1. Introduction

This document constitutes a formal specification of the Network Time Protocol (NTP) Version 3, which is used to synchronize timekeeping among a set of distributed time servers and clients. It defines the architectures, algorithms, entities and protocols used by NTP and is intended primarily for implementors. A companion document [MIL91a] summarizes the requirements, analytical models, algorithmic analysis and performance under typical Internet conditions. Another document [MIL91b] describes the NTP timescale and its relationship to other standard timescales now in use. NTP was first described in RFC-958 [MIL85c], but has since evolved in significant ways, culminating in the most recent NTP Version 2 described in RFC-1119 [MIL89]. It is built on the Internet Protocol (IP) [DAR81a] and User Datagram Protocol (UDP) [POS80], which provide a connectionless transport mechanism; however, it is readily adaptable to other protocol suites. NTP is evolved from the Time Protocol [POS83b] and the ICMP Timestamp message [DAR81b], but is specifically designed to maintain accuracy and robustness, even when used over typical Internet paths involving multiple gateways, highly dispersive delays and unreliable nets.

The service environment consists of the implementation model and service model described in Section 2. The implementation model is based on a multiple-process operating system architecture, although other architectures could be used as well. The service model is based on a returnable-time design which depends only on measured clock offsets, but does not require reliable message delivery. The synchronization subnet uses a self-organizing, hierarchical-master-slave configuration, with synchronization paths determined by a minimum-weight spanning tree. While multiple masters (primary servers) may exist, there is no requirement for an election protocol.

NTP itself is described in Section 3. It provides the protocol mechanisms to synchronize time in principle to precisions in the order of nanoseconds while preserving a non-ambiguous date well into the next century. The protocol includes provisions to specify the characteristics and estimate the error of the local clock and the time server to which it may be synchronized. It also includes provisions for operation with a number of mutually suspicious, hierarchically distributed primary reference sources such as radio-synchronized clocks.

Section 4 describes algorithms useful for deglitching and smoothing clock-offset samples collected on a continuous basis. These algorithms evolved from those suggested in [MIL85a], were refined as the results of experiments described in [MIL85b] and further evolved under typical operating conditions over the last three years. In addition, as the result of experience in operating multiple-server subnets including radio clocks at several sites in the U.S. and with clients in the U.S. and Europe, reliable algorithms for selecting good clocks from a population possibly including broken ones have been developed [DEC89], [MIL91a] and are described in Section 4.

The accuracies achievable by NTP depend strongly on the precision of the local-clock hardware and stringent control of device and process latencies. Provisions must be included to adjust the software logical-clock time and frequency in response to corrections produced by NTP. Section 5 describes a local-clock design evolved from the Fuzzball implementation described in [MIL83b] and [MIL88b]. This design includes offset-slewing, frequency compensation and deglitching

mechanisms capable of accuracies in the order of a millisecond, even after extended periods when synchronization to primary reference sources has been lost.

Details specific to NTP packet formats used with the Internet Protocol (IP) and User Datagram Protocol (UDP) are presented in Appendix A, while details of a suggested auxiliary NTP Control Message, which may be used when comprehensive network-monitoring facilities are not available, are presented in Appendix B. Appendix C contains specification and implementation details of an optional authentication mechanism which can be used to control access and prevent unauthorized data modification, while Appendix D contains a listing of differences between Version 3 of NTP and previous versions. Appendix E expands on issues involved with precision timescales and calendar dating peculiar to computer networks and NTP. Appendix F describes an optional algorithm to improve accuracy by combining the time offsets of a number of clocks. Appendix G presents a detailed mathematical model and analysis of the NTP local-clock algorithms. Appendix H analyzes the sources and propagation of errors and presents correctness principles relating to the time-transfer service. Appendix I illustrates C-language code segments for the clock-filter, clock-selection and related algorithms described in Section 4.

