NTP Working Group
Technical Report 06-6-1

David L. Mills
University of Delaware
June 2006

Network Time Protocol Version 4
Reference and Implementation Guide

Abstract

This document describes the Network Time Protocol Version 4 (NTPv4), which is widely used to synchronize the time for Internet hosts, routers and ancillary devices to Coordinated Universal Time (UTC) as disseminated by national standards laboratories. It describes the core architecture, protocol, state machine, data structures and algorithms. It explains the fundamental on-wire protocol used to exchange time values between peers, servers and clients. It summarizes the clock offset, roundtrip delay and various other statistics used by the mitigation algorithms to calculate the maximum error and nominal error inherent in computing these values. It describes several changes from Version 3 of NTP (NTPv3) originally described in RFC 1305, including the introduction of a modified protocol header to accommodate Internet Protocol Version 6 and a new header extension field to support the Autokey public key authentication scheme.

This document is based on the reference implementation available at www.ntp.org. It is intended as a reference and implementation guide, not as a formal standard. The main body of the document describes the basic model data structures and algorithms necessary for an implementation which can interoperate properly with another implementation faithful to this model. This document describes a number of crafted mitigation algorithms which can improve the accuracy and stability of the timekeeping function, especially in NTP subnets with many servers and clients. It also describes the clock discipline function used to adjust the system clock in time and frequency to agree with the available sources of synchronization.

As an implementation aid, a code skeleton for the reference implementation is presented in an appendix. It includes most of the data structures and algorithms of that program, but certain features, such as the control and monitoring protocol, Autokey public key authentication scheme, huff-’n-puff scheme and server discovery schemes are not included. These are discussed in companion documents on the Web and in print.

Keywords: network time synchronization, computer time synchronization, time synchronization protocol

Sponsored by: Naval Surface Weapons Center (Dahlgren) Contract N00178-04-1-9001.

Table of Contents

List of Figures

List of Tables

1. Introduction

This document is a reference and implementation guide for the Network Time Protocol Version 4 (NTPv4), which is widely used to synchronize the system clocks among a set of distributed time servers and clients. This document defines the core architecture, protocol, state machines, data structures and algorithms. It is intended primarily for developers and not as a formal standard. This and related documents collectively replace the Network Time Protocol Version 3 (NTPv3) specification RFC 1305 [6] and previous versions of that specification. The core protocol continues to be compatible with all prior versions except the original (unnumbered) version of the protocol. While certain minor changes have been made in some protocol header fields, these do not affect the interoperability between NTPv4 and previous versions.

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. Crafted algorithms mitigate errors that may result from network disruptions, server failures and possible hostile action. Servers and clients are configured as a forest where time values flow from the primary servers at the root via branching secondary servers toward clients at the leaves of the forest.

The NTPv4 reference implementation available at www.ntp.org is consistent with the model described in this document. However, neither this implementation nor this document is intended as a definitive formal standard. Formal standards documents consistent with IETF requirements are in process at the time of writing. When a conflict is found between this document and the formal standards documents, the latter prevail.

The NTPv4 design overcomes significant shortcomings in the NTPv3 design, corrects certain bugs and incorporates new features. In particular, the reference implementation uses floating double data types throughout, except for the first-order timestamp differences required to calculate offset and delay. The time resolution is better than one nanosecond and frequency resolution better than one nanosecond per second. Additional improvements include a new clock discipline algorithm which 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 within a few tens of milliseconds with poll intervals up to 36 hours.

The main body of this document describes only the core protocol and data structures necessary to interoperate between conforming implementations. Additional detail is provided in the form of a skeleton program included as an appendix. This program includes data structures and code segments for the core algorithms and in addition the mitigation algorithms used to enhance reliability and accuracy. While the skeleton 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. While the NTPv3 symmetric key authentication scheme described in this document carries over from NTPv3, the Autokey public key authentication scheme new to NTPv4 is described in [3].

The NTP protocol includes the modes of operation described in Section 2 using the data types described in Section 5 and the data structures in Section 6. The implementation model described in Section 4 is based on a multiple-process, threaded architecture, although other architectures could be used as well. The on-wire protocol described in Section 7 is based on a returnable-time design which depends only on measured clock offsets, but does not require reliable message delivery. 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 remaining sections of this document define the data structures and algorithms suitable for a fully featured NTPv4 implementation. Details specific to NTP packet formats commonly used with the User Datagram Protocol (UDP) are presented in Appendix A. Appendix B contains the code skeleton with definitions, structures and code segments that represent the basic structure of the reference implementation.

2. Modes of Operation

An NTP implementation operates as a *primary server*, *secondary server* or *client*. A primary server is synchronized directly to a reference clock, such as a GPS receiver or telephone modem service. A client is synchronized 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 claiming full NTPv4 compliance must implement the entire suite of algorithms described in this document. In order to maintain stability in large NTP subnets, secondary servers must be fully NTPv4 compliant.

Primary servers and clients complying with a subset of NTP, called the Simple Network Time Protocol (SNTPv4) [4], do not need to implement all algorithms. SNTP is intended for primary servers equipped with a single reference clock, as well as 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 mixed and matched in NTP subnets.

Servers retain no state after returning the response to a client packet; clients retain state in the form of a data structure called an *association*. Persistent associations are mobilized when the service starts and are never demobilized. Ephemeral associations are mobilized during operation, such as upon the arrival of a broadcast packet, and demobilized by timeout or error. Preemptable associations are mobilized when or after the service starts and demobilized when deemed no longer useful for synchronization. The code skeleton and reference implementation summarized in Appendix B includes suitable algorithms for ephemeral and preemptable associations, but they are not discussed in the main body of this document.

Association Mode	Assoc. Mode	Packet Mode	
Symmetric Active	1	1 or 2	
Symmetric Passive	2	1	
Client	3	4	
Server	4	3	
Broadcast Server	5	5	5
Broadcast Client	6	na	

Table 1. Association and Packet Modes

There are three NTP protocol variants, *symmetric*, *client/server* and *broadcast*. Each is associated with an association mode as shown in Table 1. In the client/server variant a client association sends mode-3 (client) packets to a server, which returns mode-4 (server) packets. Servers provide synchronization to one or more clients, but do not accept synchronization from them. A server can also be a reference clock which obtains time directly from a standard source such as a GPS receiver or telephone modem service. We say that 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 symmetric-active association sends mode-1 (symmetric-active) packets to a symmetric-active peer association. Alternatively, a symmetric-passive association can be mobilized upon arrival of a mode-1 packet. That association sends mode-2 (symmetric-passive) packets and persists until error or timeout. We say that peers both *push* and *pull* synchronization to and from each other. For the purposes of this document, a peer operates like a client, so a reference to client implies peer as well.

In the broadcast variant a broadcast server association sends periodic mode-5 (broadcast) packets which are received by multiple mode-6 (broadcast client) associations. It is useful to provide an initial volley where the client operating in mode 3 exchanges several packets with the server in order to calibrate the propagation delay and to run the Autokey security protocol, after which the client reverts to mode 6. We say that broadcast servers *push* synchronization to willing consumers.

Following conventions established by the telephone industry, the level of each server in the hierarchy is defined by a number called the *stratum*, with the primary servers assigned stratum one and the secondary servers at each level assigned one greater than the preceding level. As the stratum increases from one, the accuracies achievable degrade somewhat depending on the particular network path and system clock stability. It is useful to assume that mean errors, and thus a metric called the synchronization distance, increase approximately in proportion to the stratum and measured roundtrip delay.

Drawing from the experience of the telephone industry, which learned such lessons at considerable cost, the subnet topology should be organized to produce the lowest synchronization distances, but must never be allowed to form a loop. In NTP the subnet topology is determined using a variant of the Bellman-Ford distributed routing algorithm, which computes the shortest-distance spanning tree rooted on the primary servers. As a result of this design, the algorithm automatically reorganizes the subnet to produce the most accurate and reliable time, even when one or more primary or secondary servers or the network paths between them fail.

3. Definitions

A number of terms used throughout this document have a precise technical definition. A *timescale* is a frame of reference where time is expressed as the value of a monotonic-increasing binary counter with an indefinite number of bits. It counts in seconds and fraction with the decimal point somewhere in the middle. 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 operating system kernel. 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 it takes place. Dates are ephemeral values which always increase in step with reality and are designated with upper case *T* in this document. It is convenient to define another timescale coincident with the running time of the NTP program that provides the synchronization function. This is convenient in order to determine intervals for the various repetitive functions like poll events. Running time is usually designated with lower case *t*.

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 context. Let $T(t)$ be the time offset, $R(t)$ the frequency offset, $D(t)$ the ageing 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 after some interval *t* is

$$T(t + t_0) \;=\; T(t_0) + R(t_0)(t + t_0) + \frac{1}{2}D(t_0)(t + t_0)^2 + e\,, \tag{1}$$

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 ordinary neglected for computer oscillators. In this document all time values are in seconds (s) and all frequency values 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 $1e^{-6}$ s/s.

It is important in computer timekeeping applications to assess the performance of the timekeeping function. The NTP performance model includes four statistics which are updated each time a client makes a measurement with a server. The *offset* θ represents the maximum-likelihood time offset of the server clock relative to the system clock. The *delay* δ represents the roundtrip delay between the client and server. The *dispersion* ε represents the maximum error inherent in the measurement. It increases at a rate equal to the maximum disciplined system clock *frequency tolerance* Φ, typically 15 PPM. The *jitter* φ, defined as the root-mean-square (RMS) average of the most recent time offset differences, represents the nominal error in estimating θ.

While the θ, δ, ε, and φ statistics represent measurements of the system clock relative to the each server clock separately, the NTP protocol includes mechanisms to combine the statistics of

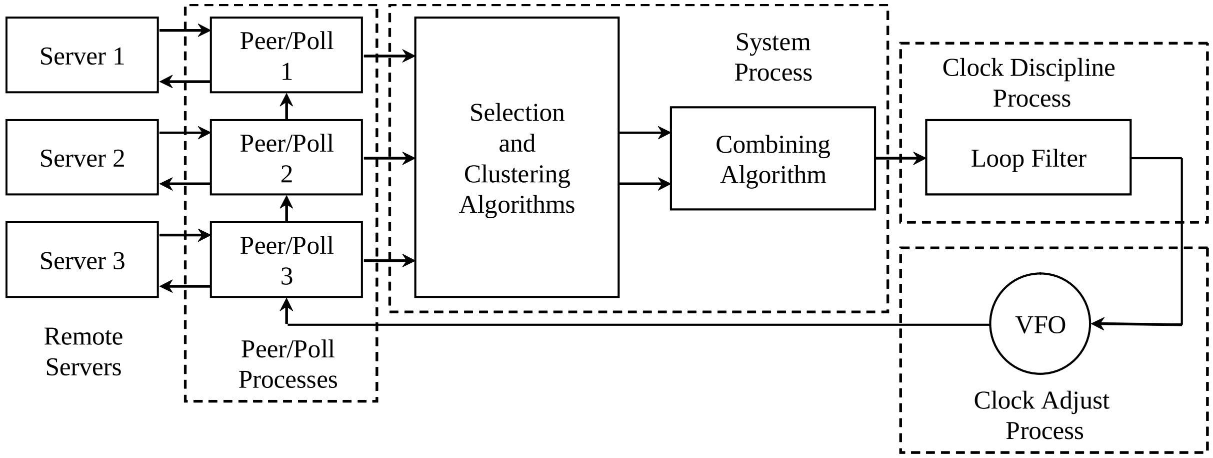


Figure 1. Implementation Model

several servers to more accurately discipline and calibrate the system clock. The *system offset* Θ represents the maximum-likelihood offset estimate for the server population. The *system jitter* ϑ represents the nominal error in estimating Θ. The δ and ε statistics are accumulated at each stratum level from the reference clocks to produce the *root delay* Δ and *root dispersion* Ε statistics. The *synchronization distance* $\Gamma = \mathrm{E} + \frac{\Delta}{2}$ represents the maximum error due all causes. The detailed formulations of these statistics are given later in this document. They are available to the dependent applications in order to assess the performance of the synchronization function.

4. Implementation Model

Figure 1 shows 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. These processes operate on a common data structure called an *association*, which contains the statistics described above along with various other data described later. A client sends an NTP packet to one or more servers and processes the replies as received. The server interchanges addresses and ports, overwrites certain fields in the packet and returns it immediately (client/ server mode) or at some time later (symmetric modes). As each NTP message is received, the offset θ between the peer clock and the system clock is computed along with the associated statistics δ, ε and φ.

The *system process* includes the selection, clustering and combining algorithms which 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 principles to cull the *falsetickers* from the incident population leaving the *truechimers* as result. The clustering algorithm uses statistical principles to sift the most accurate truechimers leaving the *survivors* as result. The combining algorithm develops the final clock offset as a statistical average of the survivors.

The *clock discipline process,* which is actually part of the system process, includes engineered algorithms to control 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 which 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 jitter* ϑ. The RMS average of past frequency offset differences represents the oscillator frequency stability or *frequency wander* Ψ.

