falsetickers.

The NTP specification assumes that the goal of the intruder is to inject false time values, disrupt the protocol, or clog the network, servers, or clients with spurious packets that exhaust resources and deny service to legitimate applications. There are a number of defense mechanisms already built in the NTP architecture, protocol, and algorithms. The on-wire timestamp exchange scheme is inherently resistant to spoofing, packet-loss, and replay attacks. The engineered clock filter, selection and clustering algorithms are designed to defend against evil cliques of Byzantine traitors. While not necessarily designed to defeat determined intruders, these algorithms and accompanying sanity checks have functioned well over the years to deflect improperly operating but presumably friendly scenarios. However, these mechanisms do not securely identify and authenticate servers to clients. Without specific further protection, an intruder can inject any or all of the following attacks:

1. An intruder can intercept and archive packets forever, as well as all the public values ever generated and transmitted over the net.

2. An intruder can generate packets faster than the server, network or client can process them, especially if they require expensive cryptographic computations.

3. In a wiretap attack, the intruder can intercept, modify, and replay a packet. However, it cannot permanently prevent onward transmission of the original packet; that is, it cannot break the wire, only tell lies and congest it. Generally, the modified packet cannot arrive at the victim before the original packet, nor does it have the server private keys or identity parameters.

4. In a middleman or masquerade attack, the intruder is positioned between the server and client, so it can intercept, modify and replay a packet and prevent onward transmission of the original packet. However, the middleman does not have the server private keys.

The NTP security model assumes the following possible limitations:

1. The running times for public key algorithms are relatively long and highly variable. In general, the performance of the time synchronization function is badly degraded if these algorithms must be used for every NTP packet.

2. In some modes of operation, it is not feasible for a server to retain state variables for every client. It is however feasible to regenerated them for a client upon arrival of a packet from that client.

3. The lifetime of cryptographic values must be enforced, which requires a reliable system clock. However, the sources that synchronize the system clock must be trusted. This circular interdependence of the timekeeping and authentication functions requires special handling.

4. Client security functions must involve only public values transmitted over the net. Private values must never be disclosed beyond the machine on which they were created, except in the case of a special trusted agent (TA) assigned for this purpose.

Unlike the Secure Shell (SSH) security model, where the client must be securely authenticated to the server, in NTP the server must be securely authenticated to the client. In SSH, each different interface address can be bound to a different name, as returned by a reverse-DNS query. In this design, separate public/private key pairs may be required for each interface address with a distinct name. A perceived advantage of this design is that the security compartment can be different for each interface. This allows a firewall, for instance, to require some interfaces to authenticate the client and others not.

In the case of NTP as specified herein, NTP broadcast clients are vulnerable to disruption by misbehaving or hostile SNTP or NTP broadcast servers elsewhere in the Internet. Such disruption can be minimized by several approaches. Filtering can be employed to limit the access of NTP clients to known or trusted NTP broadcast servers. Such filtering will prevent malicious traffic from reaching the NTP clients. Cryptographic authentication at the client will only allow

timing information from properly signed NTP messages to be utilized in synchronizing its clock. Higher levels of authentication may be gained by the use of the Autokey mechanism [RFC5906].

Section 8 describes a potential security concern with the replay of client requests. Following the recommendations in that section provides protection against such attacks.

It should be noted that this specification is describing an existing implementation. While the security shortfalls of the MD5 algorithm are well-known, its use in the NTP specification is consistent with widescale deployment in the Internet community.

16. IANA Considerations

UDP/TCP Port 123 was previously assigned by IANA for this protocol. The IANA has assigned the IPv4 multicast group address 224.0.1.1 and the IPv6 multicast address ending :101 for NTP. This document introduces NTP extension fields allowing for the development of future extensions to the protocol, where a particular extension is to be identified by the Field Type sub-field within the extension field. IANA has established and will maintain a registry for Extension Field Types associated with this protocol, populating this registry with no initial entries. As future needs arise, new Extension Field Types may be defined. Following the policies outlined in [RFC5226], new values are to be defined by IETF Review.

The IANA has created a new registry for NTP Reference Identifier codes. This includes the current codes defined in Section 7.3, and may be extended on a First-Come-First-Serve (FCFS) basis. The format of the registry is:

ID	Clock Source
GOES	Geosynchronous Orbit Environment Satellite
GPS	Global Position System
...	...

Figure 32: Reference Identifier Codes

The IANA has created a new registry for NTP Kiss-o'-Death codes. This includes the current codes defined in Section 7.4, and may be extended on a FCFS basis. The format of the registry is:

Code	Meaning
ACST	The association belongs to a unicast server.
AUTH	Server authentication failed.
...	...

Figure 33: Kiss Codes

For both Reference Identifiers and Kiss-o'-Death codes, IANA is requested to never assign a code beginning with the character "X", as this is reserved for experimentation and development.

17. Acknowledgements

The editors would like to thank Karen O'Donoghue, Brian Haberman, Greg Dowd, Mark Elliot, Harlan Stenn, Yaakov Stein, Stewart Bryant, and Danny Mayer for technical reviews and specific text contributions to this document.

18. References

18.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

18.2. Informative References

[CGPM] Bureau International des Poids et Mesures, "Comptes Rendus de la 15e CGPM", 1976.

[ITU-R_TF.460] International Telecommunications Union, "ITU-R TF.460 Standard-frequency and time-signal emissions", February 2002.

[RFC1305] Mills, D., "Network Time Protocol (Version 3) Specification, Implementation and Analysis", RFC 1305, March 1992.

[RFC1345] Simonsen, K., "Character Mnemonics and Character Sets", RFC 1345, June 1992.

[RFC4330] Mills, D., "Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI", RFC 4330, January 2006.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5906] Haberman, B., Ed. and D. Mills, "Network Time Protocol Version 4: Autokey Specification", RFC 5906, June 2010.

[ref6] Marzullo and S. Owicki, "Maintaining the time in a distributed system", ACM Operating Systems Review 19, July 1985.

[ref7] Mills, D.L., "Computer Network Time Synchronization - the Network Time Protocol", CRC Press, 304 pp, 2006.

[ref9] Mills, D.L., Electrical and Computer Engineering Technical Report 06-6-1, NDSS, June 2006, "Network Time Protocol Version 4 Reference and Implementation Guide", 2006.

Appendix A. Code Skeleton

This appendix is intended to describe the protocol and algorithms of an implementation in a general way using what is called a code skeleton program. This consists of a set of definitions, structures, and code fragments that illustrate the protocol operations without the complexities of an actual implementation of the protocol. This program is not an executable and is not designed to run in the ordinary sense.

Most of the features of the reference implementation are included here, with the following exceptions: there are no provisions for reference clocks or public key (Autokey) cryptography. There is no huff-n'-puff filter, anti-clockhop hysteresis, or monitoring provisions. Many of the values that can be tinkered in the reference implementation are assumed constants here. There are only minimal provisions for the kiss-o'-death packet and no responding code.

The program is not intended to be fast or compact, just to demonstrate the algorithms with sufficient fidelity to understand how they work. The code skeleton consists of eight segments, a header segment included by each of the other segments, plus a code segment for the main program, kernel I/O and system clock interfaces, and peer, system, clock_adjust, and poll processes. These are presented in order below along with definitions and variables specific to each process.