1.1. Related Technology

Other mechanisms have been specified in the Internet protocol suite to record and transmit the time at which an event takes place, including the Daytime protocol [POS83a], Time Protocol [POS83b], ICMP Timestamp message [DAR81b] and IP Timestamp option [SU81]. Experimental results on measured clock offsets and roundtrip delays in the Internet are discussed in [MIL83a], [MIL85b], [COL88] and [MIL88a]. Other synchronization algorithms are discussed in [LAM78], [GUS84], [HAL84], [LUN84], [LAM85], [MAR85], [MIL85a], [MIL85b], [MIL85c], [GUS85b], [SCH86], [TRI86], [RIC88], [MIL88a], [DEC89] and [MIL91a], while protocols based on them are described in [MIL81a], [MIL81b], [MIL83b], [GUS85a], [MIL85c], [TRI86], [MIL88a], [DEC89] and [MIL91a]. NTP uses techniques evolved from them and both linear-systems and agreement methodologies. Linear methods for digital telephone network synchronization are summarized in [LIN80], while agreement methods for clock synchronization are summarized in [LAM85].

The Digital Time Service (DTS) [DEC89] has many of the same service objectives as NTP. The DTS design places heavy emphasis on configuration management and correctness principles when operated in a managed LAN or LAN-cluster environment, while NTP places heavy emphasis on the accuracy and stability of the service operated in an unmanaged, global-internet environment. In DTS a synchronization subnet consists of clerks, servers, couriers and time providers. With respect to the NTP nomenclature, a time provider is a primary reference source, a courier is a secondary server intended to import time from one or more distant primary servers for local redistribution and a server is intended to provide time for possibly many end nodes or clerks. Unlike NTP, DTS does not need or use mode or stratum information in clock selection and does not include provisions to filter timing noise, select the most accurate from a set of presumed correct clocks or compensate for inherent frequency errors.

In fact, the latest revisions in NTP have adopted certain features of DTS in order to support correctness principles. These include mechanisms to bound the maximum errors inherent in the

time-transfer procedures and the use of a provably correct (subject to stated assumptions) mechanism to reject inappropriate peers in the clock-selection procedures. These features are described in Section 4 and Appendix H of this document.

The Fuzzball routing protocol [MIL83b], sometimes called Hellospeak, incorporates time synchronization directly into the routing-protocol design. One or more processes synchronize to an external reference source, such as a radio clock or NTP daemon, and the routing algorithm constructs a minimum-weight spanning tree rooted on these processes. The clock offsets are then distributed along the arcs of the spanning tree to all processes in the system and the various process clocks corrected using the procedure described in Section 5 of this document. While it can be seen that the design of Hellospeak strongly influenced the design of NTP, Hellospeak itself is not an Internet protocol and is unsuited for use outside its local-net environment.

The Unix 4.3bsd time daemon *timed* [GUS85a] uses a single master-time daemon to measure offsets of a number of slave hosts and send periodic corrections to them. In this model the master is determined using an election algorithm [GUS85b] designed to avoid situations where either no master is elected or more than one master is elected. The election process requires a broadcast capability, which is not a ubiquitous feature of the Internet. While this model has been extended to support hierarchical configurations in which a slave on one network serves as a master on the other [TRI86], the model requires handcrafted configuration tables in order to establish the hierarchy and avoid loops. In addition to the burdensome, but presumably infrequent, overheads of the election process, the offset measurement/correction process requires twice as many messages as NTP per update.

A scheme with features similar to NTP is described in [KOP87]. This scheme is intended for multi-server LANs where each of a set of possibly many time servers determines its local-time offset relative to each of the other servers in the set using periodic timestamped messages, then determines the local-clock correction using the Fault-Tolerant Average (FTA) algorithm of [LUN84]. The FTA algorithm, which is useful where up to k servers may be faulty, sorts the offsets, discards the k highest and lowest ones and averages the rest. The scheme, as described in [SCH86], is most suitable to LAN environments which support broadcast and would result in unacceptable overhead in an internet environment. In addition, for reasons given in Section 4 of this paper, the statistical properties of the FTA algorithm are not likely to be optimal in an internet environment with highly dispersive delays.

A good deal of research has gone into the issue of maintaining accurate time in a community where some clocks cannot be trusted. A *truechimer* is a clock that maintains timekeeping accuracy to a previously published (and trusted) standard, while a *falseticker* is a clock that does not. Determining whether a particular clock is a truechimer or falseticker is an interesting abstract problem which can be attacked using agreement methods summarized in [LAM85] and [SRI87].