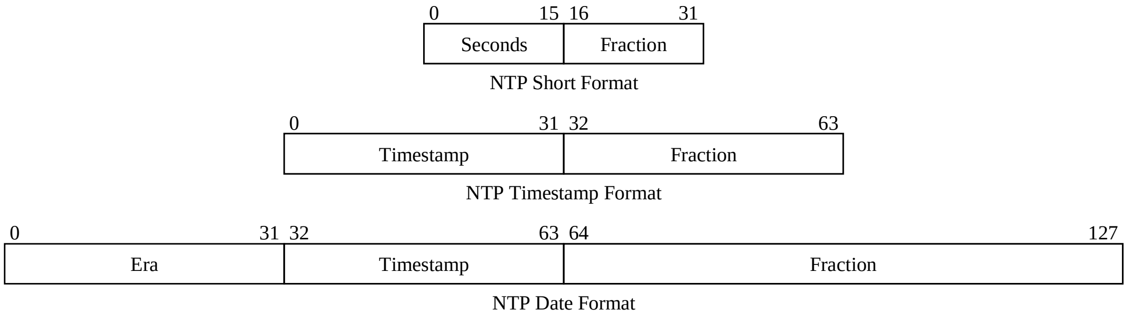
A client sends messages to each server with a *poll interval* of 2^{τ} seconds, as determined by the *poll exponent* τ. In NTPv4 τ ranges from 4 (16 s) through 17 (36 h). The value of τ is determined by the clock discipline algorithm to match the loop time constant $T_c = 2^{\tau}$. A server responds with messages at an *update interval* of μ seconds. For stateless servers, $\mu = T_c$, since the server responds immediately. However, in symmetric modes each of two peers manages the time constant as a function of current system offset and system jitter, so may not agree with the same τ. It is important that the dynamic behavior of the clock discipline algorithms be carefully controlled in order to maintain stability in the NTP subnet at large. This requires that the peers agree on a common τ equal to the minimum poll exponent of both peers. The NTP protocol includes provisions to properly negotiate this value.

While not shown in the figure, 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()[1] 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 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 is described later.

5. Data Types

All NTP time values are represented in twos-complement format, with bits numbered in big-endian 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 2. 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. For convenience in mapping between formats, the seconds field is divided into a 32-bit *era* field and a 32-bit *timestamp* 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.

[1] Empty parens following a name indicate reference to a function rather than a simple variable.



NTP Short Format

NTP Timestamp Format

NTP Date Format

Figure 2. 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 format the *prime epoch*, or base date of era 0, is 0 h 1 January 1900 UTC[1], when all bits are zero. Dates are relative to the prime epoch; values greater than zero represent times after that date; values less than zero represent times before it. 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

[1] 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.

Year	MJD	NTP Date	NTP Era	NTP Timestamp	Epoch
1 Jan -4712	–2,400,001	–208,657,814,400	–49	1,795,583,104	First day Julian Era
1 Jan -1	–679,306	–59,989,766,400	–14	139,775,744	2 BCE
1 Jan 0	–678,941	–59,958,230,400	–14	171,311,744	1 BCE
1 Jan 1	–678,575	–59,926,608,000	–14	202,934,144	1 CE
4 Oct 1582	–100,851	–10,011,254,400	–3	2,873,647,488	Last day of Julian Calendar
15 Oct 1582	–100,840	–10,010,304,000	–3	2,874,597,888	First day Gregorian Calendar
31 Dec 1899	15,019	–86,400	–1	4,294,880,896	Last day NTP Era –1
1 Jan 1900	15,020	0	0	0	First day NTP Era 0
1 Jan 1970	40,587	2,208,988,800	0	2,208,988,800	First day Unix
1 Jan 1972	41,317	2,272,060,800	0	2,272,060,800	First day UTC
31 Dec 1999	51,543	3,155,587,200	0	3,155,587,200	Last day 20th century
1 Jan 2000	51,544	3,155,673,600	0	3,155,673,600	First day 21st century
7 Feb 2036	64,730	4,294,944,000	0	4,294,944,000	Last day NTP Era 0
8 Feb 2036	64,731	4,295,030,400	1	63,104	First day NTP Era 1
16 Mar 2172	114,441	8,589,974,400	2	39,808	First day NTP Era 2
1 Jan 2500	234,166	18,934,214,400	4	1,754,345,216	2500 CE
1 Jan 3000	416,787	34,712,668,800	8	352,930,432	3000 CE

Table 2. Interesting Historic NTP Dates

unsynchronized time. Table 2 shows a number of historic NTP dates together with their corresponding Modified Julian Day (MJD), NTP era and NTP timestamp.

Let p be the number of significant bits in the second fraction. The *clock resolution* is defined 2^{-p}, in seconds. In order to minimize bias and help make timestamps unpredictable to an intruder, the nonsignificant 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 ρ, representing the precision used in this document, is the larger of the two.

The only operation permitted with 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 know the difference between signed and unsigned values; only the conditional branch instructions. 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 could happen in 2036, might result in incorrect values. 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. Further discussion on this issue is on the NTP project page linked from www.ntp.org.

Some time values are represented in exponent format, including the precision, time constant and poll interval values. These are in 8-bit signed integer format in log2 (log to the base 2) seconds.

The only 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 state 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. The era is the integer quotient and the timestamp the integer remainder as in

$$era = s/2^{32} \text{ and } timestamp = s - era \times 2^{32}, \qquad (2)$$

which works for positive and negative dates. To convert from NTP era and timestamp to system time requires the calculation

$$s = era \times 2^{32} + timestamp \qquad (3)$$

to determine the number of seconds since the prime epoch. Converting between NTP and system time can be a little messy, but 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.

6. Data Structures

The NTP protocol state machines described in the following sections are defined using state variables and flow chart fragments. 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. Additional constant and variable classes are defined in Appendix B.

6.1 Structure Conventions

In the text and diagrams to follow, state variables are rendered in fixed-width font, while equation variables are rendered in italic or Greek font. Ordinary text and named routines are rendered in native font. In order to distinguish between different variables of the same name but used in different processes, the following Unix-like structure member naming convention is adopted. Table 3 summarizes the naming conventions in this and subsequent figures and tables in

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

Table 3. Name Prefix Conventions

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

Table 4. Global Parameters

this document. 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.

Most flow chart fragments begin with a statement label and end with a named go-to or exit. A subroutine call includes a dummy () following the name and return at the end.to the point following the call.

6.2 Global Parameters

In addition to the variable classes a number of global parameters are defined in this document, including those shown with values in Table 4. While these are the only parameters needed in this document, a larger collection is necessary in the skeleton and larger still for the reference implementation. Section B.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, 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.

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.

Name	Formula	Description
`leap`	*leap*	leap indicator (LI)
`version`	*version*	version number (VN)
`mode`	*mode*	mode
`stratum`	*stratum*	stratum
`poll`	*poll*	poll exponent
`precision`	ρ_R	precision exponent
`rootdelay`	Δ_R	root delay
`rootdisp`	E_R	root dispersion
`refid`	*refid*	reference ID
`reftime`	*reftime*	reference timestamp
`org`	T_1	origin timestamp
`rec`	T_2	receive timestamp
`xmt`	T_3	transmit timestamp
`dst`	T_4	destination timestamp
`keyid`	*keyid*	key ID
`digest`	*digest*	message digest

Table 5. Packet Header Variables

6.3 Packet Header Variables

The most important state variables from an external point of view are the packet header variables described below. The NTP packet header follows the UDP and IP headers and the physical header specific to the underlying transport network. It consists of a number of 32-bit (4-octet) words, although some fields use multiple words and others are packed in smaller fields within a word. The NTP packet header shown in Appendix A has 12 words followed by optional extension fields and finally an optional message authentication code (MAC) consisting of the key identifier and message digest fields.

The optional extension fields described in Appendix A are used by the Autokey security protocol [3], which is not described here. The MAC is used by both Autokey and the symmetric key authentication scheme described in Appendix A. As is the convention in other Internet protocols, all fields are in network byte order, commonly called big-endian.

A list of the packet header variables is shown in Table 5 and described in detail below. The packet header fields apply to both transmitted (`x` prefix) and received packets (`r` prefix). The variables are interpreted as follows:

`leap` 2-bit integer warning of an impending leap second to be inserted or deleted in the last minute of the current month, coded as follows:

0	no warning
1	last minute of the day has 61 seconds
2	last minute of the day has 59 seconds
3	alarm condition (the clock has never been synchronized)

`version`. 3-bit integer representing the NTP version number, currently 4.

`mode` 3-bit integer representing the mode, with values defined as follows:

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

`stratum` 8-bit integer representing the stratum, with values defined as follows:

0 unspecified or invalid
1 primary server (e.g., equipped with a GPS receiver)
2-255 secondary server (via NTP)

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 algorithms used for external sources.

`poll` 8-bit signed integer representing the maximum interval between successive messages, in log2 seconds. In the reference implementation the limits are set by MINPOLL (4) and MAXPOLL (17), but the default limits 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 is normally determined when the service first starts up as the minimum time of several iterations to read the system clock.

`rootdelay` Total roundtrip delay to the reference clock, in NTP short format.

`rootdisp` Total dispersion to the reference clock, in NTP short format.

`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 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 radio clock. While not specifically enumerated in this document, the following have been used as ASCII identifiers:

GOES	Geosynchronous Orbit Environment Satellite
GPS	Global Position System
PPS	Generic pulse-per-second
IRIG	Inter-Range Instrumentation Group
WWVB	LF Radio WWVB Ft. Collins, CO 60 kHz
DCF77	LF Radio DCF77 Mainflingen, DE 77.5 kHz
HBG	LF Radio HBG Prangins, HB 75 kHz
MSF	LF Radio MSF Rugby, UK 60 kHz
JJY	LF Radio JJY Fukushima, JP 40 kHz, Saga, JP 60 kHz
LORC	MF Radio LORAN C 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 Kaui, HI
NIST	NIST telephone modem
USNO	USNO telephone modem
PTB	etc. European telephone modem

Above stratum 1 (secondary servers and clients) this is the reference identifier of the server. 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.

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

org Time at the client when the request departed for the server, in NTP timestamp format.

rec Time at the server when the request arrived from the client, in NTP timestamp format.

xmt Time at the server when the response left for the client, in NTP timestamp format.

dst Time at the client when the reply arrived from the server, in NTP timestamp format. Note: This value is not included in a header field; it is determined upon arrival of the packet and made avaiable in the packet buffer data structure.

keyid 32-bit unsigned integer used by the client and server to designate a secret 128-bit MD5 key. Together, the keyid and digest fields collectively are called message authentication code (MAC).

digest 128-bit bitstring computed by the keyed MD5 message digest algorithm described in Appendix A.

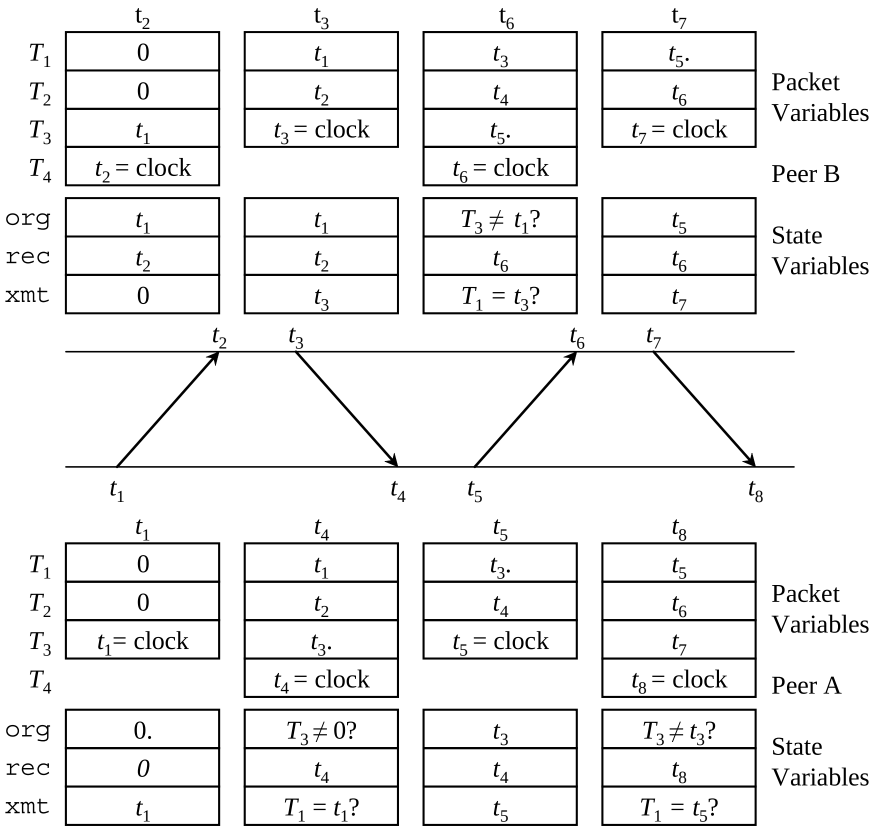


Figure 3. On-Wire Protocol

7. On-Wire Protocol

The NTP on-wire protocol is the core mechanism to exchange time values between servers, peers and clients. It is inherently resistant to lost or duplicate data 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, 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 case of link level retransmission and corrected for the time to compute a MAC on transmit[1].

The on-wire protocol uses four timestamps numbered T_1 through T_4 and three state variables `org`, `rec` and `xmt`, as shown in Figure 3. 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 individual timestamp values are shown in lower case with subscripts indicating the order of transmission and reception.

[1] The reference implementation strikes a timestamp after running the MD5 algorithm and adds the difference between it and the transmit timestamp to the next transmit timestamp.

In the figure the first packet transmitted by *A* containing only the transmit timestamp T_3 with value t_1. *B* receives the packet at t_2 and saves the origin timestamp T_1 with value t_1 in state variable `org` and the destination timestamp T_4 with value t_2 in state variable `rec`. At this time or some time later *B* sends a packet to *A* containing the `org` and `rec` state variables in T_1 and T_2, respectively and in addition the transmit timestamp T_3 with value t_3, which is saved in the `xmt` state variable. When this packet arrives at *A* the packet header variables T_1, T_2, T_3 and destination timestamp T_4 represent the four timestamps necessary to compute the offset and delay of *B* relative to *A*, as described later.