A.1. Global Definitions

A.1.1. Definitions, Constants, Parameters

```
#include <math.h>               /* avoids complaints about sqrt() */
#include <sys/time.h>           /* for gettimeofday() and friends */
#include <stdlib.h>             /* for malloc() and friends */
#include <string.h>             /* for memset() */

/*
 * Data types
 *
 * This program assumes the int data type is 32 bits and the long data
 * type is 64 bits.  The native data type used in most calculations is
 * floating double.  The data types used in some packet header fields
 * require conversion to and from this representation.  Some header
 * fields involve partitioning an octet, here represented by individual
 * octets.
 *
 * The 64-bit NTP timestamp format used in timestamp calculations is
 * unsigned seconds and fraction with the decimal point to the left of
```

```
 * bit 32.  The only operation permitted with these values is
 * subtraction, yielding a signed 31-bit difference.  The 32-bit NTP
 * short format used in delay and dispersion calculations is seconds and
 * fraction with the decimal point to the left of bit 16.  The only
 * operations permitted with these values are addition and
 * multiplication by a constant.
 *
 * The IPv4 address is 32 bits, while the IPv6 address is 128 bits.  The
 * message digest field is 128 bits as constructed by the MD5 algorithm.
 * The precision and poll interval fields are signed log2 seconds.
 */
typedef unsigned long long tstamp;   /* NTP timestamp format */
typedef unsigned int tdist;     /* NTP short format */
typedef unsigned long ipaddr;   /* IPv4 or IPv6 address */
typedef unsigned long digest;   /* md5 digest */
typedef signed char s_char;     /* precision and poll interval (log2) */

/*
 * Timestamp conversion macroni
 */
#define FRIC        65536.                   /* 2^16 as a double */
#define D2FP(r)     ((tdist)((r) * FRIC))    /* NTP short */
#define FP2D(r)     ((double)(r) / FRIC)

#define FRAC       4294967296.              /* 2^32 as a double */
#define D2LFP(a)   ((tstamp)((a) * FRAC))  /* NTP timestamp */
#define LFP2D(a)   ((double)(a) / FRAC)
#define U2LFP(a)   (((unsigned long long) \
                     ((a).tv_sec + JAN_1970) << 32) + \
                     (unsigned long long) \
                     ((a).tv_usec / 1e6 * FRAC))

/*
 * Arithmetic conversions
 */
#define LOG2D(a)        ((a) < 0 ? 1. / (1L << -(a)) : \
                            1L << (a))          /* poll, etc. */
#define SQUARE(x)       (x * x)
#define SQRT(x)         (sqrt(x))

/*
 * Global constants.  Some of these might be converted to variables
 * that can be tinkered by configuration or computed on-the-fly.  For
 * instance, the reference implementation computes PRECISION on-the-fly
 * and provides performance tuning for the defines marked with % below.
 */
#define VERSION         4       /* version number */
#define MINDISP         .01     /* % minimum dispersion (s) */
```

```
#define MAXDISP         16      /* maximum dispersion (s) */
#define MAXDIST         1       /* % distance threshold (s) */
#define NOSYNC          0x3     /* leap unsync */
#define MAXSTRAT        16      /* maximum stratum (infinity metric) */
#define MINPOLL         6       /* % minimum poll interval (64 s)*/
#define MAXPOLL         17      /* % maximum poll interval (36.4 h) */
#define MINCLOCK        3       /* minimum manycast survivors */
#define MAXCLOCK        10      /* maximum manycast candidates */
#define TTLMAX          8       /* max ttl manycast */
#define BEACON          15      /* max interval between beacons */

#define PHI             15e-6   /* % frequency tolerance (15 ppm) */
#define NSTAGE          8       /* clock register stages */
#define NMAX            50      /* maximum number of peers */
#define NSANE           1       /* % minimum intersection survivors */
#define NMIN            3       /* % minimum cluster survivors */

/*
 * Global return values
 */
#define TRUE            1       /* boolean true */
#define FALSE           0       /* boolean false */

/*
 * Local clock process return codes
 */
#define IGNORE          0       /* ignore */
#define SLEW            1       /* slew adjustment */
#define STEP            2       /* step adjustment */
#define PANIC           3       /* panic - no adjustment */

/*
 * System flags
 */
#define S_FLAGS         0       /* any system flags */
#define S_BCSTENAB      0x1     /* enable broadcast client */

/*
 * Peer flags
 */
#define P_FLAGS         0       /* any peer flags */
#define P_EPHEM         0x01    /* association is ephemeral */
#define P_BURST         0x02    /* burst enable */
#define P_IBURST        0x04    /* intial burst enable */
#define P_NOTRUST       0x08    /* authenticated access */
#define P_NOPEER        0x10    /* authenticated mobilization */
#define P_MANY          0x20    /* manycast client */
```

```
/*
 * Authentication codes
 */
#define A_NONE          0       /* no authentication */
#define A_OK            1       /* authentication OK */
#define A_ERROR         2       /* authentication error */
#define A_CRYPTO        3       /* crypto-NAK */

/*
 * Association state codes
 */
#define X_INIT          0       /* initialization */
#define X_STALE         1       /* timeout */
#define X_STEP          2       /* time step */
#define X_ERROR         3       /* authentication error */
#define X_CRYPTO        4       /* crypto-NAK received */
#define X_NKEY          5       /* untrusted key */

/*
 * Protocol mode definitions
 */
#define M_RSVD          0       /* reserved */
#define M_SACT          1       /* symmetric active */
#define M_PASV          2       /* symmetric passive */
#define M_CLNT          3       /* client */
#define M_SERV          4       /* server */
#define M_BCST          5       /* broadcast server */
#define M_BCLN          6       /* broadcast client */

/*
 * Clock state definitions
 */
#define NSET            0       /* clock never set */
#define FSET            1       /* frequency set from file */
#define SPIK            2       /* spike detected */
#define FREQ            3       /* frequency mode */
#define SYNC            4       /* clock synchronized */

#define min(a, b)   ((a) < (b) ? (a) : (b))
#define max(a, b)   ((a) < (b) ? (b) : (a))
```

A.1.2. Packet Data Structures

```
/*
 * The receive and transmit packets may contain an optional message
 * authentication code (MAC) consisting of a key identifier (keyid) and
 * message digest (mac in the receive structure and dgst in the transmit
 * structure).  NTPv4 supports optional extension fields that
 * are inserted after the header and before the MAC, but these are
 * not described here.
 *
 * Receive packet
 *
 * Note the dst timestamp is not part of the packet itself.  It is
 * captured upon arrival and returned in the receive buffer along with
 * the buffer length and data.  Note that some of the char fields are
 * packed in the actual header, but the details are omitted here.
 */
struct r {
        ipaddr  srcaddr;        /* source (remote) address */
        ipaddr  dstaddr;        /* destination (local) address */
        char    version;        /* version number */
        char    leap;           /* leap indicator */
        char    mode;           /* mode */
        char    stratum;        /* stratum */
        char    poll;           /* poll interval */
        s_char  precision;      /* precision */
        tdist   rootdelay;      /* root delay */
        tdist   rootdisp;       /* root dispersion */
        char    refid;          /* reference ID */
        tstamp  reftime;        /* reference time */
        tstamp  org;            /* origin timestamp */
        tstamp  rec;            /* receive timestamp */
        tstamp  xmt;            /* transmit timestamp */
        int     keyid;          /* key ID */
        digest  mac;            /* message digest */
        tstamp  dst;            /* destination timestamp */
} r;

/*
 * Transmit packet
 */
struct x {
        ipaddr  dstaddr;        /* source (local) address */
        ipaddr  srcaddr;        /* destination (remote) address */
        char    version;        /* version number */
        char    leap;           /* leap indicator */
        char    mode;           /* mode */
        char    stratum;        /* stratum */
```

```
        char    poll;           /* poll interval */
        s_char  precision;      /* precision */
        tdist   rootdelay;      /* root delay */
        tdist   rootdisp;       /* root dispersion */
        char    refid;          /* reference ID */
        tstamp  reftime;        /* reference time */
        tstamp  org;            /* origin timestamp */
        tstamp  rec;            /* receive timestamp */
        tstamp  xmt;            /* transmit timestamp */
        int     keyid;          /* key ID */
        digest  dgst;           /* message digest */
} x;
```

A.1.3. Association Data Structures

```
   /*
    * Filter stage structure.  Note the t member in this and other
    * structures refers to process time, not real time.  Process time
    * increments by one second for every elapsed second of real time.
    */
   struct f {
           tstamp  t;              /* update time */
           double  offset;         /* clock ofset */
           double  delay;          /* roundtrip delay */
           double  disp;           /* dispersion */
   } f;

   /*
    * Association structure.  This is shared between the peer process
    * and poll process.
    */
   struct p {

           /*
            * Variables set by configuration
            */
           ipaddr  srcaddr;        /* source (remote) address */
           ipaddr  dstaddr;        /* destination (local) address */
           char    version;        /* version number */
           char    hmode;          /* host mode */
           int     keyid;          /* key identifier */
           int     flags;          /* option flags */

           /*
            * Variables set by received packet
            */
           char    leap;           /* leap indicator */
           char    pmode;          /* peer mode */
```

```
        char    stratum;        /* stratum */
        char    ppoll;          /* peer poll interval */
        double  rootdelay;      /* root delay */
        double  rootdisp;       /* root dispersion */
        char    refid;          /* reference ID */
        tstamp  reftime;        /* reference time */
#define begin_clear org         /* beginning of clear area */
        tstamp  org;            /* originate timestamp */
        tstamp  rec;            /* receive timestamp */
        tstamp  xmt;            /* transmit timestamp */

        /*
         * Computed data
         */
        double  t;              /* update time */
        struct f f[NSTAGE];     /* clock filter */
        double  offset;         /* peer offset */
        double  delay;          /* peer delay */
        double  disp;           /* peer dispersion */
        double  jitter;         /* RMS jitter */

        /*
         * Poll process variables
         */
        char    hpoll;          /* host poll interval */
        int     burst;          /* burst counter */
        int     reach;          /* reach register */
        int     ttl;            /* ttl (manycast) */
#define end_clear unreach       /* end of clear area */
        int     unreach;        /* unreach counter */
        int     outdate;        /* last poll time */
        int     nextdate;       /* next poll time */
} p;
```

A.1.4. System Data Structures

```
   /*
    * Chime list.  This is used by the intersection algorithm.
    */
   struct m {                       /* m is for Marzullo */
           struct p *p;             /* peer structure pointer */
           int     type;            /* high +1, mid 0, low -1 */
           double  edge;            /* correctness interval edge */
   } m;

   /*
    * Survivor list.  This is used by the clustering algorithm.
    */
   struct v {
           struct p *p;             /* peer structure pointer */
           double  metric;          /* sort metric */
   } v;

   /*
    * System structure
    */
   struct s {
           tstamp  t;               /* update time */
           char    leap;            /* leap indicator */
           char    stratum;         /* stratum */
           char    poll;            /* poll interval */
           char    precision;       /* precision */
           double  rootdelay;       /* root delay */
           double  rootdisp;        /* root dispersion */
           char    refid;           /* reference ID */
           tstamp  reftime;         /* reference time */
           struct m m[NMAX];        /* chime list */
           struct v v[NMAX];        /* survivor list */
           struct p *p;             /* association ID */
           double  offset;          /* combined offset */
           double  jitter;          /* combined jitter */
           int     flags;           /* option flags */
           int     n;               /* number of survivors */
   } s;
```