A convergence function operates upon the offsets between the clocks in a system to increase the accuracy by reducing or eliminating errors caused by falsetickers. There are two classes of convergence functions, those involving interactive-convergence algorithms and those involving interactive-consistency algorithms. Interactive-convergence algorithms use statistical clustering

techniques such as the fault-tolerant average algorithm of [HAL84], the CNV algorithm of [LUN84], the majority-subset algorithm of [MIL85a], the non-Byzantine algorithm of [RIC88], the egocentric algorithm of [SCH86], the intersection algorithm of [MAR85] and [DEC89] and the algorithms in Section 4 of this document.

Interactive-consistency algorithms are designed to detect faulty clock processes which might indicate grossly inconsistent offsets in successive readings or to different readers. These algorithms use an agreement protocol involving successive rounds of readings, possibly relayed and possibly augmented by digital signatures. Examples include the fireworks algorithm of [HAL84] and the optimum algorithm of [SRI87]. However, these algorithms require large numbers of messages, especially when large numbers of clocks are involved, and are designed to detect faults that have rarely been found in the Internet experience. For these reasons they are not considered further in this document.

In practice it is not possible to determine the truechimers from the falsetickers on other than a statistical basis, especially with hierarchical configurations and a statistically noisy Internet. While it is possible to bound the maximum errors in the time-transfer procedures, assuming sufficiently generous tolerances are adopted for the hardware components, this generally results in rather poor accuracies and stabilities. The approach taken in the NTP design and its predecessors involves mutually coupled oscillators and maximum-likelihood estimation and clock-selection procedures, together with a design that allows provable assertions on error bounds to be made relative to stated assumptions on the correctness of the primary reference sources. From the analytical point of view, the system of distributed NTP peers operates as a set of coupled phase-locked oscillators, with the update algorithm functioning as a phase detector and the local clock as a disciplined oscillator, but with deterministic error bounds calculated at each step in the time-transfer process.

The particular choice of offset measurement and computation procedure described in Section 3 is a variant of the returnable-time system used in some digital telephone networks [LIN80]. The clock filter and selection algorithms are designed so that the clock synchronization subnet self-organizes into a hierarchical-master-slave configuration [MIT80]. With respect to timekeeping accuracy and stability, the similarity of NTP to digital telephone systems is not accidental, since systems like this have been studied extensively [LIN80], [BRA80]. What makes the NTP model unique is the adaptive configuration, polling, filtering, selection and correctness mechanisms which tailor the dynamics of the system to fit the ubiquitous Internet environment.

2. System Architecture

In the NTP model a number of primary reference sources, synchronized by wire or radio to national standards, are connected to widely accessible resources, such as backbone gateways, and operated as primary time servers. The purpose of NTP is to convey timekeeping information from these servers to other time servers via the Internet and also to cross-check clocks and mitigate errors due to equipment or propagation failures. Some number of local-net hosts or gateways, acting as secondary time servers, run NTP with one or more of the primary servers. In order to reduce the protocol overhead, the secondary servers distribute time via NTP to the remaining local-net hosts. In the interest of reliability, selected hosts can be equipped with less accurate but less expensive

radio clocks and used for backup in case of failure of the primary and/or secondary servers or communication paths between them.

Throughout this document a standard nomenclature has been adopted: the *stability* of a clock is how well it can maintain a constant frequency, the *accuracy* is how well its frequency and time compare with national standards and the *precision* is how precisely these quantities can be maintained within a particular timekeeping system. Unless indicated otherwise, the *offset* of two clocks is the time difference between them, while the *skew* is the frequency difference (first derivative of offset with time) between them. Real clocks exhibit some variation in skew (second derivative of offset with time), which is called *drift*; however, in this version of the specification the drift is assumed zero.