Before the *A* state variables are updated, two sanity checks are performed in order to protect against duplicate or bogus packets. A packet is a duplicate if the transmit timestamp T_3 in the packet matches the `xmt` state variable. A packet is bogus if the origin timestamp T_1 in the packet does not match the `org` state variable. In either of these cases the state variables are updated, but the packet is discarded.

The four most recent timestamps, T_1 through T_4, are used to compute the offset of *B* relative to *A*

$$\theta = T(B) - T(A) = \frac{1}{2}[(T_2 - T_1) + (T_3 - T_4)] \tag{4}$$

and the roundtrip delay

$$\delta = T(ABA) = (T_4 - T_1) - (T_3 - T_2) . \tag{5}$$

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 the sum and difference of these values, which contain 62 significant bits and two sign bits, so 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 timestamps themselves, there is no loss in significance, yet the unambiguous range is increased from 34 years to 68 years. Additional considerations on these issues, as well as the behavior when moving beyond the prime era, are discussed in online white papers at www.ntp.org but beyond the scope of this document.

In some scenarios where the frequency offset between the client and server is relatively large and the actual propagation time small, it is possible that the delay computation becomes negative. For instance, if the frequency difference is 100 PPM and the interval $T_4 - T_1$ is 64 s, the apparent delay is −6.4 ms. Since negative values are misleading in subsequent computations, the value of δ should be clamped not less than the system precision `s.precision` ρ defined below.

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 T_3 and T_4 from the client packet to T_1 and T_2 of the server packet and tacks on the transmit timestamp T_3 before sending it to the client. Additional details for filling in the remaining protocol fields are given in the next section and in Appendix B.

A SNTP primary server implementing the on-wire protocol has no upstream servers except a single reference clock. In principle, it is indistinguishable from an NTP primary server which has the mitigation algorithms, presumably to mitigate between multiple reference clocks. Upon receiving a client request, a SNTP primary server constructs and sends the reply packet as shown in Figure 5 below. Note that the dispersion field in the packet header must be calculated in the same way as in the NTP case.

A SNTP client using the on-wire protocol has a single server and no downstream clients. It can operate with any subset of the NTP on-wire protocol, the simplest 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 and is encouraged.

8. Peer Process

The peer process is called upon arrival of a server packet. It runs the on-wire protocol to determine the clock offset and roundtrip delay and in addition 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 for each server.

The discussion in this section covers only the variables and routines necessary for a conforming NTPv4 implementation. Additional implementation details are in Section B.5. The engineering principles and measured performance with the reference implementation are discussed in [2].

8.1 Peer Process Variables

Table 6 summarizes the common names, formula names and a short description of each peer variable, all of which have prefix `p`. The following configuration variables are normally initialized when the association is mobilized, either from a configuration file or upon arrival of the first packet for an ephemeral association.

`p.srcadr` IP address of the remote server or reference clock. In the reference implementation reference clock addresses are by convention in IPv4 format with prefix 127.127.`t`.`u`, where `t` is the device driver number and `u` the instantiation number. This becomes the destination IP address in packets sent from this association.

Name	Formula	Description
Configuration Variables		
srcaddr	*srcaddr*	source address
srcport	*srcport*	source port
dstaddr	*dstaddr*	destination address
dstport	*destport*	destination port
keyid	*keyid*	key identifier key ID
Packet Variables		
leap	*leap*	leap indicator
version	*version*	version number
mode	*mode*	mode
stratum	*stratum*	stratum
ppoll	*ppoll*	peer poll exponent
rootdelay	Δ_R	root delay
rootdisp	E_R	root dispersion
refid	*refid*	reference ID
reftime	*reftime*	reference timestamp
Timestamp Variables		
t	t	epoch
org	T_1	origin timestamp
rec	T_2	receive timestamp
xmt	T_3	transmit timestamp
Statistics Variables		
offset	θ	clock offset
delay	δ	roundtrip delay
disp	ε	dispersion
jitter	φ	jitter

Table 6. Peer Process Variables

p.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.

p.dstadr IP address of the client. This becomes the source IP address in packets sent from this association.

p.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.

p.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 below are updated from the packet header as each packet arrives. They are interpreted in the same way as the as the packet variables of the same names.

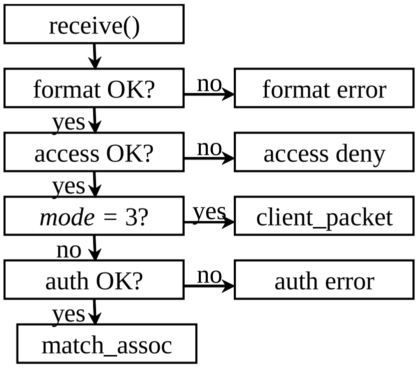


Figure 4. Receive Processing

> `p.leap`, `p.version`, `p.mode`, `p.stratum`, `p.ppoll`, `p.rootdelay`, `p.rootdisp`, `p.refid`, `p.reftime`

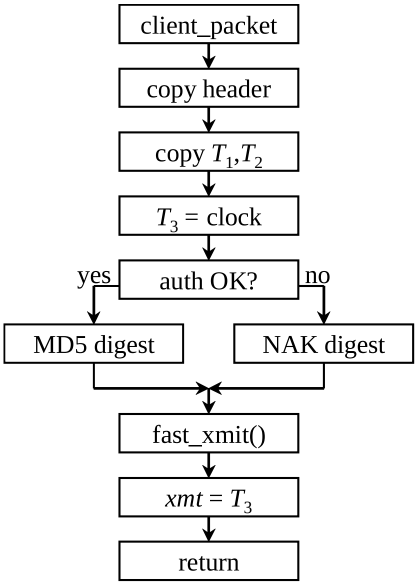
It is convenient for later processing to convert the NTP short format packet values `p.rootdelay` and `p.rootdisp` to floating doubles as peer variables.

The `p.org`, `p.rec`, `p.xmt` variables represent the timestamps computed by the on-wire protocol described previously. The `p.offset`, `p.delay`, `p.disp`, `p.jitter` variables represent the current time values and statistics produced by the clock filter algorithm. The offset and delay are computed by the on-wire protocol; the dispersion and jitter are calculated as described below. Strictly speaking, the epoch `p.t` is not a timestamp; it records the system timer upon arrival of the latest packet selected by the clock filter algorithm.

8.2 Peer Process Operations

Figure 4 shows the peer process code flow upon the arrival of a packet. Additional details specific to the reference implementation are shown in the receive() and access() routines in Section B.5. There is no specific method required for access control, although it is recommended that implementations include a match-and-mask scheme similar to many others now in widespread use, as well as in the reference implementation. 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, the symmetric key scheme described in Appendix A must be included among the supported. This scheme uses the MD5 keyed hash algorithm Section B.2. For the most vulnerable applications the Autokey public key scheme described in [3] and supported by the reference implementation in is recommended.

Next, the association table is searched for matching source address and source port using the find_assoc() routine in Section B.5. The dispatch table near the beginning of that section is indexed by the packet mode and association mode (0 if no matching association) to determine the dispatch code and thus the case target. The significant cases are FXMT, NEWPS and NEWBC.



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 5. Client Packet Processing

FXMIT. This is a client (mode 3) packet matching no association. The server constructs a server (mode 4) packet and returns it to the client without retaining state. The server packet is constructed as in Figure 5 and the fast_xmit() routine in Section B.5. If the s.rootdelay and s.rootdisp system variables are stored in floating double, they must be converted to NTP short format first. 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 the figure, but with a MAC consisting of four octets of zeros. The client is free to accept or reject the data in the message.

NEWBC. This is a broadcast (mode 5) packet matching no association. The client mobilizes a client (mode 3) association as shown in the mobilize() and clear() routines in Section B.2. The reference implementation first performs the necessary steps to run the Autokey protocol and determine the propagation delay, then continues in listen-only (mode 6) to receive further packets. 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 is a symmetric active (1) packet matching no association. The client mobilizes a symmetric passive (mode 2) association as shown in the mobilize() and clear() routines in Section B.2. Code flow continues to the match_assoc fragment described below.

In other cases the packet matches an existing association and code flows to the match_assoc fragment in Figure 6. The packet timestamps are carefully checked to avoid invalid, duplicate or bogus packets, as shown in the figure. Note that a crypto-NAK is considered valid only if it survives these tests. Next, the peer variables are copied from the packet header variables as shown in Figure 7 and the packet() routine in Section B.5. The reference implementation

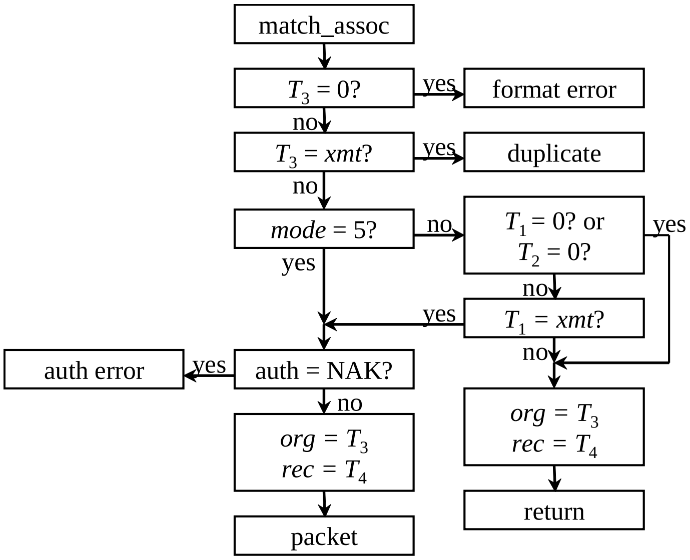
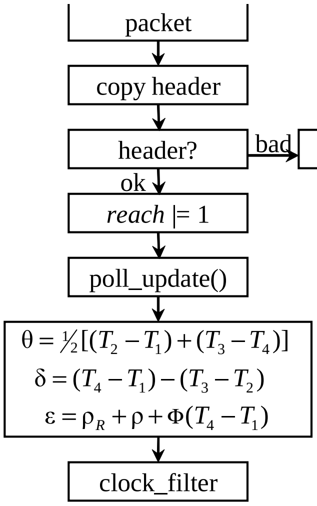


Figure 6. Timestamp Processing


packet
copy header
header?
bad
header error
ok
reach |= 1
poll_update()
$\theta = \frac{1}{2}[(T_2 - T_1) + (T_3 - T_4)]$
$\delta = (T_4 - T_1) - (T_3 - T_2)$
$\varepsilon = \rho_R + \rho + \Phi(T_4 - T_1)$
clock_filter

Peer Variables		Packet Variables
`p.leap`	←	`r.leap`
`p.mode`	←	`r.mode`
`p.stratum`	←	`r.stratum`
`p.ppoll`	←	`r.ppoll`
`p.rootdelay`	←	`r.rootdelay`
`p.rootdisp`	←	`r.rootdisp`
`p.refid`	←	`r.refid`
`p.reftime`	←	`r.reftime`

Figure 7. Packet Processing

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	The packet is not a reply to the most recent packet sent. This can happen in symmetric modes if the poll intervals are uneven.
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 address.
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.
8 autokey error	Public key cryptography has failed to authenticate the packet.
9 crypto error	Mismatched or missing cryptographic keys or certificates.

Table 7. Packet Error Checks

includes a number of data range checks shown in Table 7 and discards the packet if the ranges are exceeded; however, the header fields are copied even if errors occur, since they are necessary in symmetric modes to construct the subsequent poll message.

The 8-bit `p.reach` shift register in the poll process described later is used to determine whether the server is reachable or not and provide information useful to insure 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 poll_update() routine in Section B.8 is called to redetermine the host poll interval.

The on-wire protocol calculates the clock offset θ and roundtrip delay δ from the four most recent timestamps as shown in Figure 3. 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. The dispersion statistic ε(t) represents the maximum error due to the frequency tolerance and time since the last measurement. It is initialized

$$\varepsilon(t_0) = \rho_R + \rho + \Phi(T_4 - T_1) \tag{6}$$

when the measurement is made at t_0. Here ρ_R is the peer precision in the packet header `r.precision` and ρ the system precision `s.precision`, 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 TOLERANCE (Φ); in other words, at time t, $\varepsilon(t) = \varepsilon(t_0) + \Phi(t - t_0)$. With the default value Φ = 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 ε. The remaining statistics are computed by the clock filter algorithm described in the next section.

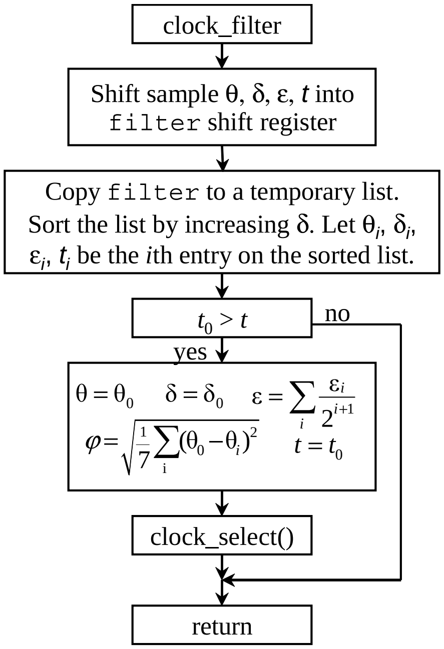

clock_filter
Shift sample θ, δ, ε, t into filter shift register
Copy filter to a temporary list. Sort the list by increasing δ. Let θ_i, δ_i, ε_i, t_i be the ith entry on the sorted list.
$t_0 > t$
no
yes
$\theta = \theta_0$ $\delta = \delta_0$ $\varepsilon = \sum_i \frac{\varepsilon_i}{2^{i+1}}$
$\varphi = \sqrt{\frac{1}{7}\sum_i (\theta_0 - \theta_i)^2}$ $t = t_0$
clock_select()
return

Figure 8. Clock Filter Processing

8.3 Clock Filter Algorithm

The clock filter algorithm grooms the stream of on-wire data to select the samples most likely to represent the correct time. The algorithm produces the `p.offset` θ, `p.delay` δ, `p.dispersion` ε, `p.jitter` φ, and time of arrival `p.t` t 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 core processing steps of this algorithm are shown in Figure 8 with more detail in the clock_filter() routine in Section B.5.