A.1.5. Local Clock Data Structures

```
   /*
    * Local clock structure
    */
   struct c {
           tstamp  t;              /* update time */
           int     state;          /* current state */
           double  offset;         /* current offset */
           double  last;           /* previous offset */
           int     count;          /* jiggle counter */
           double  freq;           /* frequency */
           double  jitter;         /* RMS jitter */
           double  wander;         /* RMS wander */
   } c;
```

A.1.6. Function Prototypes

```
  /*
   * Peer process
   */
  void    receive(struct r *);    /* receive packet */
  void    packet(struct p *, struct r *); /* process packet */
  void    clock_filter(struct p *, double, double, double); /* filter */
  double  root_dist(struct p *);  /* calculate root distance */
  int     fit(struct p *);        /* determine fitness of server */
  void    clear(struct p *, int); /* clear association */
  int     access(struct r *);     /* determine access restrictions */

  /*
   * System process
   */
  int     main();                 /* main program */
  void    clock_select();         /* find the best clocks */
  void    clock_update(struct p *); /* update the system clock */
  void    clock_combine();        /* combine the offsets */

  /*
   * Local clock process
   */
  int     local_clock(struct p *, double); /* clock discipline */
  void    rstclock(int, double, double); /* clock state transition */

  /*
   * Clock adjust process
   */
  void    clock_adjust();         /* one-second timer process */
```

```
  /*
   * Poll process
   */
  void    poll(struct p *);               /* poll process */
  void    poll_update(struct p *, int); /* update the poll interval */
  void    peer_xmit(struct p *);  /* transmit a packet */
  void    fast_xmit(struct r *, int, int); /* transmit a reply packet */

  /*
   * Utility routines
   */
  digest  md5(int);               /* generate a message digest */
  struct p *mobilize(ipaddr, ipaddr, int, int, int, int); /* mobilize */
  struct p *find_assoc(struct r *); /* search the association table */

  /*
   * Kernel interface
   */
  struct r *recv_packet();        /* wait for packet */
  void    xmit_packet(struct x *); /* send packet */
  void    step_time(double);      /* step time */
  void    adjust_time(double);    /* adjust (slew) time */
  tstamp  get_time();             /* read time */
```

A.2. Main Program and Utility Routines

```
/*
 * Definitions
 */
#define PRECISION       -18     /* precision (log2 s)  */
#define IPADDR          0       /* any IP address */
#define MODE            0       /* any NTP mode */
#define KEYID           0       /* any key identifier */

/*
 * main() - main program
 */
int
main()
{
        struct p *p;            /* peer structure pointer */
        struct r *r;            /* receive packet pointer */
```

```
        /*
         * Read command line options and initialize system variables.
         * The reference implementation measures the precision specific
         * to each machine by measuring the clock increments to read the
         * system clock.
         */
        memset(&s, sizeof(s), 0);
        s.leap = NOSYNC;
        s.stratum = MAXSTRAT;
        s.poll = MINPOLL;
        s.precision = PRECISION;
        s.p = NULL;

        /*
         * Initialize local clock variables
         */
        memset(&c, sizeof(c), 0);
        if (/* frequency file */ 0) {
                c.freq = /* freq */ 0;
                rstclock(FSET, 0, 0);
        } else {
                rstclock(NSET, 0, 0);
        }
        c.jitter = LOG2D(s.precision);

        /*
         * Read the configuration file and mobilize persistent
         * associations with specified addresses, version, mode, key ID,
         * and flags.
         */
        while (/* mobilize configurated associations */ 0) {
                p = mobilize(IPADDR, IPADDR, VERSION, MODE, KEYID,
                    P_FLAGS);
        }

        /*
         * Start the system timer, which ticks once per second.  Then,
         * read packets as they arrive, strike receive timestamp, and
         * call the receive() routine.
         */
        while (0) {
                r = recv_packet();
                r->dst = get_time();
                receive(r);
        }

        return(0);
}
```

```
/*
 * mobilize() - mobilize and initialize an association
 */
struct p
*mobilize(
        ipaddr  srcaddr,        /* IP source address */
        ipaddr  dstaddr,        /* IP destination address */
        int     version,        /* version */
        int     mode,           /* host mode */
        int     keyid,          /* key identifier */
        int     flags           /* peer flags */
        )
{
        struct p *p;            /* peer process pointer */

        /*
         * Allocate and initialize association memory
         */
        p = malloc(sizeof(struct p));
        p->srcaddr = srcaddr;
        p->dstaddr = dstaddr;
        p->version = version;
        p->hmode = mode;
        p->keyid = keyid;
        p->hpoll = MINPOLL;
        clear(p, X_INIT);
        p->flags = flags;
        return (p);
}

/*
 * find_assoc() - find a matching association
 */
struct p                        /* peer structure pointer or NULL */
*find_assoc(
        struct r *r             /* receive packet pointer */
        )
{
        struct p *p;            /* dummy peer structure pointer */

        /*
         * Search association table for matching source
         * address, source port and mode.
         */
        while (/* all associations */ 0) {
                if (r->srcaddr == p->srcaddr && r->mode == p->hmode)
                        return(p);
        }
```

```
        return (NULL);
}

/*
 * md5() - compute message digest
 */
digest
md5(
       int     keyid           /* key identifier */
       )
{
       /*
        * Compute a keyed cryptographic message digest.  The key
        * identifier is associated with a key in the local key cache.
        * The key is prepended to the packet header and extension fields
        * and the result hashed by the MD5 algorithm as described in
        * RFC 1321.  Return a MAC consisting of the 32-bit key ID
        * concatenated with the 128-bit digest.
        */
       return (/* MD5 digest */ 0);
}
```

A.3. Kernel Input/Output Interface

```
   /*
    * Kernel interface to transmit and receive packets.  Details are
    * deliberately vague and depend on the operating system.
    *
    * recv_packet - receive packet from network
    */
   struct r                        /* receive packet pointer*/
   *recv_packet() {
           return (/* receive packet r */ 0);
   }


   /*
    * xmit_packet - transmit packet to network
    */
   void
   xmit_packet(
           struct x *x             /* transmit packet pointer */
           )
   {
           /* send packet x */
   }
```

A.4. Kernel System Clock Interface

```
/*
 * System clock utility functions
 *
 * There are three time formats: native (Unix), NTP, and floating
 * double.  The get_time() routine returns the time in NTP long format.
 * The Unix routines expect arguments as a structure of two signed
 * 32-bit words in seconds and microseconds (timeval) or nanoseconds
 * (timespec).  The step_time() and adjust_time() routines expect signed
 * arguments in floating double.  The simplified code shown here is for
 * illustration only and has not been verified.
 */
#define JAN_1970        2208988800UL /* 1970 - 1900 in seconds */

/*
 * get_time - read system time and convert to NTP format
 */
tstamp
get_time()
{
        struct timeval unix_time;

        /*
         * There are only two calls on this routine in the program.  One
         * when a packet arrives from the network and the other when a
         * packet is placed on the send queue.  Call the kernel time of
         * day routine (such as gettimeofday()) and convert to NTP
         * format.
         */
        gettimeofday(&unix_time, NULL);
        return (U2LFP(unix_time));
}
```

```
/*
 * step_time() - step system time to given offset value
 */
void
step_time(
        double  offset          /* clock offset */
        )
{
        struct timeval unix_time;
        tstamp  ntp_time;

        /*
         * Convert from double to native format (signed) and add to the
         * current time.  Note the addition is done in native format to
         * avoid overflow or loss of precision.
         */
        gettimeofday(&unix_time, NULL);
        ntp_time = D2LFP(offset) + U2LFP(unix_time);
        unix_time.tv_sec = ntp_time >> 32;
        unix_time.tv_usec = (long)(((ntp_time - unix_time.tv_sec) <<
            32) / FRAC * 1e6);
        settimeofday(&unix_time, NULL);
}


/*
 * adjust_time() - slew system clock to given offset value
 */
void
adjust_time(
        double  offset          /* clock offset */
        )
{
        struct timeval unix_time;
        tstamp  ntp_time;

        /*
         * Convert from double to native format (signed) and add to the
         * current time.
         */
        ntp_time = D2LFP(offset);
        unix_time.tv_sec = ntp_time >> 32;
        unix_time.tv_usec = (long)(((ntp_time - unix_time.tv_sec) <<
            32) / FRAC * 1e6);
        adjtime(&unix_time, NULL);
}
```