NTP is designed to produce three products: *clock offset*, *roundtrip delay* and *dispersion*, all of which are relative to a selected reference clock. Clock offset represents the amount to adjust the local clock to bring it into correspondence with the reference clock. Roundtrip delay provides the capability to launch a message to arrive at the reference clock at a specified time. Dispersion represents the maximum error of the local clock relative to the reference clock. Since most host time servers will synchronize via another peer time server, there are two components in each of these three products, those determined by the peer relative to the primary reference source of standard time and those measured by the host relative to the peer. Each of these components are maintained separately in the protocol in order to facilitate error control and management of the subnet itself. They provide not only precision measurements of offset and delay, but also definitive maximum error bounds, so that the user interface can determine not only the time, but the quality of the time as well.

There is no provision for peer discovery or virtual-circuit management in NTP. Data integrity is provided by the IP and UDP checksums. No flow-control or retransmission facilities are provided or necessary. Duplicate detection is inherent in the processing algorithms. The service can operate in a symmetric mode, in which servers and clients are indistinguishable, yet maintain a small amount of state information, or in client/server mode, in which servers need maintain no state other than that contained in the client request. A lightweight association-management capability, including dynamic reachability and variable poll-rate mechanisms, is included only to manage the state information and reduce resource requirements. Since only a single NTP message format is used, the protocol is easily implemented and can be used in a variety of solicited or unsolicited polling mechanisms.

It should be recognized that clock synchronization requires by its nature long periods and multiple comparisons in order to maintain accurate timekeeping. While only a few measurements are usually adequate to reliably determine local time to within a second or so, periods of many hours and dozens of measurements are required to resolve oscillator skew and maintain local time to the order of a millisecond. Thus, the accuracy achieved is directly dependent on the time taken to achieve it. Fortunately, the frequency of measurements can be quite low and almost always non-intrusive to normal net operations.

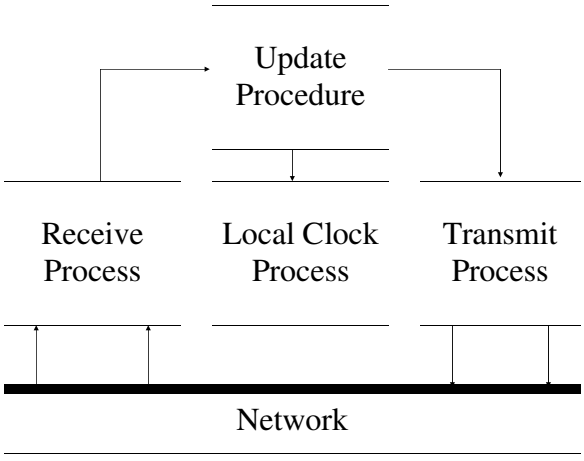

Figure 1. Implementation Model

2.1. Implementation Model

In what may be the most common client/server model a client sends an NTP message to one or more servers and processes the replies as received. The server interchanges addresses and ports, overwrites certain fields in the message, recalculates the checksum and returns the message immediately. Information included in the NTP message allows the client to determine the server time with respect to local time and adjust the local clock accordingly. In addition, the message includes information to calculate the expected timekeeping accuracy and reliability, as well as select the best from possibly several servers.

While the client/server model may suffice for use on local nets involving a public server and perhaps many workstation clients, the full generality of NTP requires distributed participation of a number of client/servers or peers arranged in a dynamically reconfigurable, hierarchically distributed configuration. It also requires sophisticated algorithms for association management, data manipulation and local-clock control. Throughout the remainder of this document the term *host* refers to an instantiation of the protocol on a local processor, while the term *peer* refers to the instantiation of the protocol on a remote processor connected by a network path.

Figure 1 shows an implementation model for a host including three processes sharing a partitioned data base, with a partition dedicated to each peer, and interconnected by a message-passing system. The transmit process, driven by independent timers for each peer, collects information in the data base and sends NTP messages to the peers. Each message contains the local timestamp when the message is sent, together with previously received timestamps and other information necessary to determine the hierarchy and manage the association. The message transmission rate is determined by the accuracy required of the local clock, as well as the accuracies of its peers.

The receive process receives NTP messages and perhaps messages in other protocols, as well as information from directly connected radio clocks. When an NTP message is received, the offset between the peer clock and the local clock is computed and incorporated into the data base along

with other information useful for error determination and peer selection. A filtering algorithm described in Section 4 improves the accuracy by discarding inferior data.