The clock filter algorithm saves the most recent sample tuples (θ, δ, ε, t) in an 8-stage shift register in the order that packets arrive. Here t is the system timer, not the peer variable of the same name. The following scheme is used to insure sufficient samples are in the register and that old stale data are discarded. Initially, the tuples of all stages are set to the dummy tuple (0, MAXDISP, MAXDISP, t). As valid packets arrive, the (θ, δ, ε, t) tuples are shifted into the register causing old samples to be discarded, so eventually only valid samples 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 the 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 δ. Let j index the stages starting with the lowest δ. If the sample epoch t_0 is not later than the last valid sample epoch `p.t`, the routine exits without affecting the current peer variables. Otherwise, let ε_j be the dispersion of the jth entry, then

$$\varepsilon = \sum_{j=0}^{7} \frac{\varepsilon_j}{2^{j+1}} \tag{7}$$

is the peer dispersion `p.disp`. Note the overload of ε, 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, (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 the peer variables are acceptable or not.

Let the first stage offset in the sorted list be θ_0; then, for the other stages in any order, the jitter is the RMS average

$$\varphi = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n-1}(\theta_0 - \theta_j)^2}, \tag{8}$$

where *n* is the number of valid tuples in the register. In order to insure consistency and avoid divide exceptions in other computations, the φ is bounded from below by the system precision ρ 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 root delay and root dispersion. The root delay is

$$\delta' = \Delta_R + \delta \tag{9}$$

and the root dispersion is

$$\varepsilon' = E_R + \varepsilon + \varphi . \tag{10}$$

Note that ε and therefore ε' increase at rate Φ. The peer synchronization distance is defined

$$\lambda = \frac{\delta'}{2} + \varepsilon' \tag{11}$$

and recalculated as necessary. The λ is a component of the root synchronization distance Λ used by the mitigation algorithms as a metric to evaluate the quality of time available from each server. Note that there is no state variable for λ, as it depends on the time since the last update.

As a practical matter, a not uncommon hazard in global Internet timekeeping is an occasional isolated offset surge, called a *popcorn spike*, due to some transient delay phenomenon in the network. The reference implementation uses a popcorn spike suppressor to reduce errors due this cause. It operates by tracking the exponentially averaged jitter and discarding an offset spike that exceeds a threshold equal to some multiple of the average. The spike itself is then used to update the average, so the threshold is self-adaptive.

9. System Process

As each new sample (θ, δ, ε, t) is produced by the clock filter algorithm, the sample is processed by the mitigation algorithms consisting of the selection, clustering, combining 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 clustering algorithm discards the association statistically furthest from the centroid until a minimum number of survivors remain. The combining algorithm produces the best and final offset on a weighted average basis and selects one of the associations as the *system peer* providing the best statistics for performance evaluation. The final offset is passed to the clock discipline algorithm to steer the system clock to the correct time. The statistics (θ, δ, ε, t) associated with the system peer are used to construct the system variables inherited by dependent servers and clients and made available to other applications running on the same machine.

The discussion in following sections covers only the basic variables and routines necessary for a conforming NTPv4 implementation. Additional implementation details are in Section B.6. An interface that might be considered in a formal specification is represented by the function prototypes in Section B.1. The engineering principles and measured performance with the reference implementation are discussed in [2].

9.1 System Process Variables

The variables and parameters associated with the system process are summarized in Table 8, which gives the variable name, formula name and short description. Unless noted otherwise, all variables have assumed prefix `s`. All the variables except `s.t` and `s.p` have the same format and interpretation as the peer variables of the same name. The remaining variables are defined below.

`s.t`	Integer representing the value of the system timer at the last update.
`s.p`	System peer association pointer.

Name	Formula	Description
t	*t*	epoch
leap	*leap*	leap indicator
stratum	*stratum*	stratum
precision	ρ	precision
p	p	system peer pointer
offset	Θ	combined offset
jitter	ϑ	combined jitter
rootdelay	Δ	root delay
rootdisp	E	root dispersion
refid	*refid*	reference ID
reftime	*reftime*	reference time
NMIN	3	minimum survivors
CMIN	1	minimum candidates

Table 8. System Process Variables and Parameters

s.precision 8-bit signed integer representing the precision of the system clock, in log2 seconds.ZA

s.offset Offset computed by the combining algorithm.

s.jitter Jitter computed by the cluster and combining algorithms.

The variables defined below are updated from the system peer process as described later. They are interpreted in the same way as the as the peer variables of the same names.

s.leap, s.stratum, s.rootdelay, s.rootdisp, s.refid, s.reftime

Initially, all variables are cleared to zero, then the s.leap is set to 3 (unsynchronized) and s.stratum is set to MAXSTRAT (16). The remaining statistics are determined as described below.

9.2 System Process Operations

The system process implements the selection, clustering, combining and clock discipline algorithms shown in Figure 1. The clock_select() routine in Figure 9 includes the selection algorithm of Section 9.2.1 that produces a majority clique of truechimers based on agreement principles. The clustering algorithm of Section 9.2.2 discards the outliers of the clique to produce the survivors used by the combining algorithm in Section 9.2.3, which in turn provides the final offset for the clock discipline algorithm in Section 9.2.4.

If the selection algorithm cannot produce a majority clique, or if the clustering algorithm cannot produce at least CMIN survivors, the system process terminates with no further processing. If successful, the clustering algorithm selects the statistically best candidate as the system peer and its variables are inherited as the system variables.

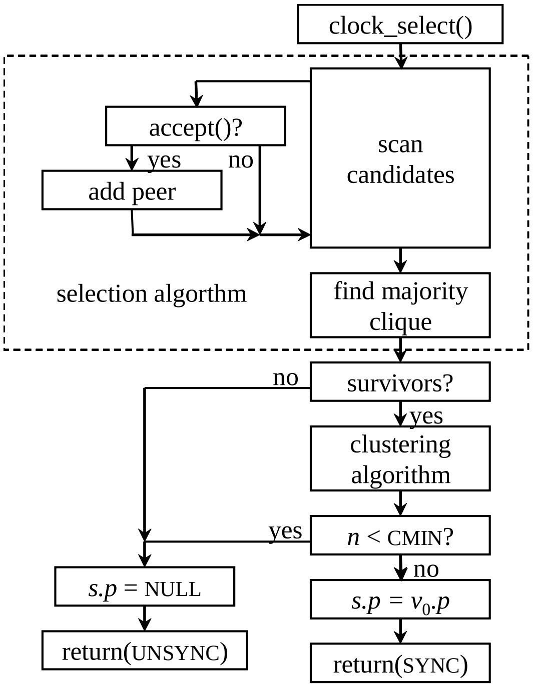


Figure 9. clock_select() Routine

The selection and clustering algorithms are described below separately, but combined in the code skeleton and reference implementation.

9.2.1 Selection Algorithm

The selection algorithm operates to find the truechimers using Byzantine agreement principles originally proposed by Marzullo [1], but modified to improve accuracy. An overview of the algorithm is in Figure 10 and the first half of the clock_select() routine in Section B.6.1. First, those servers which are unusable according to the rules of the protocol are detected and discarded by the accept() routine in Figure 11 and Section B.6.3. Next, a set of tuples {*p*, *type*, *edge*} is generated for the remaining servers, where *p* is an association pointer, *type* and *edge* identifies the upper (+1), middle (0) and lower (–1) endpoint of a *correctness interval* $[\theta - \lambda, \theta + \lambda]$, where λ is the root distance calculated in (11).

The tuples are placed on a list and sorted by *edge*. The list is processed from the lowest to the highest, then from highest to lowest using the algorithm in Figure 10 and in Section B.6.1 and described in detail in [2]. The algorithm starts with the assumption that there are no falsetickers ($f = 0$) and attempts to find a nonempty *intersection interval* containing the midpoints of all correct servers, i.e., truechimers. If a nonempty interval cannot be found, it increases the number of assumed falsetickers by one and tries again. If a nonempty interval is found and the number of falsetickers is less than the number of truechimers, a majority clique has been found and the midpoints (offsets) represent the survivors available for the clustering algorithm. Otherwise, there are no suitable candidates to synchronize the system clock.

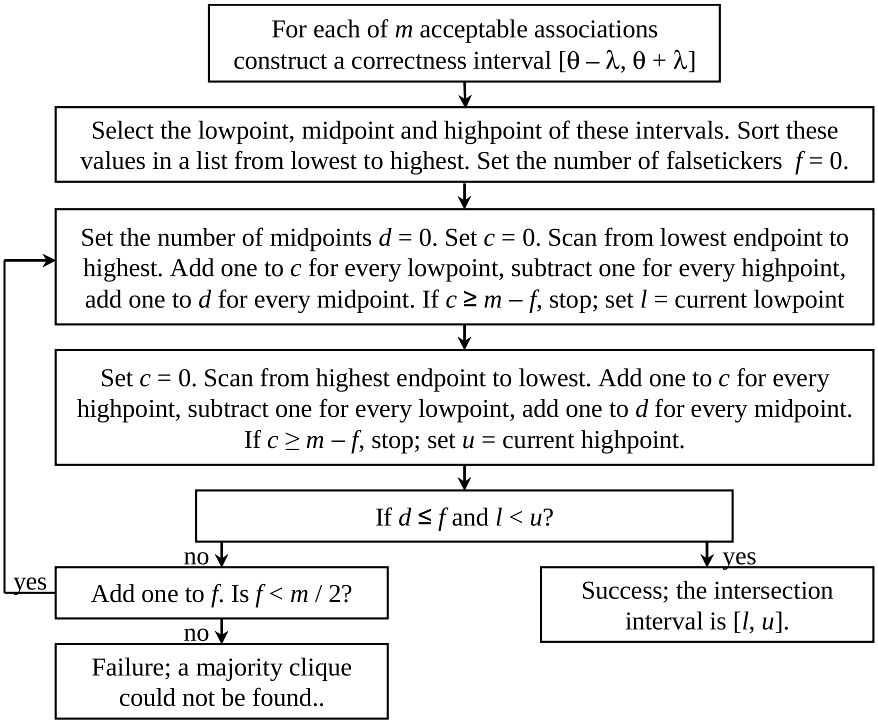

For each of m acceptable associations construct a correctness interval [θ – λ, θ + λ]
Select the lowpoint, midpoint and highpoint of these intervals. Sort these values in a list from lowest to highest. Set the number of falsetickers f = 0.
Set the number of midpoints d = 0. Set c = 0. Scan from lowest 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
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.
If d ≤ f and l < u?
no
yes
yes
Add one to f. Is f < m / 2?
Success; the intersection interval is [l, u].
no
Failure; a majority clique could not be found..

Figure 10. Selection Algorithm

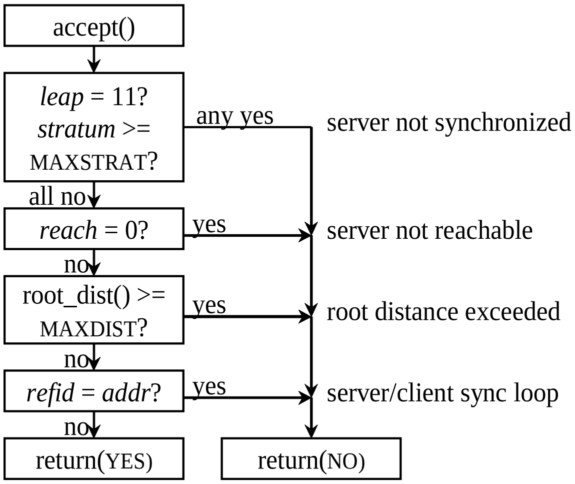


Figure 11. accept() Routine

While not shown on the flow chart, λ is increased by MINDISP (.005 s) when constructing the tuples. The reason for this is to avoid problems with very fast processors and networks. Occasionally, due to random fluctuations and jitter, two legitimate correctness intervals fail to overlap and may cause both to be declared falseticker. The MINDISP increment acts like a shim to decrease the likelihood this might occur. In the reference implementation MINDISP is a configurable value that can be changed to fit each scenario.

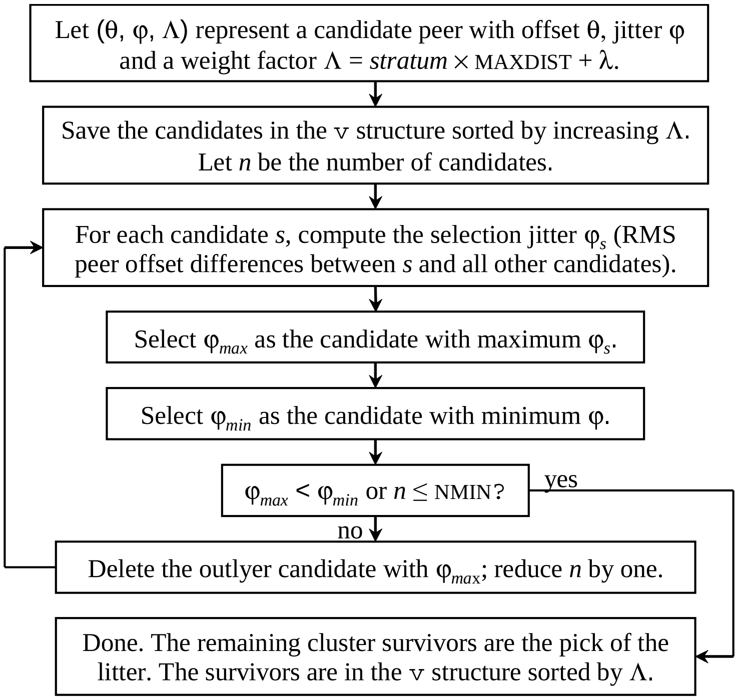

Let (θ, φ, Λ) represent a candidate peer with offset θ, jitter φ and a weight factor Λ = stratum × MAXDIST + λ.
Save the candidates in the v structure sorted by increasing Λ. Let n be the number of candidates.
For each candidate s, compute the selection jitter φs (RMS peer offset differences between s and all other candidates).
Select φmax as the candidate with maximum φs.
Select φmin as the candidate with minimum φ.
φmax < φmin or n ≤ NMIN?
yes
no
Delete the outlyer candidate with φmax; reduce n by one.
Done. The remaining cluster survivors are the pick of the litter. The survivors are in the v structure sorted by Λ.