A.5. Peer Process

```
/*
 * A crypto-NAK packet includes the NTP header followed by a MAC
 * consisting only of the key identifier with value zero.  It tells
 * the receiver that a prior request could not be properly
 * authenticated, but the NTP header fields are correct.
 *
 * A kiss-o'-death packet is an NTP header with leap 0x3 (NOSYNC) and
 * stratum 16 (MAXSTRAT).  It tells the receiver that something
 * drastic has happened, as revealed by the kiss code in the refid
 * field.  The NTP header fields may or may not be correct.
 */
/*
 * Peer process parameters and constants
 */
#define SGATE           3       /* spike gate (clock filter */
#define BDELAY          .004    /* broadcast delay (s) */

/*
 * Dispatch codes
 */
#define ERR             -1      /* error */
#define DSCRD           0       /* discard packet */
#define PROC            1       /* process packet */
#define BCST            2       /* broadcast packet */
#define FXMIT           3       /* client packet */
#define MANY            4       /* manycast packet */
#define NEWPS           5       /* new symmetric passive client */
#define NEWBC           6       /* new broadcast client */

/*
 * Dispatch matrix
 *              active  passv  client server bcast */
int table[7][5] = {
/* nopeer  */   { NEWPS, DSCRD, FXMIT, MANY, NEWBC },
/* active  */   { PROC,  PROC,  DSCRD, DSCRD, DSCRD },
/* passv   */   { PROC,  ERR,   DSCRD, DSCRD, DSCRD },
/* client  */   { DSCRD, DSCRD, DSCRD, PROC,  DSCRD },
/* server  */   { DSCRD, DSCRD, DSCRD, DSCRD, DSCRD },
/* bcast   */   { DSCRD, DSCRD, DSCRD, DSCRD, DSCRD },
/* bclient */   { DSCRD, DSCRD, DSCRD, DSCRD, PROC}
};
```

```
   /*
    * Miscellaneous macroni
    *
    * This macro defines the authentication state.  If x is 0,
    * authentication is optional; otherwise, it is required.
    */
   #define AUTH(x, y)      ((x) ? (y) == A_OK : (y) == A_OK || \
                               (y) == A_NONE)

   /*
    * These are used by the clear() routine
    */
   #define BEGIN_CLEAR(p)  ((char *)&((p)->begin_clear))
   #define END_CLEAR(p)    ((char *)&((p)->end_clear))
   #define LEN_CLEAR       (END_CLEAR((struct p *)0) - \
                               BEGIN_CLEAR((struct p *)0))
```

A.5.1. receive()

```
/*
 * receive() - receive packet and decode modes
 */
void
receive(
        struct r *r             /* receive packet pointer */
        )
{
        struct p *p;            /* peer structure pointer */
        int     auth;           /* authentication code */
        int     has_mac;        /* size of MAC */
        int     synch;          /* synchronized switch */

        /*
         * Check access control lists.  The intent here is to implement
         * a whitelist of those IP addresses specifically accepted
         * and/or a blacklist of those IP addresses specifically
         * rejected.  There could be different lists for authenticated
         * clients and unauthenticated clients.
         */
        if (!access(r))
                return;                 /* access denied */

        /*
         * The version must not be in the future.  Format checks include
         * packet length, MAC length and extension field lengths, if
         * present.
         */
```

```
        if (r->version > VERSION /* or format error */)
                return;                 /* format error */

        /*
         * Authentication is conditioned by two switches that can be
         * specified on a per-client basis.
         *
         * P_NOPEER     do not mobilize an association unless
         *              authenticated.
         * P_NOTRUST    do not allow access unless authenticated
         *              (implies P_NOPEER).
         *
         * There are four outcomes:
         *
         * A_NONE       the packet has no MAC.
         * A_OK         the packet has a MAC and authentication
         *               succeeds.
         * A_ERROR      the packet has a MAC and authentication fails.
         * A_CRYPTO     crypto-NAK.  The MAC has four octets only.
         *
         * Note: The AUTH (x, y) macro is used to filter outcomes.  If x
         * is zero, acceptable outcomes of y are NONE and OK.  If x is
         * one, the only acceptable outcome of y is OK.
         */

        has_mac = /* length of MAC field */ 0;
        if (has_mac == 0) {
                auth = A_NONE;          /* not required */
        } else if (has_mac == 4) {
                auth = A_CRYPTO;       /* crypto-NAK */
        } else {
                if (r->mac != md5(r->keyid))
                        auth = A_ERROR; /* auth error */
                else
                        auth = A_OK;    /* auth OK */
        }

        /*
         * Find association and dispatch code.  If there is no
         * association to match, the value of p->hmode is assumed NULL.
         */
        p = find_assoc(r);
        switch(table[(unsigned int)(p->hmode)][(unsigned int)(r->mode)])
        {
```

```
        /*
         * Client packet and no association.  Send server reply without
         * saving state.
         */
        case FXMIT:

                /*
                 * If unicast destination address, send server packet.
                 * If authentication fails, send a crypto-NAK packet.
                 */

                /* not multicast dstaddr */
                if (0) {
                        if (AUTH(p->flags & P_NOTRUST, auth))
                                fast_xmit(r, M_SERV, auth);
                        else if (auth == A_ERROR)
                                fast_xmit(r, M_SERV, A_CRYPTO);
                        return;         /* M_SERV packet sent */
                }

                /*
                 * This must be manycast.  Do not respond if we are not
                 * synchronized or if our stratum is above the
                 * manycaster.
                 */
                if (s.leap == NOSYNC || s.stratum > r->stratum)
                        return;

                /*
                 * Respond only if authentication is OK.  Note that the
                 * unicast address is used, not the multicast.
                 */
                if (AUTH(p->flags & P_NOTRUST, auth))
                        fast_xmit(r, M_SERV, auth);
                return;

        /*
         * New manycast client ephemeral association.  It is mobilized
         * in the same version as in the packet.  If authentication
         * fails, ignore the packet.  Verify the server packet by
         * comparing the r->org timestamp in the packet with the p->xmt
         * timestamp in the multicast client association.  If they
         * match, the server packet is authentic.  Details omitted.
         */
```

```
        case MANY:
                if (!AUTH(p->flags & (P_NOTRUST | P_NOPEER), auth))
                        return;         /* authentication error */

                p = mobilize(r->srcaddr, r->dstaddr, r->version, M_CLNT,
                    r->keyid, P_EPHEM);
                break;

       /*
        * New symmetric passive association.  It is mobilized in the
        * same version as in the packet.  If authentication fails,
        * send a crypto-NAK packet.  If restrict no-moblize, send a
        * symmetric active packet instead.
        */
        case NEWPS:
                if (!AUTH(p->flags & P_NOTRUST, auth)) {
                        if (auth == A_ERROR)
                                fast_xmit(r, M_SACT, A_CRYPTO);
                        return;         /* crypto-NAK packet sent */
                }
                if (!AUTH(p->flags & P_NOPEER, auth)) {
                        fast_xmit(r, M_SACT, auth);
                        return;         /* M_SACT packet sent */
                }
                p = mobilize(r->srcaddr, r->dstaddr, r->version, M_PASV,
                    r->keyid, P_EPHEM);
                break;

        /*
         * New broadcast client association.  It is mobilized in the
         * same version as in the packet.  If authentication fails,
         * ignore the packet.  Note this code does not support the
         * initial volley feature in the reference implementation.
         */
        case NEWBC:
                if (!AUTH(p->flags & (P_NOTRUST | P_NOPEER), auth))
                        return;         /* authentication error */

                if (!(s.flags & S_BCSTENAB))
                        return;         /* broadcast not enabled */

                p = mobilize(r->srcaddr, r->dstaddr, r->version, M_BCLN,
                    r->keyid, P_EPHEM);
                break;                  /* processing continues */
```

```
        /*
         * Process packet.  Placeholdler only.
         */
        case PROC:
                break;                  /* processing continues */

        /*
         * Invalid mode combination.  We get here only in case of
         * ephemeral associations, so the correct action is simply to
         * toss it.
         */
        case ERR:
                clear(p, X_ERROR);
                return;                 /* invalid mode combination */

        /*
         * No match; just discard the packet.
         */
        case DSCRD:
                return;                 /* orphan abandoned */
        }

        /*
         * Next comes a rigorous schedule of timestamp checking.  If the
         * transmit timestamp is zero, the server is horribly broken.
         */
        if (r->xmt == 0)
                return;                 /* invalid timestamp */

        /*
         * If the transmit timestamp duplicates a previous one, the
         * packet is a replay.
         */
        if (r->xmt == p->xmt)
                return;                 /* duplicate packet */

        /*
         * If this is a broadcast mode packet, skip further checking.
         * If the origin timestamp is zero, the sender has not yet heard
         * from us.  Otherwise, if the origin timestamp does not match
         * the transmit timestamp, the packet is bogus.
         */
```

```
synch = TRUE;
if (r->mode != M_BCST) {
        if (r->org == 0)
                synch = FALSE;  /* unsynchronized */

        else if (r->org != p->xmt)
                synch = FALSE;  /* bogus packet */
}

/*
 * Update the origin and destination timestamps.  If
 * unsynchronized or bogus, abandon ship.
 */
p->org = r->xmt;
p->rec = r->dst;
if (!synch)
        return;                 /* unsynch */

/*
 * The timestamps are valid and the receive packet matches the
 * last one sent.  If the packet is a crypto-NAK, the server
 * might have just changed keys.  We demobilize the association
 * and wait for better times.
 */
if (auth == A_CRYPTO) {
        clear(p, X_CRYPTO);
        return;                 /* crypto-NAK */
}

/*
 * If the association is authenticated, the key ID is nonzero
 * and received packets must be authenticated.  This is designed
 * to avoid a bait-and-switch attack, which was possible in past
 * versions.
 */
if (!AUTH(p->keyid || (p->flags & P_NOTRUST), auth))
        return;                 /* bad auth */
```

```
        /*
         * Everything possible has been done to validate the timestamps
         * and prevent bad guys from disrupting the protocol or
         * injecting bogus data.  Earn some revenue.
         */
        packet(p, r);
}
```