The update procedure is initiated upon receipt of a message and at other times. It processes the offset data from each peer and selects the best one using the algorithms of Section 4. This may involve many observations of a few peers or a few observations of many peers, depending on the accuracies required.

The local-clock process operates upon the offset data produced by the update procedure and adjusts the phase and frequency of the local clock using the mechanisms described in Section 5. This may result in either a step-change or a gradual phase adjustment of the local clock to reduce the offset to zero. The local clock provides a stable source of time information to other users of the system and for subsequent reference by NTP itself.

2.2. Network Configurations

The synchronization subnet is a connected network of primary and secondary time servers, clients and interconnecting transmission paths. A primary time server is directly synchronized to a primary reference source, usually a radio clock. A secondary time server derives synchronization, possibly via other secondary servers, from a primary server over network paths possibly shared with other services. Under normal circumstances it is intended that the synchronization subnet of primary and secondary servers assumes a hierarchical-master-slave configuration with the primary servers at the root and secondary servers of decreasing accuracy at successive levels toward the leaves.

Following conventions established by the telephone industry [BEL86], the accuracy of each server is defined by a number called the *stratum*, with the topmost level (primary servers) assigned as one and each level downwards (secondary servers) in the hierarchy assigned as one greater than the preceding level. With current technology and available radio clocks, single-sample accuracies in the order of a millisecond can be achieved at the network interface of a primary server. Accuracies of this order require special care in the design and implementation of the operating system and the local-clock mechanism, such as described in Section 5.

As the stratum increases from one, the single-sample accuracies achievable will degrade depending on the network paths and local-clock stabilities. In order to avoid the tedious calculations [BRA80] necessary to estimate errors in each specific configuration, it is useful to assume the mean measurement errors accumulate approximately in proportion to the measured delay and dispersion relative to the root of the synchronization subnet. Appendix H contains an analysis of errors, including a derivation of maximum error as a function of delay and dispersion, where the latter quantity depends on the precision of the timekeeping system, frequency tolerance of the local clock and various residuals. Assuming the primary servers are synchronized to standard time within known accuracies, this provides a reliable, determistic specification on timekeeping accuracies throughout the synchronization subnet.

Again drawing from the experience of the telephone industry, which learned such lessons at considerable cost [ABA89], the synchronization subnet topology should be organized to produce the highest accuracy, but must never be allowed to form a loop. An additional factor is that each

increment in stratum involves a potentially unreliable time server which introduces additional measurement errors. The selection algorithm used in NTP uses a variant of the Bellman-Ford distributed routing algorithm [37] to compute the minimum-weight spanning trees rooted on the primary servers. The distance metric used by the algorithm consists of the (scaled) stratum plus the *synchronization distance*, which itself consists of the dispersion plus one-half the absolute delay. Thus, the synchronization path will always take the minimum number of servers to the root, with ties resolved on the basis of maximum error.

As a result of this design, the subnet reconfigures automatically in a hierarchical-master-slave configuration to produce the most accurate and reliable time, even when one or more primary or secondary servers or the network paths between them fail. This includes the case where all normal primary servers (e.g., highly accurate WWVB radio clock operating at the lowest synchronization distances) on a possibly partitioned subnet fail, but one or more backup primary servers (e.g., less accurate WWV radio clock operating at higher synchronization distances) continue operation. However, should all primary servers throughout the subnet fail, the remaining secondary servers will synchronize among themselves while distances ratchet upwards to a preselected maximum "infinity" due to the well-known properties of the Bellman-Ford algorithm. Upon reaching the maximum on all paths, a server will drop off the subnet and free-run using its last determined time and frequency. Since these computations are expected to be very precise, especially in frequency, even extended outage periods can result in timekeeping errors not greater than a few milliseconds per day with appropriately stabilized oscillators (see Section 5).

In the case of multiple primary servers, the spanning-tree computation will usually select the server at minimum synchronization distance. However, when these servers are at approximately the same distance, the computation may result in random selections among them as the result of normal dispersive delays. Ordinarily, this does not degrade accuracy as long as any discrepancy between the primary servers is small compared to the synchronization distance. If not, the filter and selection algorithms will select the best of the available servers and cast out outlyers as intended.