Figure 12. Clustering Algorithm

9.2.2 Clustering Algorithm

The members of the majority clique are placed on the *survivor list*, and sorted first by stratum, then by root distance λ. The sorted list is processed by the clustering algorithm in Figure 12 and the second half of the clock_select() algorithm in Section B.6.1. It operates in a series of rounds where each round discards the furthest statistical outlier until a specified minimum number of survivors NMIN (3) are left or until no further improvement is possible.

In each round let *n* be the number of survivors and *s* index the survivor list. Assume φ_p is the peer jitter of the *s* survivor. Compute

$$\varphi_s = \sqrt{\frac{1}{n-1}\sum_{j=0}^{n-1}(\theta_s - \theta_j)^2} \qquad (12)$$

as the *selection jitter*. Then choose $\varphi_{max} = \max(\varphi_s)$ and $\varphi_{min} = \min(\varphi_p)$. If $\varphi_{max} < \varphi_{min}$ or $n < \mathrm{NMIN}$, no further reduction in selection jitter is possible, so the algorithm terminates and the remaining survivors are processed by the combining algorithm. Otherwise, the algorithm casts off the φ_{max} survivor, reduces *n* by one and makes another round.

Variable	Process	Description
Θ	system	combined clock offset
ϑ_p	system	combined jitter
θ_0	survivor list	first survivor offset
θ_i	survivor list	*i*th survivor offset
x, y, z, w		temporaries

Variables and Parameters

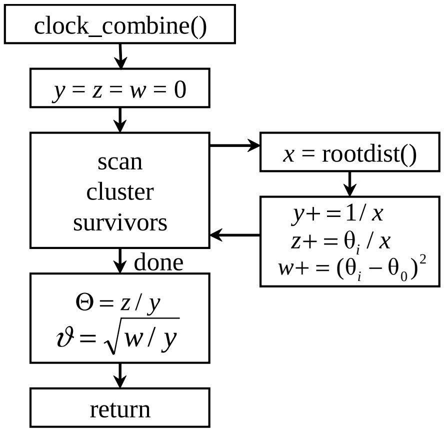
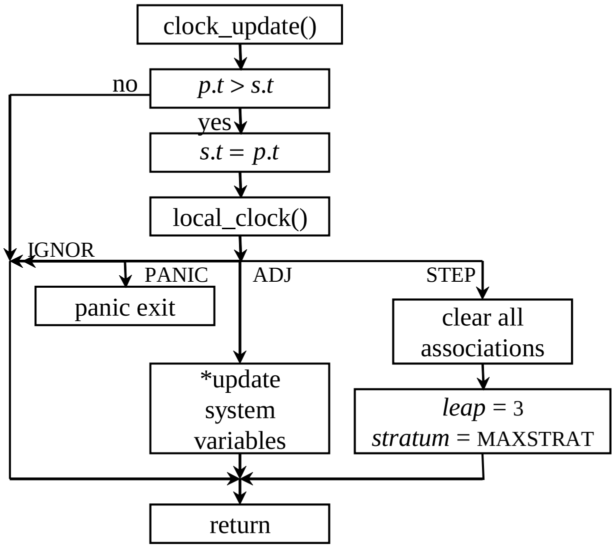


Figure 13. clock_combine() Routine



System Variables	System Peer Variables
`leap`	← `leap`
`stratum`	← `stratum + 1`
`refid`	← `refid`
`reftime`	← `reftime`
Δ	$\leftarrow \Delta_R + \delta$
E	$\leftarrow E_R + \varepsilon + \Phi\mu + \varphi + \lvert\Theta\rvert$

*Update System Variables

Figure 14. clock_update() Routine

9.2.3 Combining Algorithm

The remaining survivors are processed by the clock_combine() routine in Figure 13 and Section B.6.4 to produce the best and final data for the clock discipline algorithm. The routine processes the peer offset θ and jitter φ to produce the system offset Θ and system peer jitter ϑ_p, where each server statistic is weighted by the reciprocal of the root distance. and the result normalized. The system peer jitter ϑ_p is a component of the system jitter described later.

The system statistics are passed to the clock_update() routine in Figure 14 and Section B.6.4. If there is only one survivor, the offset passed to the clock discipline algorithm is $\Theta = \theta$ and the system peer jitter is $\vartheta = \varphi$. Otherwise, the selection jitter ϑ_S is computed as in (8), where θ_0 represents the offset of the system peer and *j* ranges over the survivors.

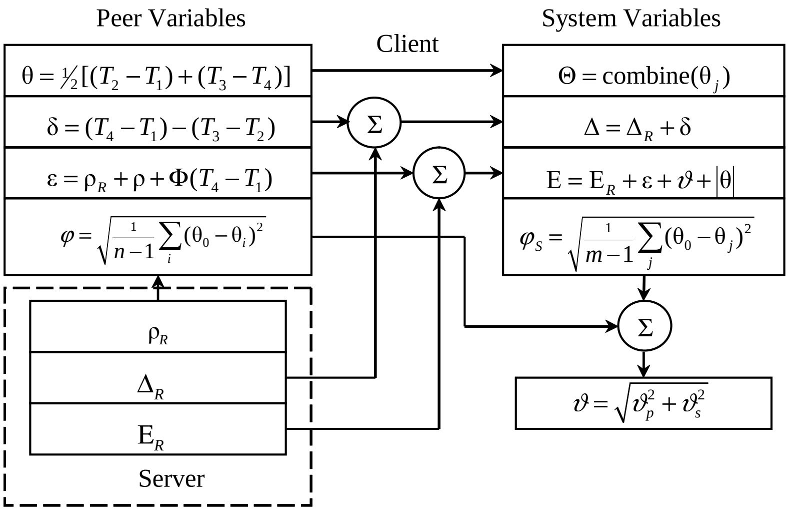

Peer Variables
Client
System Variables
$\theta = \frac{1}{2}[(T_2 - T_1) + (T_3 - T_4)]$
$\delta = (T_4 - T_1) - (T_3 - T_2)$
$\varepsilon = \rho_R + \rho + \Phi(T_4 - T_1)$
$\varphi = \sqrt{\frac{1}{n-1}\sum_i(\theta_0 - \theta_i)^2}$
Σ
Σ
$\Theta = \text{combine}(\theta_j)$
$\Delta = \Delta_R + \delta$
$E = E_R + \varepsilon + \vartheta + |\theta|$
$\varphi_S = \sqrt{\frac{1}{m-1}\sum_j(\theta_0 - \theta_j)^2}$
Σ
$\vartheta = \sqrt{\vartheta_p^2 + \vartheta_s^2}$
ρ_R
Δ_R
E_R
Server

Figure 15. System Variables Processing

The first survivor on the survivor list is selected as the system peer, here represented by the statistics (θ, δ, ε, φ). By rule, an update is discarded if its time of arrival `p.t` is not strictly later than the last update used `s.t`. Let μ = `p.t` – `s.t` be the time since the last update or *update interval*. If the update interval is less than or equal to zero, the update is discarded. Otherwise, the system variables are updated from the system peer variables as shown in Figure 14. Note that `s.stratum` is set to `p.stratum` plus one.

The arrows labelled 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 normally causes 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). Since this means all peer data have been invalidated, all associations are reset and the client begins as at initial start. ADJ means the offset is less than the step threshold and thus a valid update for the local_clock() routine described later. In this case the system variables are updated as shown in Figure 14.

There is one exception not shown. The dispersion increment is bounded from below by MINDISP. In subnets with very fast processors and networks and very small dispersion and delay this forces a monotone-definite increase in E, which avoids loops between peers operating at the same stratum.

Figure 15 shows how the error budget grows from the packet variables, on-wire protocol and system peer process to produce the system variables that are passed to dependent applications and clients. The system jitter is defined

$$\vartheta = \sqrt{\vartheta_p^2 + \vartheta_s^2}, \qquad (13)$$

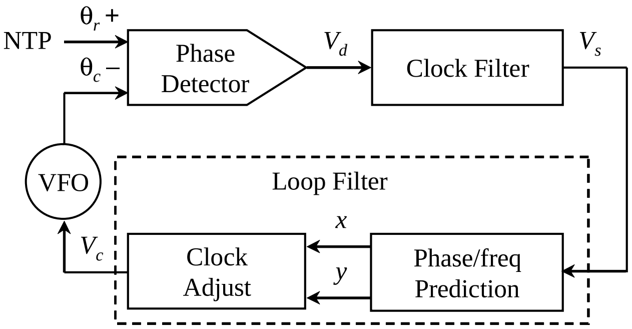


Figure 16. Clock Discipline Feedback Loop

where ϑ_s is the selection jitter relative to the system peer computed as in (12). The system jitter is passed to dependent applications programs as the *nominal error* statistic. The root delay Δ and root dispersion E statistics are relative to the primary server reference clock and thus inherited by each server along the path. The system synchronization distance is defined

$$\Lambda = \frac{\Delta}{2} + \mathrm{E}, \qquad (14)$$

which is passed to dependent application programs as the *maximum error* statistic.

9.2.4 Clock Discipline Algorithm

The NTPv4 clock discipline algorithm, shortened to *discipline* in the following, functions as a combination of two philosophically quite different feedback control systems. In a phase-locked loop (PLL) design, periodic phase updates at update intervals μ 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 μ are used directly to minimize the frequency error and indirectly the time error. As shown in [2], a PLL usually works better when network jitter dominates, while a 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 [2], which also includes a performance analysis.

Recall from Figure 1 how the clock discipline and clock adjust processes interact with the other algorithms in NTPv4. The output of the combining algorithm represents the best estimate of the system clock offset relative to the server ensemble. The discipline adjusts the frequency of the VFO to minimize this offset. Finally, the timestamps of each server are compared to the timestamps derived from the VFO in order to calculate the server offsets and close the feedback loop.

The discipline is implemented as the feedback control system shown in Figure 16. The variable θ_r represents the combining algorithm offset (reference phase) and θ_c the VFO offset (control phase). Each update produces a signal V_d representing the instantaneous phase difference

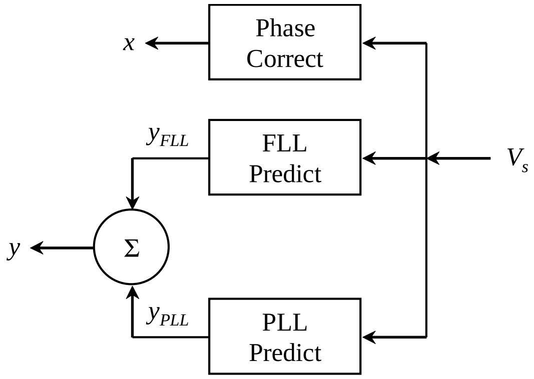


Figure 17. Clock Discipline Loop Filter

$\theta_r - \theta_c$. 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, clustering and combining 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 ω_C and thus its phase $\theta_c = \int \omega_c dt$, which closes the loop. The V_C signal is generated by the clock adjust process in Section 9.3. It runs at intervals of one second in the NTP daemon or one tick in the kernel implementation. The characteristic behavior of this model, which is determined by *F*(*t*) and the various gain factors given in Section B.6.6.

The transient behavior of the PLL/FLL feedback loop is determined by the impulse response of the loop filter *F*(*t*). The loop filter shown in Figure 17 predicts a phase adjustment *x* as a function of V_S. The PLL predicts a frequency adjustment y_{FLL} as an integral $\int V_s \mu dt$, while the FLL predicts an adjustment y_{PLL} as a function of V_s / μ. The two adjustments are combined to correct the frequency *y* as shown in Figure 16. The *x* and *y* are then used by the clock_adjust() routine to control the VFO frequency. The detailed equations that implement these functions are best presented in the routines of Sections B.6.6 and B.7.1.

Ordinarily, the pseudo-linear feedback loop described above operates to discipline the system clock. However, there are cases where a nonlinear 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 nonlinear 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.

The remainder of this section describes how the discipline works. Table 9 contains a summary of the variables and parameters including the program name, formula name and short description. Unless noted otherwise, all variables have assumed prefix `c`. The variables `c.t`, `c.tc`, `c.state`, and `c.count` are integers; the remainder are floating doubles. The function of each will be explained in the algorithm descriptions below.

Name	Formula	Description
t	*timer*	seconds counter
offset	Θ	combined offset
resid	Θ_R	residual offset
freq	ϕ	clock frequency
jitter	φ	clock jitter
wander	Ψ	frequency wander
tc	τ	time constant (log2)
state	*state*	state
adj	*adj*	frequency adjustment
count	*count*	hysteresis counter
STEPT	125	step thresh (.125 s)
WATCH	900	stepout thresh (s)
PANICT	1000	panic thresh. (1000 s)
LIMIT	30	hysteresis limit
PGATE	4	hysteresis gate
TC	16	time constant scale
AVG	8	averaging constant

Table 9. Clock Discipline Variables and Parameters

State	$\Theta <$ STEP	$\Theta \geq$ STEP	Comments
NSET	→FREQ, adj. time	→FREQ, step time	Frequency never set.
FSET	→SYNC, adj. time	→SYNC, step time	Frequency set from file.
SPIK	→SYNC, adj. freq, adj. time	if $\mu <$ WATCH, →SPIK else step time	Outlyer found.
FREQ	if $\mu <$ WATCH, →FREQ else step freq, adj. time	if $\mu <$ WATCH, →FREQ else →SYNC, step freq, step time	Frequency measurement.
SYNC	→SYNC, adj freq, adj. time	→SPIK	Normal operation.