A.5.1.1. packet()

```
/*
 * packet() - process packet and compute offset, delay, and
 * dispersion.
 */
void
packet(
        struct p *p,            /* peer structure pointer */
        struct r *r             /* receive packet pointer */
        )
{
        double  offset;         /* sample offsset */
        double  delay;          /* sample delay */
        double  disp;           /* sample dispersion */

        /*
         * By golly the packet is valid.  Light up the remaining header
         * fields.  Note that we map stratum 0 (unspecified) to MAXSTRAT
         * to make stratum comparisons simpler and to provide a natural
         * interface for radio clock drivers that operate for
         *  convenience at stratum 0.
         */
        p->leap = r->leap;
        if (r->stratum == 0)
                p->stratum = MAXSTRAT;
        else
                p->stratum = r->stratum;
        p->pmode = r->mode;
        p->ppoll = r->poll;
        p->rootdelay = FP2D(r->rootdelay);
        p->rootdisp = FP2D(r->rootdisp);
        p->refid = r->refid;
        p->reftime = r->reftime;

        /*
         * Verify the server is synchronized with valid stratum and
         * reference time not later than the transmit time.
         */
```

```
        if (p->leap == NOSYNC || p->stratum >= MAXSTRAT)
                return;                 /* unsynchronized */

        /*
         * Verify valid root distance.
         */
        if (r->rootdelay / 2 + r->rootdisp >= MAXDISP || p->reftime >
            r->xmt)
                return;                 /* invalid header values */

        poll_update(p, p->hpoll);
        p->reach |= 1;

        /*
         * Calculate offset, delay and dispersion, then pass to the
         * clock filter.  Note carefully the implied processing.  The
         * first-order difference is done directly in 64-bit arithmetic,
         * then the result is converted to floating double.  All further
         * processing is in floating-double arithmetic with rounding
         * done by the hardware.  This is necessary in order to avoid
         * overflow and preserve precision.
         *
         * The delay calculation is a special case.  In cases where the
         * server and client clocks are running at different rates and
         * with very fast networks, the delay can appear negative.  In
         * order to avoid violating the Principle of Least Astonishment,
         * the delay is clamped not less than the system precision.
         */
        if (p->pmode == M_BCST) {
                offset = LFP2D(r->xmt - r->dst);
                delay = BDELAY;
                disp = LOG2D(r->precision) + LOG2D(s.precision) + PHI *
                    2 * BDELAY;
        } else {
                offset = (LFP2D(r->rec - r->org) + LFP2D(r->dst -
                    r->xmt)) / 2;
                delay = max(LFP2D(r->dst - r->org) - LFP2D(r->rec -
                    r->xmt), LOG2D(s.precision));
                disp = LOG2D(r->precision) + LOG2D(s.precision) + PHI *
                    LFP2D(r->dst - r->org);
        }
        clock_filter(p, offset, delay, disp);
}
```

A.5.2. clock_filter()

```
/*
 * clock_filter(p, offset, delay, dispersion) - select the best from the
 * latest eight delay/offset samples.
 */
void
clock_filter(
        struct p *p,            /* peer structure pointer */
        double  offset,         /* clock offset */
        double  delay,          /* roundtrip delay */
        double  disp            /* dispersion */
        )
{
        struct f f[NSTAGE];     /* sorted list */
        double  dtemp;
        int     i;

        /*
         * The clock filter contents consist of eight tuples (offset,
         * delay, dispersion, time).  Shift each tuple to the left,
         * discarding the leftmost one.  As each tuple is shifted,
         * increase the dispersion since the last filter update.  At the
         * same time, copy each tuple to a temporary list.  After this,
         * place the (offset, delay, disp, time) in the vacated
         * rightmost tuple.
         */
        for (i = 1; i < NSTAGE; i++) {
                p->f[i] = p->f[i - 1];
                p->f[i].disp += PHI * (c.t - p->t);
                f[i] = p->f[i];
        }
        p->f[0].t = c.t;
        p->f[0].offset = offset;
        p->f[0].delay = delay;
        p->f[0].disp = disp;
        f[0] = p->f[0];

        /*
         * Sort the temporary list of tuples by increasing f[].delay.
         * The first entry on the sorted list represents the best
         * sample, but it might be old.
         */
        dtemp = p->offset;
        p->offset = f[0].offset;
        p->delay = f[0].delay;
        for (i = 0; i < NSTAGE; i++) {
                p->disp += f[i].disp / (2 ^ (i + 1));
```

```
                p->jitter += SQUARE(f[i].offset - f[0].offset);
        }
        p->jitter = max(SQRT(p->jitter), LOG2D(s.precision));

        /*
         * Prime directive: use a sample only once and never a sample
         * older than the latest one, but anything goes before first
         * synchronized.
         */
        if (f[0].t - p->t <= 0 && s.leap != NOSYNC)
                return;

        /*
         * Popcorn spike suppressor.  Compare the difference between the
         * last and current offsets to the current jitter.  If greater
         * than SGATE (3) and if the interval since the last offset is
         * less than twice the system poll interval, dump the spike.
         * Otherwise, and if not in a burst, shake out the truechimers.
         */
        if (fabs(p->offset - dtemp) > SGATE * p->jitter && (f[0].t -
            p->t) < 2 * s.poll)
                return;

        p->t = f[0].t;
        if (p->burst == 0)
                clock_select();
        return;
}

/*
 * fit() - test if association p is acceptable for synchronization
 */
int
fit(
        struct p *p             /* peer structure pointer */
        )
{
        /*
         * A stratum error occurs if (1) the server has never been
         * synchronized, (2) the server stratum is invalid.
         */
        if (p->leap == NOSYNC || p->stratum >= MAXSTRAT)
                return (FALSE);
```

```
        /*
         * A distance error occurs if the root distance exceeds the
         * distance threshold plus an increment equal to one poll
         * interval.
         */
        if (root_dist(p) > MAXDIST + PHI * LOG2D(s.poll))
                return (FALSE);

        /*
         * A loop error occurs if the remote peer is synchronized to the
         * local peer or the remote peer is synchronized to the current
         * system peer.  Note this is the behavior for IPv4; for IPv6
         * the MD5 hash is used instead.
         */
        if (p->refid == p->dstaddr || p->refid == s.refid)
                return (FALSE);

        /*
         * An unreachable error occurs if the server is unreachable.
         */
        if (p->reach == 0)
                return (FALSE);

        return (TRUE);
}

/*
 * clear() - reinitialize for persistent association, demobilize
 * for ephemeral association.
 */
void
clear(
        struct p *p,            /* peer structure pointer */
        int     kiss            /* kiss code */
        )
{
        int i;

        /*
         * The first thing to do is return all resources to the bank.
         * Typical resources are not detailed here, but they include
         * dynamically allocated structures for keys, certificates, etc.
         * If an ephemeral association and not initialization, return
         * the association memory as well.
         */
        /* return resources */
        if (s.p == p)
                s.p = NULL;
```

```
        if (kiss != X_INIT && (p->flags & P_EPHEM)) {
                free(p);
                return;
        }

        /*
         * Initialize the association fields for general reset.
         */
        memset(BEGIN_CLEAR(p), LEN_CLEAR, 0);
        p->leap = NOSYNC;
        p->stratum = MAXSTRAT;
        p->ppoll = MAXPOLL;
        p->hpoll = MINPOLL;
        p->disp = MAXDISP;
        p->jitter = LOG2D(s.precision);
        p->refid = kiss;
        for (i = 0; i < NSTAGE; i++)
                p->f[i].disp = MAXDISP;

        /*
         * Randomize the first poll just in case thousands of broadcast
         * clients have just been stirred up after a long absence of the
         * broadcast server.
         */
        p->outdate = p->t = c.t;
        p->nextdate = p->outdate + (random() & ((1 << MINPOLL) - 1));
}
```

A.5.3. fast_xmit()

```
/*
 * fast_xmit() - transmit a reply packet for receive packet r
 */
void
fast_xmit(
        struct r *r,            /* receive packet pointer */
        int     mode,           /* association mode */
        int     auth            /* authentication code */
        )
{
        struct x x;

        /*
         * Initialize header and transmit timestamp.  Note that the
         * transmit version is copied from the receive version.  This is
         * for backward compatibility.
         */
```

```
        x.version = r->version;
        x.srcaddr = r->dstaddr;
        x.dstaddr = r->srcaddr;
        x.leap = s.leap;
        x.mode = mode;
        if (s.stratum == MAXSTRAT)
                x.stratum = 0;
        else
                x.stratum = s.stratum;
        x.poll = r->poll;
        x.precision = s.precision;
        x.rootdelay = D2FP(s.rootdelay);
        x.rootdisp = D2FP(s.rootdisp);
        x.refid = s.refid;
        x.reftime = s.reftime;
        x.org = r->xmt;
        x.rec = r->dst;
        x.xmt = get_time();

        /*
         * If the authentication code is A.NONE, include only the
         * header; if A.CRYPTO, send a crypto-NAK; if A.OK, send a valid
         * MAC.  Use the key ID in the received packet and the key in
         * the local key cache.
         */
        if (auth != A_NONE) {
                if (auth == A_CRYPTO) {
                        x.keyid = 0;
                } else {
                        x.keyid = r->keyid;
                        x.dgst = md5(x.keyid);
                }
        }
        xmit_packet(&x);
}
```