Table 10. Clock Discipline State Transition Table

The discipline is implemented by the local_clock() routine, which is called from the clock_update() routine in Figure 14. The local_clock() routine pseudo code in Section B.6.6 has two parts; first the state machine shown in Figure 18 and second the algorithm that determines the time constant and thus the poll interval in Figure 19. The state transition function in Figure 18 is implemented by the rst() function shown at the lower left of the figure. The local_clock() routine exits immediately if the offset is greater than the panic threshold. The reference implementation sends a message to the operator and system log to set the clock manually within that range. A command line option can be used to override this behavior, but only for the first occurrence. Subsequent behavior depends on the offset magnitude and a state machine that determines if the offset and frequency are adjusted in steps or gradual increments.

The state machine transition function is shown in Table 10. The first column is the state name; the second column shows the next state and actions if the offset is less than the step threshold; the third column shows the next state and actions if the offset is greater than the step threshold; the fourth shows a brief comment. The next state is indicated by an arrow → followed by the

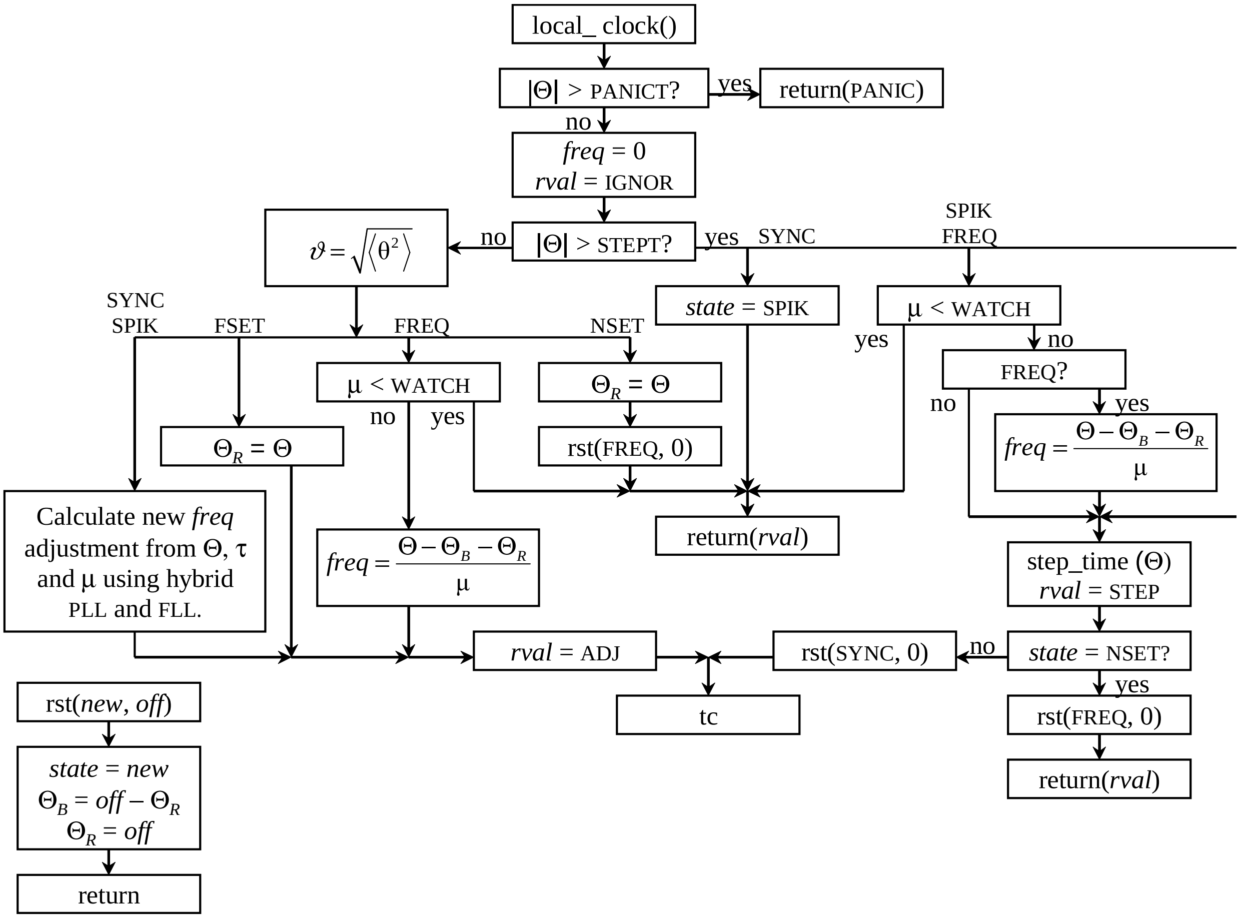


Figure 18. local_clock() Routine (1 of 2)

next state name. Some actions are conditional on whether the update interval μ exceeds the *stepout threshold* WATCH (900 s).

In order to speed convergence when the program is restarted, the reference implementation records the frequency offset in a file once each hour. When the program is started and the file has not been created, the machine starts in the NSET state, where it initializes the time and transitions to the FREQ state. The machine stays in that state until the first update after the stepout interval, when it computes the frequency directly and transitions to the SYNC state. When started after the frequency file has been written, the machine starts in the FSET state, where it initializes the time, reads the frequency file and transitions to the SYNC state, but steps the time if the offset exceeds the step threshold. The calculations are carefully designed so that time corrections and frequency corrections can be done independently.

In SYNC state the time and frequency are normally adjusted in small increments, unless the offset exceeds the step threshold, in which case the state machine transitions to the SPIK state and avoids setting the time. If further updates exceed the step threshold, they are ignored until after the stepout interval, when the time is stepped. If during the stepout interval an offset is less than the step threshold, the machine transitions to SYNC state and operations resume normally.

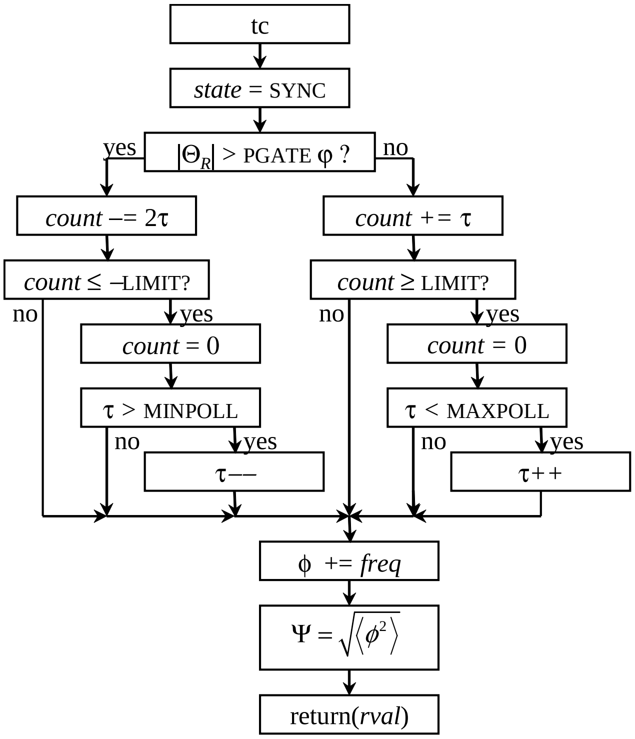


Figure 19. local_clock() Routine (2 of 2)

The remaining portion of the local_clock() routine is shown in Figure 19. The time constant τ is determined by comparing the clock jitter φ with the magnitude of the current residual offset $\Theta_{R.}$ produced by the clock adjust routine in the next section. If the residual offset is greater than PGATE (4) times the clock jitter, be hysteresis counter is reduced by two; otherwise, it is increased by one. If the hysteresis counter increases to the upper limit LIMIT (30), the time constant is increased by one; if it decreases to the lower limit –LIMIT (–30), the time constant is decreased by one. Normally, the time constant hovers near MAXPOLL, but quickly decreases it frequency surges due to a temperature spike, for example.

The clock jitter statistic ϑ in Figure 18 and the clock wander statistic Ψ in Figure 19 are implemented as exponential averages of RMS offset differences and RMS frequency differences, respectively. Let x_i be a measurement at time i of either ϑ or Ψ, $y_i = x_i - x_{i-1}$ the first-order sample difference and $\hat{y}_i$ the exponential average,. Then,

$$\hat{y}_{i+1} = \sqrt{\hat{y}_i^2 + \frac{y_i^2 - \hat{y}_i^2}{\mathrm{AVG}}}, \qquad (15)$$

.where AVG (4) is the averaging parameter in Table 9, is the exponential average at time $i + 1$. The clock jitter statistic is used by the poll-adjust algorithm above; the clock wander statistic is used only for performance monitoring. It is most useful as a canary to detect stalled fans and failed air conditioning systems.

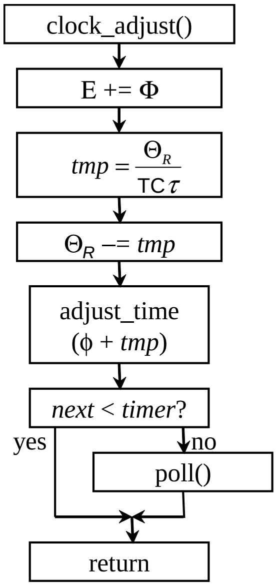


Figure 20. clock_adjust() Routine

9.3 Clock Adjust Process

The actual clock adjustment is performed by the clock_adjust() routine shown in Figure 20 and Section B.7.1. It runs at one-second intervals to add the frequency offset ϕ in Figure 19 and a fixed percentage of the residual offset $\Theta_{R.}$ in Figure 18. The Θ_R. is in effect the exponential decay of the Θ 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 E increases by Φ at each second.

The clock adjust process includes a timer interrupt facility driving the system timer `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 system timer 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.

10. 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. The discussion in this section covers only the variables and routines necessary for a conforming NTPv4 implementation. Additional implementation details are in Section B.8. Further details and rationale for the engineering design are discussed in [2].

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

Table 11. Poll Process Variables and Parameters

10.1 Poll Process Variables and Parameters

The poll process variables are allocated in the association data structure along with the peer process variables. Table 11 shows the names, formula names and short definition for each one. Following is a detailed description of the variables, all of which carry the p prefix.

p.hpoll Signed integer representing the poll exponent, in log2 seconds.

p.last Integer representing the system timer value when the most recent packet was sent.

p.next Integer representing the system timer value when the next packet is to be sent.

p.reach 8-bit integer shift register. When a packet is sent, the register is shifted left one bit, with zero entering from the right and overflow bits discarded.

p.unreach Integer representing the number of seconds the server has been unreachable.

10.2 Poll Process Operations

As described previously, once each second the clock_adjust() routine is called. This routine calls the poll() routine in Figure 21 and Section B.8.1 for each association in turn. If the time for the next poll message is greater than the system timer, the routine returns immediately. A mode-5 (broadcast server) association always sends a packet, but a mode-6 (broadcast client) association never sends a packet, but runs the routine to update the p.reach and p.unreach variables. The poll() routine calls the peer_xmit() routine in Figure 22 and Section B.8.3 to send a packet. If in a burst (p.burst > 0), nothing further is done except call the poll_update() routine to set the next poll interval.

If not in a burst, the p.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()

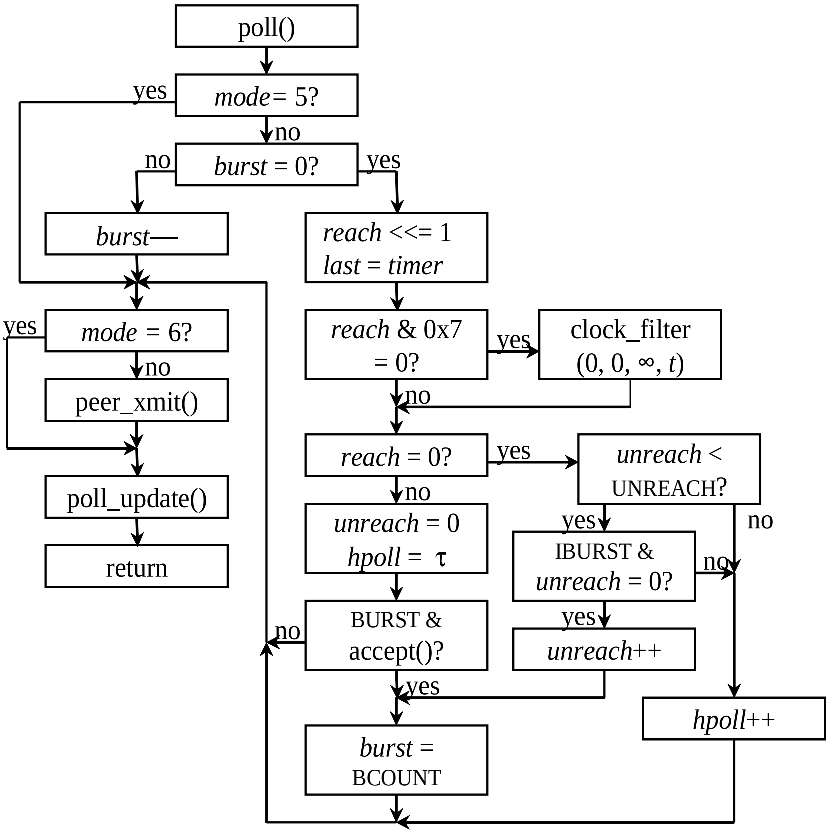
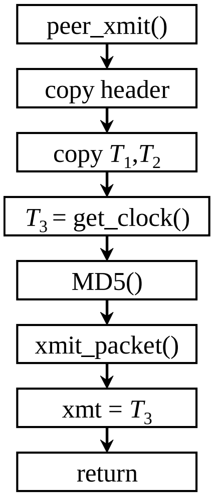


Figure 21. poll() Routine

routine is called to increase the dispersion as described in Section 8.3. 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. 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 becomes unreachable, the client sends a burst. This is useful to quickly reduce the synchronization distance below the distance threshold and synchronize the clock.

The figure also shows the mechanism which backs off the poll interval if the server becomes unreachable. If `p.reach` is nonzero, the server is reachable and `p.unreach` is set to zero; otherwise, `p.unreach` is incremented by one for each poll to the maximum UNREACH (24). Thereafter for each poll `p.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, `p.unreach` is set to zero, `p.hpoll` is reset to τ and operation resumes normally.

When a packet is sent from an association, some header values are copied from the peer variables left by a previous packet and others from the system variables. Figure 22 includes a flow diagram and a table showing 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`	←	`p.version`
`x.mode`	←	`p.mode`
`x.stratum`	←	`s.stratum`
`x.poll`	←	`p.hpoll`
`x.precision`	←	`s.prec`
`x.rootdelay`	←	`s.rootdelay`
`x.rootdisp`	←	`s.rootdisp`
`x.refid`	←	`s.refid`
`x.reftime`	←	`s.reftime`
`x.`*`org`*	←	`p.org`
`x.`*`rec`*	←	`p.rec`
`x.`*`xmt`*	←	clock
`x.keyid`	←	`p.keyid`
`x.digest`	←	md5()

Figure 22. transmit() Routine

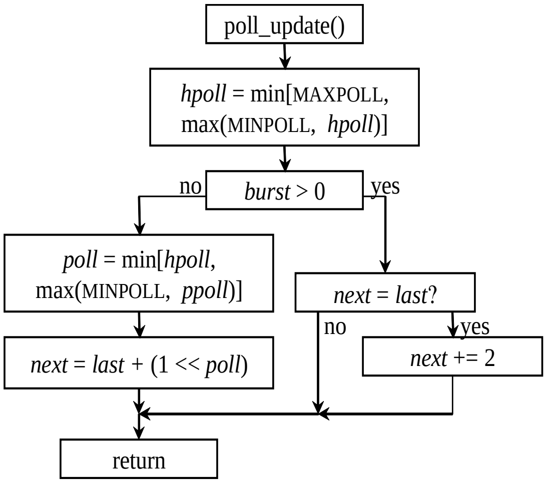

poll_update()
hpoll = min[MAXPOLL, max(MINPOLL, *hpoll*)]
no
burst > 0
yes
poll = min[*hpoll*, max(MINPOLL, *ppoll*)]
next = *last*?
no
yes
next = *last* + (1 << *poll*)
next += 2
return

Figure 23. poll_update() Routine

The poll_update() routine shown in Figure 23 and Section B.8.2 is called when a valid packet is received and immediately after a poll message is sent. If in a burst, the poll interval is fixed at 2 s; otherwise, the host poll exponent is set to the minimum of `p.poll` from the last packet received and `p.hpoll` from the poll() routine, but not less than MINPOLL nor 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. Finally, the poll exponent is converted to an interval which establishes the time at the next poll `p.next`.

11. Simple Network Protocol (SNTP)

In general, a fully conforming NTPv4 design must include all the protocols and algorithms described in this document. However, there are many configurations where some algorithms are not necessary or even useful and a simplified design is possible. This section discusses common configurations and algorithm requirements and, in particular, the Simple Network Time Protocol (SNTP) described in RFC 4330 [4]. That document contains an informal specification and best practices summary for noninvasive product designs with millions of clients in the Internet. This section updates that document and is intended as a basis for formal specification.

Recall that primary servers are synchronized to one or more reference clocks and provide synchronization to possibly many downstream secondary servers and clients. Secondary servers are synchronized to one or more upstream servers and provide synchronization to possibly many downstream secondary servers and clients. Accuracy expectations for primary and secondary servers are stringent ranging from microseconds to low milliseconds. Clients are synchronized to one or more upstream serves but do not provide synchronization to any other host. Accuracy expectations for typical client workstations and PCs is only modest ranging from a tenth of a second to several seconds.

There are many configurations where the full suite of the NTP protocol and algorithms are not required. In particular, the selection, clustering and combining algorithms are not required if a host has only a single upstream server or reference clock. The clock discipline algorithm is not necessary if a host has no dependent downstream clients. The clock filter algorithm is required for secondary servers, but not for primary servers or clients.

In order to insure that a multi-stratum NTP subnet is stable and well behaved in response to nominal network transients, the impulse response of the clock discipline algorithm must be carefully controlled. In a multiple-stratum subnet each stratum behaves as a lowpass filter, so the impulse response for a path from a primary reference clock to a dependent subnet host is determined by a cascade of these filters. In general, this requires that the NTP clock discipline algorithm is implemented in all servers that provide synchronization to downstream secondary servers and clients.

Some primary server designs include specialized hardware which can discipline the system clock time and frequency with high accuracy and in these cases the NTP algorithm is not necessary. In some cases, such as relatively small and constant poll intervals the NTP design can be simplified; however, whatever algorithm is used it must have the same impulse response as the NTP algorithm.

11.1 SNTPv4 Primary Server Configuration

With the above considerations in mind, it is possible to establish the protocol and algorithm requirements for a subset of NTPv4 defined as SNTPv4. There are two configurations that apply, one the SNTPv4 primary server and the other the SNTPv4 client. The SNTPv4 primary server requires:

1. Only a single reference clock can be used.

2. Support for symmetric key authentication is not required; however, if provided it must conform to the MD5 authentication scheme described in Appendix A.

3. If hardware means (e.g., pulse-per-second (PPS) signal and a kernel-based discipline) is not available, the NTP clock discipline algorithm is required.

4. The packet processing steps are defined in Figure 5 in which the packet variables (`r` and `x` prefixes) are defined in Section 6.3 and the system variables (`s` prefix) are defined in Section 9.1. The system variables are determined as follows:

   `s.leap` Set by the reference clock driver from data in the radio or modem protocol.

   `s.stratum` 1 (one).

   `s.precision` Set as described in Section 9.1.

   `s.rootdelay` 0 (zero),

   `s.rootdisp` Computed as in (6), where ρ_R is the precision assigned to the reference clock itself and ρ is the `s.precision` defined above. For the timestamps, T_1 is the time the radio or modem protocol last determined the time, ordinarily the `s.reftime` defined below, and T_1 is the current time. The `s.roodisp` then grows at rate Φ (15 PPM) after that.

   `s.refid` Set by the reference clock driver to one of the values described in Section 6.3.

   `s.reftime` Set by the reference clock driver to the time the radio or modem protocol last determined the time.

11.2 SNTPv4 Client Configuration

There is a wide spectrum of SNTPv4 client configurations, with each providing different levels of accuracy and reliability. Since only local client applications are supported, to use or not use one or more of the NTPv4 algorithms is a matter of local choice. As a bare minimum, a SNTPv4 client application constructs a packet with only the version number filled in and sends it to the NTPv4 or SNTPv4 server. The server operates as in the previous section to return the packet, but only the transmit timestamp (T_3) is useful. The SNTPv4 client application converts from NTP timestamp format to system time format, sets the system clock to this value and exits. The application can be called at defined intervals or manually.

The above scheme provides only nominal accuracy, as it does not correct for the client-server propagation time. Where the resolution of the system clock is in the order of 10 ms or less, a simple improvement is to employ the on-wire protocol described in Section 7 to calculate the clock offset and roundtrip delay as in (4) and (5), respectively. This requires the client to convert from system time format to NTP timestamp format. In addition, the designer may choose to employ the error checks summarized in Table 7. It is possible to go beyond this model all the way to a conforming NTPv4 client. However, if full NTPv4 conformance is claimed, the implementation must include all of the algorithms in this document; anything less is an SNTPv4 implementation.

It is important in SNTPv4 client implementations to conform to the best practices described in RFC 4330 [4], which specify the minimum time between successive messages, the renewal time for DNS caching and other related matters. In the real world of NTP today there are many millions of workstation and PC clients, so clients are expected to be on good behavior to minimize the Internet and server load as accuracy expectations permit.

12. References

1. Marzullo, K., and S. Owicki. Maintaining the time in a distributed system. *ACM Operating Systems Review 19, 3* (July 1985), 44-54.

2. Mills, D.L. *Computer Network Time Synchronization - the Network Time Protocol.* CRC Press, 2006, 304 pp.

3. Mills, D.L. The Autokey security architecture, protocol and algorithms. Electrical and Computer Engineering Technical Report 06-1-1, University of Delaware, January 2006, 59 pp.

4. Mills, D., D. Plonka and J. Montgomery. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. Network Working Group Report RFC-4330, University of Delaware, December 2005, 27 pp.

5. Mills, D.L., A. Thyagarajan and B.C. Huffman. Internet timekeeping around the globe. *Proc. Precision Time and Time Interval (PTTI) Applications and Planning Meeting* (Long Beach CA, December 1997).

6. Mills, D.L., Network Time Protocol (Version 3) specification, implementation and analysis. Network Working Group Request for Comments RFC-1305, University of Delaware, March 1992.

7. Rivest, R. “The MD5 message-digest algorithm. Network Working Group Request for Comments RFC-1321. MIT Laboratory for Computer Science and RSA Data Security, Inc., April 1992.

Appendix A. NTPv4 Packet Formats

The NTP packet consists of a 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 [3]. The optional MAC is used by both Autokey and the symmetric key cryptographic algorithms described in the main body of this report.

A.1 NTP Header Field Format

The NTP header format is shown in Figure 24, where the size of some multiple-word fields is shown in bits if not the default 32 bits. The header extends from the beginning of the packet to the end of the Transmit Timestamp field. The interpretation of the header fields is shown in the main body of this report. When using the IPv4 address family these fields are backwards compatible with NTPv3. 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. The incidence of this, which would be considered a birthday event, will be very rare.

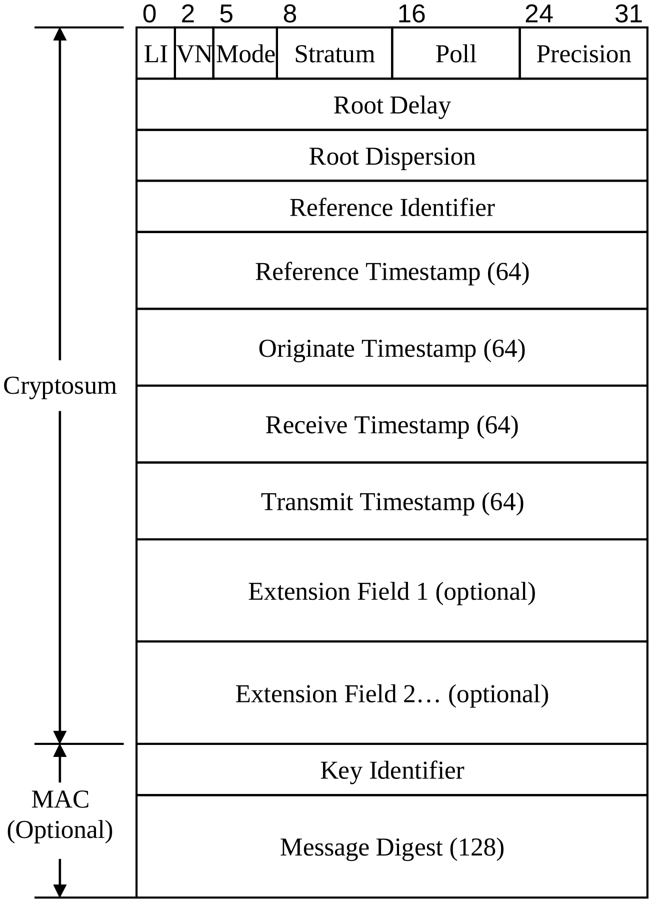


Figure 24. NPv4 Header Format

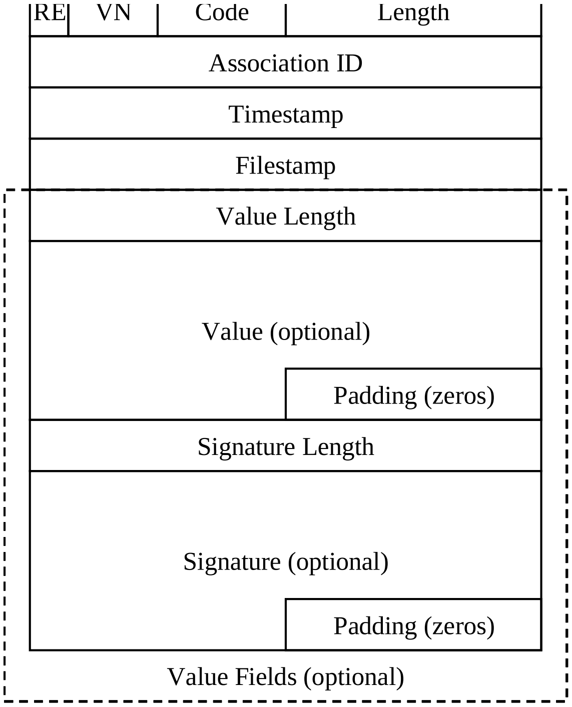


Figure 25. NTPv4 Extension Field Format

The message authentication code (MAC) consists of a 32-bit Key Identifier followed by a 128-bit Message Digest. The message digest, or cryptosum, is calculated as in RFC-1321 [7] over the fields shown in the figure.