A.5.4. access()

```
 /*
  * access() - determine access restrictions
  */
 int
 access(
         struct r *r             /* receive packet pointer */
         )
```

```
 {
         /*
          * The access control list is an ordered set of tuples
          * consisting of an address, mask, and restrict word containing
          * defined bits.  The list is searched for the first match on
          * the source address (r->srcaddr) and the associated restrict
          * word is returned.
          */
         return (/* access bits */ 0);
 }
```

A.5.5. System Process

A.5.5.1. clock_select()

```
/*
 * clock_select() - find the best clocks
 */
void
clock_select() {
       struct p *p, *osys;     /* peer structure pointers */
       double  low, high;      /* correctness interval extents */
       int     allow, found, chime; /* used by intersection algorithm */
       int     n, i, j;

        /*
         * We first cull the falsetickers from the server population,
         * leaving only the truechimers.  The correctness interval for
         * association p is the interval from offset - root_dist() to
         * offset + root_dist().  The object of the game is to find a
         * majority clique; that is, an intersection of correctness
         * intervals numbering more than half the server population.
         *
         * First, construct the chime list of tuples (p, type, edge) as
         * shown below, then sort the list by edge from lowest to
         * highest.
         */
        osys = s.p;
        s.p = NULL;
        n = 0;
        while (fit(p)) {
                s.m[n].p = p;
                s.m[n].type = +1;
                s.m[n].edge = p->offset + root_dist(p);
                n++;
                s.m[n].p = p;
                s.m[n].type = 0;
                s.m[n].edge = p->offset;
```

```
                n++;
                s.m[n].p = p;
                s.m[n].type = -1;
                s.m[n].edge = p->offset - root_dist(p);
                n++;
        }

        /*
         * Find the largest contiguous intersection of correctness
         * intervals.  Allow is the number of allowed falsetickers;
         * found is the number of midpoints.  Note that the edge values
         * are limited to the range +-(2 ^ 30) < +-2e9 by the timestamp
         * calculations.
         */
        low = 2e9; high = -2e9;
        for (allow = 0; 2 * allow < n; allow++) {

                /*
                 * Scan the chime list from lowest to highest to find
                 * the lower endpoint.
                 */
                found = 0;
                chime = 0;
                for (i = 0; i < n; i++) {
                        chime -= s.m[i].type;
                        if (chime >= n - found) {
                                low = s.m[i].edge;
                                break;
                        }
                        if (s.m[i].type == 0)
                                found++;
                }

                /*
                 * Scan the chime list from highest to lowest to find
                 * the upper endpoint.
                 */
                chime = 0;
                for (i = n - 1; i >= 0; i--) {
                        chime += s.m[i].type;
                        if (chime >= n - found) {
                                high = s.m[i].edge;
                                break;
                        }
                        if (s.m[i].type == 0)
                                found++;
                }
```

```
                /*
                 * If the number of midpoints is greater than the number
                 * of allowed falsetickers, the intersection contains at
                 * least one truechimer with no midpoint.  If so,
                 * increment the number of allowed falsetickers and go
                 * around again.  If not and the intersection is
                 * non-empty, declare success.
                 */
                if (found > allow)
                        continue;

                if (high > low)
                        break;
        }

        /*
         * Clustering algorithm.  Construct a list of survivors (p,
         * metric) from the chime list, where metric is dominated first
         * by stratum and then by root distance.  All other things being
         * equal, this is the order of preference.
         */
        s.n = 0;
        for (i = 0; i < n; i++) {
                if (s.m[i].edge < low || s.m[i].edge > high)
                        continue;

                p = s.m[i].p;
                s.v[n].p = p;
                s.v[n].metric = MAXDIST * p->stratum + root_dist(p);
                s.n++;
        }

        /*
         * There must be at least NSANE survivors to satisfy the
         * correctness assertions.  Ordinarily, the Byzantine criteria
         * require four survivors, but for the demonstration here, one
         * is acceptable.
         */
        if (s.n < NSANE)
                return;

        /*
         * For each association p in turn, calculate the selection
         * jitter p->sjitter as the square root of the sum of squares
         * (p->offset - q->offset) over all q associations.  The idea is
         * to repeatedly discard the survivor with maximum selection
         * jitter until a termination condition is met.
         */
```

```
        while (1) {
                struct p *p, *q, *qmax; /* peer structure pointers */
                double  max, min, dtemp;

                max = -2e9; min = 2e9;
                for (i = 0; i < s.n; i++) {
                        p = s.v[i].p;
                        if (p->jitter < min)
                                min = p->jitter;
                        dtemp = 0;
                        for (j = 0; j < n; j++) {
                                q = s.v[j].p;
                                dtemp += SQUARE(p->offset - q->offset);
                        }
                        dtemp = SQRT(dtemp);
                        if (dtemp > max) {
                                max = dtemp;
                                qmax = q;
                        }
                }

                /*
                 * If the maximum selection jitter is less than the
                 * minimum peer jitter, then tossing out more survivors
                 * will not lower the minimum peer jitter, so we might
                 * as well stop.  To make sure a few survivors are left
                 * for the clustering algorithm to chew on, we also stop
                 * if the number of survivors is less than or equal to
                 * NMIN (3).
                 */
                if (max < min || n <= NMIN)
                        break;

                /*
                 * Delete survivor qmax from the list and go around
                 * again.
                 */
                s.n--;
        }

        /*
         * Pick the best clock.  If the old system peer is on the list
         * and at the same stratum as the first survivor on the list,
         * then don't do a clock hop.  Otherwise, select the first
         * survivor on the list as the new system peer.
         */
        if (osys->stratum == s.v[0].p->stratum)
                s.p = osys;
```

```
        else
                s.p = s.v[0].p;
        clock_update(s.p);
}
```

A.5.5.2. root_dist()

```
/*
 * root_dist() - calculate root distance
 */
double
root_dist(
        struct p *p             /* peer structure pointer */
        )
{

        /*
         * The root synchronization distance is the maximum error due to
         * all causes of the local clock relative to the primary server.
         * It is defined as half the total delay plus total dispersion
         * plus peer jitter.
         */
        return (max(MINDISP, p->rootdelay + p->delay) / 2 +
            p->rootdisp + p->disp + PHI * (c.t - p->t) + p->jitter);
}
```

A.5.5.3. accept()

```
/*
 * accept() - test if association p is acceptable for synchronization
 */
int
accept(
        struct p *p             /* peer structure pointer */
        )
{
        /*
         * A stratum error occurs if (1) the server has never been
         * synchronized, (2) the server stratum is invalid.
         */
        if (p->leap == NOSYNC || p->stratum >= MAXSTRAT)
                return (FALSE);

        /*
         * A distance error occurs if the root distance exceeds the
         * distance threshold plus an increment equal to one poll
         * interval.
         */
```

```
        if (root_dist(p) > MAXDIST + PHI * LOG2D(s.poll))
                return (FALSE);

        /*
         * A loop error occurs if the remote peer is synchronized to the
         * local peer or the remote peer is synchronized to the current
         * system peer.  Note this is the behavior for IPv4; for IPv6
         * the MD5 hash is used instead.
         */
        if (p->refid == p->dstaddr || p->refid == s.refid)
                return (FALSE);

        /*
         * An unreachable error occurs if the server is unreachable.
         */
        if (p->reach == 0)
                return (FALSE);

        return (TRUE);
}
```

A.5.5.4. clock_update()

```
/*
 * clock_update() - update the system clock
 */
void
clock_update(
        struct p *p             /* peer structure pointer */
        )
{
        double dtemp;

        /*
         * If this is an old update, for instance, as the result of a
         * system peer change, avoid it.  We never use an old sample or
         * the same sample twice.
         */
        if (s.t >= p->t)
                return;

        /*
         * Combine the survivor offsets and update the system clock; the
         * local_clock() routine will tell us the good or bad news.
         */
        s.t = p->t;
        clock_combine();
        switch (local_clock(p, s.offset)) {
```

```
        /*
         * The offset is too large and probably bogus.  Complain to the
         * system log and order the operator to set the clock manually
         * within PANIC range.  The reference implementation includes a
         * command line option to disable this check and to change the
         * panic threshold from the default 1000 s as required.
         */
        case PANIC:
                exit (0);

        /*
         * The offset is more than the step threshold (0.125 s by
         * default).  After a step, all associations now have
         * inconsistent time values, so they are reset and started
         * fresh.  The step threshold can be changed in the reference
         * implementation in order to lessen the chance the clock might
         * be stepped backwards.  However, there may be serious
         * consequences, as noted in the white papers at the NTP project
         * site.
         */
        case STEP:
                while (/* all associations */ 0)
                        clear(p, X_STEP);
                s.stratum = MAXSTRAT;
                s.poll = MINPOLL;
                break;

        /*
         * The offset was less than the step threshold, which is the
         * normal case.  Update the system variables from the peer
         * variables.  The lower clamp on the dispersion increase is to
         * avoid timing loops and clockhopping when highly precise
         * sources are in play.  The clamp can be changed from the
         * default .01 s in the reference implementation.
         */
        case SLEW:
                s.leap = p->leap;
                s.stratum = p->stratum + 1;
                s.refid = p->refid;
                s.reftime = p->reftime;
                s.rootdelay = p->rootdelay + p->delay;
                dtemp = SQRT(SQUARE(p->jitter) + SQUARE(s.jitter));
                dtemp += max(p->disp + PHI * (c.t - p->t) +
                    fabs(p->offset), MINDISP);
                s.rootdisp = p->rootdisp + dtemp;
                break;
```

```
        /*
         * Some samples are discarded while, for instance, a direct
         * frequency measurement is being made.
         */
        case IGNORE:
                break;
        }
}
```