A.2 NTPv4 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 extension fields are present. The extension fields can occur in any order; however, in some cases there is a preferred order which improves the protocol efficiency. While previous versions of the Autokey protocol used several different extension field formats, in version 2 of the protocol only a single extension field format is used.

An extension field contains a request or response message in the format shown in Figure 25. All extension fields are zero-padded to a word (4 octets) boundary. The Length field covers the entire extension field, including the Length and Padding fields. While the minimum field length is 4 words (16 octets), a maximum field length remains to be established. The reference implementation discards any packet with an extension field length more than 1024 octets.

The presence of the MAC and extension fields in the packet is determined from the length of the remaining area after the header to the end of the packet. The parser initializes a pointer just after the header. If the Length field is not a multiple of 4, a format error has occurred and the packet is discarded. The following cases are possible based on the remaining length in words.

0 The packet is not authenticated.
1 The packet is an error report or crypto-NAK.
2, 3, 4 The packet is discarded with a format error.
5 The remainder of the packet is the MAC.
>5 One or more extension fields are present.

If an extension field is present, the parser examines the Length field. If the length is less than 4 or not a multiple of 4, a format error has occurred and the packet is discarded; otherwise, the parser increments the pointer by this value. The parser now uses the same rules as above to determine whether a MAC is present and/or another extension field. An additional implementation-dependent test is necessary to ensure the pointer does not stray outside the buffer space occupied by the packet.

In the Autokey Version 2 format, the 8-bit Code field specifies the request or response operation, while the 6-bit Version Number (VN) field is 2 for the current protocol version. The R bit is lit for a response and the E bit lit for an error. The Timestamp and Filestamp fields carry the seconds field of an NTP timestamp. The Timestamp field establishes the signature epoch of the data field in the message, while the Filestamp field establishes the generation epoch of the file that ultimately produced the data that is signed. The optional Value field carries the data and the optional Signature field the signature that validates the data. Further details are in [3].

Appendix B. Code Skeleton

This appendix is intended to describe the protocol and algorithms of the reference implementation in a general way using what is called a code skeleton program. This consists of a set of definitions, structures and code segments which illustrate the protocol operations without the complexities of the actual reference implementation code. This program is not an executable and is not designed to run in the ordinary sense. It is designed to be compiled only in order to verify consistent variable and type usage. The program is not intended to be fast or compact, just to demonstrate the algorithms with sufficient fidelity to understand how they work.

Most of the features of the reference implementation are included in the code skeleton, with the following exceptions: There are no provisions for reference clocks, server discovery 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 code skeleton consists of five segments, a header segment included by each of the other segments, plus a code segment for the main program and peer, system, clock_adjust and poll processes. These are presented in order below along with definitions and variables specific to each process.

B.1 Global Definitions

Following are definitions and other data shared by all programs. These values are defined in a header file ntp4.h which is included in all files.

B.1.1 Definitions, Constants and Parameters

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

/*
 * 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 represeted 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 tstamp;         /* NTP timestamp format */
typedef unsigned int tdist;           /* NTP short format */
typedef unsigned long ipaddr;         /* IPv4 or IPv6 address */
typedef unsinged int ipport;          /* IP port number */
typedef unsigned long digest;         /* md5 digest */
typedef signed char s_char;           /* precision and poll interval (log2) */

/*
 * Arithmetic conversion macroni
 */
#define   LOG2D(a)  ((a) < 0 ? 1. / (1L << -(a)) : \
                    1L << (a))              /* poll, etc. */
#define   LFP2D(a)  ((double)(a) / 0x100000000L) /* NTP timestamp */
#define   D2LFP(a)  ((tstamp)((a) * 0x100000000L))
#define   FP2D(a)   (double)(a) / 0x10000L)     /* NTP short */
#define   D2FP(a)   ((tdist)((a) * 0x10000L))
#define   SQUARE(x) (x * x)
#define   SQRT(x)   (sqrt(x))

/*
 * Global constants. Some of these might be converted to variables
 * which can be tinkered by configuration or computed on-fly. For
 * instance, the reference implementation computes PRECISION on-fly and
 * provides performance tuning for the defines marked with % below.
 */
#define   VERSION             4         /* version number */
#define   PORT                123       /* NTP port number */
#define   MINDISP             .01       /* % minimum dispersion (s) */
#define   MAXDISP             16        /* % maximum dispersion (s) */
#define   MAXDIST             1         /* % distance threshold (s) */
#define   NOSYNC              3         /* leap unsync */
#define   MAXSTRAT            16        /* maximum stratum (infinity metric) */
#define   MINPOLL             4         /* % minimum poll interval (64 s)*/
#define   MAXPOLL             17        /* % maximum poll interval (36.4 h) */
#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 */
```

```
#define   NULL               0         /* empty pointer */

/*
 * 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 */

/*
 * 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 definitionss
 */
#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 */
```

B.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). NTPv4 supports optional extension fields which
 * are inserted after the 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 omited 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   digest;             /* 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    digest;             /* message digest */
} x;
```

B.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 */
          ipport    srcport;            /* source port number *.
          ipaddr    dstaddr;            /* destination (local) address */
          ipport    dstport;            /* destination port number */
          char      version;            /* version number */
          char      mode;               /* mode */
          int       keyid;              /* key identifier */
          int       flags;              /* option flags */

          /*
           * Variables set by received packet
           */
          char      leap;               /* leap indicator */
          char      mode;               /* 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 */
#define   end_clear unreach             /* end of clear area */
          int       unreach;            /* unreach counter */
          int       last;               /* last poll time */
          int       next;               /* next poll time */
} p;
```

B.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 */
} s;
```

B.1.5 Local Clock Data Structure

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

B.1.6 Function Prototypes

```
/*
 * Peer process
 */
void     receive(struct r *);        /* receive packet */
void     fast_xmit(struct r *, int, int); /* transmit a reply packet */
struct p *find_assoc(struct r *);     /* search the association table */
void     packet(struct p *, struct r *); /* process packet */
void     clock_filter(struct p *, double, double, double); /* filter */
int      accept(struct p *);         /* determine fitness of server */
int      access(struct r *);         /* determine access restrictions */
```

```
/*
 * System process
 */
void      clock_select();              /* find the best clocks */
void      clock_update(struct p *);    /* update the system clock */
void      clock_combine();             /* combine the offsets */
double    root_dist(struct p *);       /* calculate root distance */

/*
 * Clock discipline 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 */

/*
 * Main program and utility routines
 */
int       main();                      /* main program */
struct p *mobilize(ipaddr, ipaddr, int, int, int, int); /* mobilize */
void      clear(struct p *, int);      /* clear association */
digest    md5(int);                    /* generate a message digest */

/*
 * Kernel I/O Interface
 */
struct r *recv_packet();               /* wait for packet */
void      xmit_packet(struct x *);     /* send packet */

.*
 * Kernel system clock interface
 */
void      step_time(double);           /* step time */
void      adjust_time(double);         /* adjust (slew) time */
tstamp    get_time();                  /* read time */
```

B.2 Main Program and Utility Routines

```
#include “ntp4.h”

/*
```

```
 * 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 spcified 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);
	}
}

/*
 * 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->srcport = PORT;
	p->dstaddr = dstaddr;
	p->dstport = PORT;
	p->version = version;
	p->mode = mode;
	p->keyid = keyid;
	p->hpoll = MINPOLL;
	clear(p, X_INIT);
	p->flags == flags;
	return (p);
}

/*
 * 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->last = p->t = c.t;
        p->next = p->last + (random() & ((1 << MINPOLL) - 1));
}

/*
 * 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 fieds
         * 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);
```

```
}
```

B.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 */
}
```

B.4 Kernel System Clock Interface

```
*
 * 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 ((unix_time.tv_sec + JAN_1970) * 0x100000000L +
            (unix_time.tv_usec * 0x100000000L) / 1000000);
}

/*
 * step_time() - step system time to given offset valuet
 */
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.
         */
        ntp_time = D2LFP(offset);
        gettimeofday(&unix_time, NULL);
        unix_time.tv_sec += ntp_time / 0x100000000L;
        unix_time.tv_usec += ntp_time % 0x100000000L;
        unix_time.tv_sec += unix_time.tv_usec / 1000000;
        unix_time.tv_usec %= 1000000;
        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 / 0x100000000L;
        unix_time.tv_usec = ntp_time % 0x100000000L;
        unix_time.tv_sec += unix_time.tv_usec / 1000000;
        unix_time.tv_usec %= 1000000;
        adjtime(&unix_time, NULL);
```

```
}
```

B.5 Peer Process

```
#include “ntp4.h”

/*
 * 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 has an NTP header with leap 3 (NOSYNC) and
 * stratum 0. It tells the receiver that something drastic
 * has happened, as revealled by the kiss code in the refid field. The
 * NTP header fields may or may not be correct.
 */
/*
 * Definitions
 */
#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   NEWPS     4                  /* new symmetric passive client */
#define   NEWBC     5                  /* new broadcast client */

/*
 * Dispatch matrix
 *                  active  passv  client server bcast */
int table[7][5] = {
/* nopeer  */{ NEWPS, DSCRD, FXMIT, DSCRD, 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, othewise 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))
```

B.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 */
          int       auth;              /* authentication code */

          /*
           * 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 which 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->mode is assumed NULL.
           */
          p = find_assoc(r);
          switch(table[p->mode][r->mode]) {

          /*
           * Client packet. Send server reply (no association). If
           * authentication fails, send a crypto-NAK packet.
           */
          case FXMIT:
                  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 */

          /*
           * New symmetric passive client (ephemeral 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 (ephemeral association). It is mobilized
           * in the same version as in the packet. If authentication
           * error, 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);
}

/*
 * 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 and source port.
         */
        while (/* all associations */ 0) {
                if (r->srcaddr == p->srcaddr && r->port == p->port)
                        return(p);
        }
        return (NULL);
}
```

B.5.2 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->mode = 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 preseve 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->mode == 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);
}
```

B.5.3 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;
}
```

B.5.4 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.digest = md5(x.keyid);
                }
        }
        xmit_packet(&x);
}
```

B.5.5 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);
}
```

B.6 System Process

```
#include “ntp4.h”
```

B.6.1 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 intersecion 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 (accept(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
                 * nonempty, 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.
         */
        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);
                n++;
        }

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

B.6.2 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);
}
```

B.6.3 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);
}
```

B.6.4 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 valurs, 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;
        }
}
```

B.6.5 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);
}
```

B.6.6 local_clock()

```
#include “ntp4.h”

/*
```

```
 * 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              /* maximum frequency tolerance (s/s) */
#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 outlyer amd
                 * switch to S_SPIK state.
                 */
                case SYNC:
                        state = SPIK;
                        return (rval);
```

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

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

/*
 * In S_SPIK state we ignore succeeding outlyers until
 * either an inlyer 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;
                        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:
                        c.offset = offset;
                        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:
                        c.offset = 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.base - 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
                         * depend 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);
                        break;
                }
                rstclock(SYNC, p->t, offset);
        }

        /*
         * 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 withthe 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);
}
```

B.6.7 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.base = offset - c.offset;
         c.last = c.offset = offset;
         s.t = t;
}
```

B.7 Clock Adjust Process

B.7.1 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 unaccept for
           * synchroniztion.
           */
          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->next)
                              poll(p);
          }

          /*
           * Once per hour write the clock frequency to a file
           */
          if (c.t % 3600 == 3599)
                    /* write c.freq to file */ 0;
}
```

B.8 Poll Process

```
#include “ntp4.h”

/*
 * Constants
 */
#define   UNREACH             12        /* unreach counter threshold */
#define   BCOUNT              8         /* packets in a burst */
#define   BTIME               2         /* burst interval (s) */
```

B.8.1 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->next. The value p->last is
           * the last time this routine was executed. The poll_update()
           * routine determines the next execution time p->next.
           *
           * If broadcasting, just do it, but only if we are synchronized.
           */
          hpoll = p->hpoll;
          if (p->mode == M_BCST) {
                    p->last = c.t;
                    if (s.p != NULL)
                              peer_xmit(p);
                    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.
                     */
```

```
                p->last = c.t;
                oreach = p->reach;
                p->reach << 1;
                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.
                         */
                        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. However, if has not
                         * been heard for three consecutive poll
                         * intervals, stuff the clock register to
                         * increase the peer dispersion. This makes old
                         * servers less desirable and eventually boots
                         * them off the island.
                         */
                        p->unreach = 0;
                        if (!(p->reach & 0x7))
                                clock_filter(p, 0, 0, MAXDISP);
                        hpoll = s.poll;
                        if (p->flags & P_BURST && accept(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->mode != M_BCLN)
                peer_xmit(p);
        poll_update(p, hpoll);
}
```

B.8.2 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       hpoll               /* poll interval (log2 s) */
          )
{
          int       poll;

          /*
           * 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 insures that a longer interval can be preempted by a
           * shorter one if required for rapid response.
           */
          p->hpoll = min(MAXPOLL, max(MINPOLL, hpoll));
          if (p->burst != 0) {
                    if(c.t != p->next)
                              return;

                    p->next += BTIME;
          } else {
                    poll = min(p->hpoll, max(MINPOLL, ppoll));
          }
                    /*
                     * While not shown here, the reference implementation
                     * randonizes the poll interval by a small factor.
                     */
                    p->next = p->last + (1 << poll);
          }

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

B.8.3 transmit()

```
/*
 * 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 = VERSION;
          x.mode = p->mode;
          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.digest = md5(p->keyid);
          xmit_packet(&x);
}
```