A.5.5.5. clock_combine()

```
/*
 * clock_combine() - combine offsets
 */
void
clock_combine()
{
        struct p *p;            /* peer structure pointer */
        double x, y, z, w;
        int     i;

        /*
         * Combine the offsets of the clustering algorithm survivors
         * using a weighted average with weight determined by the root
         * distance.  Compute the selection jitter as the weighted RMS
         * difference between the first survivor and the remaining
         * survivors.  In some cases, the inherent clock jitter can be
         * reduced by not using this algorithm, especially when frequent
         * clockhopping is involved.  The reference implementation can
         * be configured to avoid this algorithm by designating a
         * preferred peer.
         */
        y = z = w = 0;
        for (i = 0; s.v[i].p != NULL; i++) {
                p = s.v[i].p;
                x = root_dist(p);
                y += 1 / x;
                z += p->offset / x;
                w += SQUARE(p->offset - s.v[0].p->offset) / x;
        }
        s.offset = z / y;
        s.jitter = SQRT(w / y);
}
```

A.5.5.6. local_clock()

```
/*
 * Clock discipline parameters and constants
 */
#define STEPT           .128    /* step threshold (s) */
#define WATCH           900     /* stepout threshold (s) */
#define PANICT          1000    /* panic threshold (s) */
#define PLL             65536   /* PLL loop gain */
#define FLL             MAXPOLL + 1 /* FLL loop gain */
#define AVG             4       /* parameter averaging constant */
#define ALLAN           1500    /* compromise Allan intercept (s) */
#define LIMIT           30      /* poll-adjust threshold */
#define MAXFREQ         500e-6  /* frequency tolerance (500 ppm) */
#define PGATE           4       /* poll-adjust gate */

/*
 * local_clock() - discipline the local clock
 */
int                             /* return code */
local_clock(
        struct p *p,            /* peer structure pointer */
        double  offset          /* clock offset from combine() */
        )
{
        int     state;          /* clock discipline state */
        double  freq;           /* frequency */
        double  mu;             /* interval since last update */
        int     rval;
        double  etemp, dtemp;

        /*
         * If the offset is too large, give up and go home.
         */
        if (fabs(offset) > PANICT)
                return (PANIC);

        /*
         * Clock state machine transition function.  This is where the
         * action is and defines how the system reacts to large time
         * and frequency errors.  There are two main regimes: when the
         * offset exceeds the step threshold and when it does not.
         */
        rval = SLEW;
        mu = p->t - s.t;
        freq = 0;
        if (fabs(offset) > STEPT) {
                switch (c.state) {
```

```
                /*
                 * In S_SYNC state, we ignore the first outlier and
                 * switch to S_SPIK state.
                 */
                case SYNC:
                        state = SPIK;
                        return (rval);

                /*
                 * In S_FREQ state, we ignore outliers and inliers.  At
                 * the first outlier after the stepout threshold,
                 * compute the apparent frequency correction and step
                 * the time.
                 */
                case FREQ:
                        if (mu < WATCH)
                                return (IGNORE);

                        freq = (offset - c.offset) / mu;
                        /* fall through to S_SPIK */

                /*
                 * In S_SPIK state, we ignore succeeding outliers until
                 * either an inlier is found or the stepout threshold is
                 * exceeded.
                 */
                case SPIK:
                        if (mu < WATCH)
                                return (IGNORE);

                        /* fall through to default */

                /*
                 * We get here by default in S_NSET and S_FSET states
                 * and from above in S_FREQ state.  Step the time and
                 * clamp down the poll interval.
                 *
                 * In S_NSET state, an initial frequency correction is
                 * not available, usually because the frequency file has
                 * not yet been written.  Since the time is outside the
                 * capture range, the clock is stepped.  The frequency
                 * will be set directly following the stepout interval.
                 *
                 * In S_FSET state, the initial frequency has been set
                 * from the frequency file.  Since the time is outside
                 * the capture range, the clock is stepped immediately,
                 * rather than after the stepout interval.  Guys get
                 * nervous if it takes 17 minutes to set the clock for
```

```
                 * the first time.
                 *
                 * In S_SPIK state, the stepout threshold has expired
                 * and the phase is still above the step threshold.
                 * Note that a single spike greater than the step
                 * threshold is always suppressed, even at the longer
                 * poll intervals.
                 */
                default:

                        /*
                         * This is the kernel set time function, usually
                         * implemented by the Unix settimeofday() system
                         * call.
                         */
                        step_time(offset);
                        c.count = 0;
                        s.poll = MINPOLL;
                        rval = STEP;
                        if (state == NSET) {
                                rstclock(FREQ, p->t, 0);
                                return (rval);
                        }
                        break;
                }
                rstclock(SYNC, p->t, 0);
        } else {

                /*
                 * Compute the clock jitter as the RMS of exponentially
                 * weighted offset differences.  This is used by the
                 * poll-adjust code.
                 */
                etemp = SQUARE(c.jitter);
                dtemp = SQUARE(max(fabs(offset - c.last),
                    LOG2D(s.precision)));
                c.jitter = SQRT(etemp + (dtemp - etemp) / AVG);
                switch (c.state) {

                /*
                 * In S_NSET state, this is the first update received
                 * and the frequency has not been initialized.  The
                 * first thing to do is directly measure the oscillator
                 * frequency.
                 */
                case NSET:
                        rstclock(FREQ, p->t, offset);
                        return (IGNORE);
```

```
/*
 * In S_FSET state, this is the first update and the
 * frequency has been initialized.  Adjust the phase,
 * but don't adjust the frequency until the next update.
 */
case FSET:
        rstclock(SYNC, p->t, offset);
        break;

/*
 * In S_FREQ state, ignore updates until the stepout
 * threshold.  After that, correct the phase and
 * frequency and switch to S_SYNC state.
 */
case FREQ:
        if (c.t - s.t < WATCH)
                return (IGNORE);

        freq = (offset - c.offset) / mu;
        break;

/*
 * We get here by default in S_SYNC and S_SPIK states.
 * Here we compute the frequency update due to PLL and
 * FLL contributions.
 */
default:

        /*
         * The FLL and PLL frequency gain constants
         * depending on the poll interval and Allan
         * intercept.  The FLL is not used below one
         * half the Allan intercept.  Above that the
         * loop gain increases in steps to 1 / AVG.
         */
        if (LOG2D(s.poll) > ALLAN / 2) {
                etemp = FLL - s.poll;
                if (etemp < AVG)
                        etemp = AVG;
                freq += (offset - c.offset) / (max(mu,
                    ALLAN) * etemp);
        }
```

```
                        /*
                         * For the PLL the integration interval
                         * (numerator) is the minimum of the update
                         * interval and poll interval.  This allows
                         * oversampling, but not undersampling.
                         */
                        etemp = min(mu, LOG2D(s.poll));
                        dtemp = 4 * PLL * LOG2D(s.poll);
                        freq += offset * etemp / (dtemp * dtemp);
                        rstclock(SYNC, p->t, offset);
                        break;
                }
        }

        /*
         * Calculate the new frequency and frequency stability (wander).
         * Compute the clock wander as the RMS of exponentially weighted
         * frequency differences.  This is not used directly, but can,
         * along with the jitter, be a highly useful monitoring and
         * debugging tool.
         */
        freq += c.freq;
        c.freq = max(min(MAXFREQ, freq), -MAXFREQ);
        etemp = SQUARE(c.wander);
        dtemp = SQUARE(freq);
        c.wander = SQRT(etemp + (dtemp - etemp) / AVG);

        /*
         * Here we adjust the poll interval by comparing the current
         * offset with the clock jitter.  If the offset is less than the
         * clock jitter times a constant, then the averaging interval is
         * increased; otherwise, it is decreased.  A bit of hysteresis
         * helps calm the dance.  Works best using burst mode.
         */
        if (fabs(c.offset) < PGATE * c.jitter) {
                c.count += s.poll;
                if (c.count > LIMIT) {
                        c.count = LIMIT;
                        if (s.poll < MAXPOLL) {
                                c.count = 0;
                                s.poll++;
                        }
                }
        } else {
                c.count -= s.poll << 1;
                if (c.count < -LIMIT) {
                        c.count = -LIMIT;
                        if (s.poll > MINPOLL) {
```

```
                                c.count = 0;
                                s.poll--;
                        }
                }
        }
        return (rval);
}
```

A.5.5.7. rstclock()

```
  /*
   * rstclock() - clock state machine
   */
  void
  rstclock(
          int     state,          /* new state */
          double  offset,         /* new offset */
          double  t               /* new update time */
          )
  {
          /*
           * Enter new state and set state variables.  Note, we use the
           * time of the last clock filter sample, which must be earlier
           * than the current time.
           */
          c.state = state;
          c.last = c.offset = offset;
          s.t = t;
  }
```

A.5.6. Clock Adjust Process

A.5.6.1. clock_adjust()

```
 /*
  * clock_adjust() - runs at one-second intervals
  */
 void
 clock_adjust() {
         double  dtemp;

         /*
          * Update the process time c.t.  Also increase the dispersion
          * since the last update.  In contrast to NTPv3, NTPv4 does not
          * declare unsynchronized after one day, since the dispersion
          * threshold serves this function.  When the dispersion exceeds
          * MAXDIST (1 s), the server is considered unfit for
          * synchronization.
```

```
         */
        c.t++;
        s.rootdisp += PHI;

        /*
         * Implement the phase and frequency adjustments.  The gain
         * factor (denominator) is not allowed to increase beyond the
         * Allan intercept.  It doesn't make sense to average phase
         * noise beyond this point and it helps to damp residual offset
         * at the longer poll intervals.
         */
        dtemp = c.offset / (PLL * min(LOG2D(s.poll), ALLAN));
        c.offset -= dtemp;

        /*
         * This is the kernel adjust time function, usually implemented
         * by the Unix adjtime() system call.
         */
        adjust_time(c.freq + dtemp);

        /*
         * Peer timer.  Call the poll() routine when the poll timer
         * expires.
         */
        while (/* all associations */ 0) {
                struct p *p;    /* dummy peer structure pointer */

                if (c.t >= p->nextdate)
                        poll(p);
        }

        /*
         * Once per hour, write the clock frequency to a file.
         */
        /*
         * if (c.t % 3600 == 3599)
         *   write c.freq to file
         */
 }
```

A.5.7. Poll Process

```
   /*
    * Poll process parameters and constants
    */
   #define UNREACH         12      /* unreach counter threshold */
   #define BCOUNT          8       /* packets in a burst */
   #define BTIME           2       /* burst interval (s) */
```

A.5.7.1. poll()

```
/*
 * poll() - determine when to send a packet for association p->
 */
void
poll(
        struct p *p             /* peer structure pointer */
        )
{
        int     hpoll;
        int     oreach;

        /*
         * This routine is called when the current time c.t catches up
         * to the next poll time p->nextdate.  The value p->outdate is
         * the last time this routine was executed.  The poll_update()
         * routine determines the next execution time p->nextdate.
         *
         * If broadcasting, just do it, but only if we are synchronized.
         */
        hpoll = p->hpoll;
        if (p->hmode == M_BCST) {
                p->outdate = c.t;
                if (s.p != NULL)
                        peer_xmit(p);
                poll_update(p, hpoll);
                return;
        }

        /*
         * If manycasting, start with ttl = 1.  The ttl is increased by
         * one for each poll until MAXCLOCK servers have been found or
         * ttl reaches TTLMAX.  If reaching MAXCLOCK, stop polling until
         * the number of servers falls below MINCLOCK, then start all
         * over.
         */
        if (p->hmode == M_CLNT && p->flags & P_MANY) {
                p->outdate = c.t;
                if (p->unreach > BEACON) {
                        p->unreach = 0;
                        p->ttl = 1;
                        peer_xmit(p);
                } else if (s.n < MINCLOCK) {
                        if (p->ttl < TTLMAX)
                                p->ttl++;
                        peer_xmit(p);
                }
```

```
                p->unreach++;
                poll_update(p, hpoll);
                return;
        }
        if (p->burst == 0) {

                /*
                 * We are not in a burst.  Shift the reachability
                 * register to the left.  Hopefully, some time before
                 * the next poll a packet will arrive and set the
                 * rightmost bit.
                 */
                oreach = p->reach;
                p->outdate = c.t;
                p->reach = p->reach << 1;
                if (!(p->reach & 0x7))
                        clock_filter(p, 0, 0, MAXDISP);
                if (!p->reach) {

                        /*
                         * The server is unreachable, so bump the
                         * unreach counter.  If the unreach threshold
                         * has been reached, double the poll interval
                         * to minimize wasted network traffic.  Send a
                         * burst only if enabled and the unreach
                         * threshold has not been reached.
                         */
                        if (p->flags & P_IBURST && p->unreach == 0) {
                                p->burst = BCOUNT;
                        } else if (p->unreach < UNREACH)
                                p->unreach++;
                        else
                                hpoll++;
                        p->unreach++;
                } else {

                        /*
                         * The server is reachable.  Set the poll
                         * interval to the system poll interval.  Send a
                         * burst only if enabled and the peer is fit.
                         */
                        p->unreach = 0;
                        hpoll = s.poll;
                        if (p->flags & P_BURST && fit(p))
                                p->burst = BCOUNT;
                }
        } else {
```

```
                /*
                 * If in a burst, count it down.  When the reply comes
                 * back the clock_filter() routine will call
                 * clock_select() to process the results of the burst.
                 */
                p->burst--;
        }
        /*
         * Do not transmit if in broadcast client mode.
         */
        if (p->hmode != M_BCLN)
                peer_xmit(p);
        poll_update(p, hpoll);
}
```

A.5.7.2. poll_update()

```
/*
 * poll_update() - update the poll interval for association p
 *
 * Note: This routine is called by both the packet() and poll() routine.
 * Since the packet() routine is executed when a network packet arrives
 * and the poll() routine is executed as the result of timeout, a
 * potential race can occur, possibly causing an incorrect interval for
 * the next poll.  This is considered so unlikely as to be negligible.
 */
void
poll_update(
        struct p *p,            /* peer structure pointer */
        int     poll            /* poll interval (log2 s) */
        )
{
        /*
         * This routine is called by both the poll() and packet()
         * routines to determine the next poll time.  If within a burst
         * the poll interval is two seconds.  Otherwise, it is the
         * minimum of the host poll interval and peer poll interval, but
         * not greater than MAXPOLL and not less than MINPOLL.  The
         * design ensures that a longer interval can be preempted by a
         * shorter one if required for rapid response.
         */
        p->hpoll = max(min(MAXPOLL, poll), MINPOLL);
        if (p->burst > 0) {
                if (p->nextdate != c.t)
                        return;
                else
                        p->nextdate += BTIME;
        } else {
```

```
                /*
                 * While not shown here, the reference implementation
                 * randomizes the poll interval by a small factor.
                 */
                p->nextdate = p->outdate + (1 << max(min(p->ppoll,
                    p->hpoll), MINPOLL));
        }

        /*
         * It might happen that the due time has already passed.  If so,
         * make it one second in the future.
         */
        if (p->nextdate <= c.t)
                p->nextdate = c.t + 1;
}
```

A.5.7.3. peer_xmit()

```
/*
 * transmit() - transmit a packet for association p
 */
void
peer_xmit(
        struct p *p             /* peer structure pointer */
        )
{
        struct x x;             /* transmit packet */

        /*
         * Initialize header and transmit timestamp
         */
        x.srcaddr = p->dstaddr;
        x.dstaddr = p->srcaddr;
        x.leap = s.leap;
        x.version = p->version;
        x.mode = p->hmode;
        if (s.stratum == MAXSTRAT)
                x.stratum = 0;
        else
                x.stratum = s.stratum;
        x.poll = p->hpoll;
        x.precision = s.precision;
        x.rootdelay = D2FP(s.rootdelay);
        x.rootdisp = D2FP(s.rootdisp);
        x.refid = s.refid;
        x.reftime = s.reftime;
        x.org = p->org;
        x.rec = p->rec;
```

```
        x.xmt = get_time();
        p->xmt = x.xmt;

        /*
         * If the key ID is nonzero, send a valid MAC using the key ID
         * of the association and the key in the local key cache.  If
         * something breaks, like a missing trusted key, don't send the
         * packet; just reset the association and stop until the problem
         * is fixed.
         */
        if (p->keyid)
                if (/* p->keyid invalid */ 0) {
                        clear(p, X_NKEY);
                        return;
                }
                x.dgst = md5(p->keyid);
        xmit_packet(&x);
}
```

Authors' Addresses

Dr. David L. Mills
University of Delaware
Newark, DE 19716
US

Phone: +1 302 831 8247
EMail: mills@udel.edu

Jim Martin (editor)
Internet Systems Consortium
950 Charter Street
Redwood City, CA 94063
US

Phone: +1 650 423 1378
EMail: jrmii@isc.org

Jack Burbank
The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Road
Laurel, MD 20723-6099
US

Phone: +1 443 778 7127
EMail: jack.burbank@jhuapl.edu

William Kasch
The Johns Hopkins University Applied Physics Laboratory
11100 Johns Hopkins Road
Laurel, MD 20723-6099
US

Phone: +1 443 778 7463
EMail: william.kasch@jhuapl.edu