3. Network Time Protocol

This section consists of a formal definition of the Network Time Protocol, including its data formats, entities, state variables, events and event-processing procedures. The specification is based on the implementation model illustrated in Figure 1, but it is not intended that this model is the only one upon which a specification can be based. In particular, the specification is intended to illustrate and clarify the intrinsic operations of NTP, as well as to serve as a foundation for a more rigorous, comprehensive and verifiable specification.

3.1. Data Formats

All mathematical operations expressed or implied herein are in two's-complement, fixed-point arithmetic. Data are specified as integer or fixed-point quantities, with bits numbered in big-endian fashion from zero starting at the left, or high-order, position. Since various implementations may scale externally derived quantities for internal use, neither the precision nor decimal-point placement for fixed-point quantities is specified. Unless specified otherwise, all quantities are unsigned and

may occupy the full field width with an implied zero preceding bit zero. Hardware and software packages designed to work with signed quantities will thus yield surprising results when the most significant (sign) bit is set. It is suggested that externally derived, unsigned fixed-point quantities such as timestamps be shifted right one bit for internal use, since the precision represented by the full field width is seldom justified.

Since NTP timestamps are cherished data and, in fact, represent the main product of the protocol, a special timestamp format has been established. NTP timestamps are represented as a 64-bit unsigned fixed-point number, in seconds relative to 0^h on 1 January 1900. The integer part is in the first 32 bits and the fraction part in the last 32 bits. This format allows convenient multiple-precision arithmetic and conversion to Time Protocol representation (seconds), but does complicate the conversion to ICMP Timestamp message representation (milliseconds). The precision of this representation is about 200 picoseconds, which should be adequate for even the most exotic requirements.

Timestamps are determined by copying the current value of the local clock to a timestamp when some significant event, such as the arrival of a message, occurs. In order to maintain the highest accuracy, it is important that this be done as close to the hardware or software driver associated with the event as possible. In particular, departure timestamps should be redetermined for each link-level retransmission. In some cases a particular timestamp may not be available, such as when the host is rebooted or the protocol first starts up. In these cases the 64-bit field is set to zero, indicating the value is invalid or undefined.

Note that since some time in 1968 the most significant bit (bit 0 of the integer part) has been set and that the 64-bit field will overflow some time in 2036. Should NTP be in use in 2036, some external means will be necessary to qualify time relative to 1900 and time relative to 2036 (and other multiples of 136 years). Timestamped data requiring such qualification will be so precious that appropriate means should be readily available. There will exist an 200-picosecond interval, henceforth ignored, every 136 years when the 64-bit field will be zero and thus considered invalid.

3.2. State Variables and Parameters

Following is a summary of the various state variables and parameters used by the protocol. They are separated into classes of system variables, which relate to the operating system environment and local-clock mechanism; peer variables, which represent the state of the protocol machine specific to each peer; packet variables, which represent the contents of the NTP message; and parameters, which represent fixed configuration constants for all implementations of the current version. For each class the description of the variable is followed by its name and the procedure or value which controls it. Note that variables are in lower case, while parameters are in upper case. Additional details on formats and use are presented in later sections and Appendices.

3.2.1. Common Variables

The following variables are common to two or more of the system, peer and packet classes. Additional variables are specific to the optional authentication mechanism as described in Appendix

C. When necessary to distinguish between common variables of the same name, the variable identifier will be used.

Peer Address (peer.peeraddr, pkt.peeraddr), Peer Port (peer.peerport, pkt.peerport): These are the 32-bit Internet address and 16-bit port number of the peer.

Host Address (peer.hostaddr, pkt.hostaddr), Host Port (peer.hostport, pkt.hostport): These are the 32-bit Internet address and 16-bit port number of the host. They are included among the state variables to support multi-homing.

Leap Indicator (sys.leap, peer.leap, pkt.leap): This is a two-bit code warning of an impending leap second to be inserted in the NTP timescale. The bits are set before 23:59 on the day of insertion and reset after 00:00 on the following day. This causes the number of seconds (rollover interval) in the day of insertion to be increased or decreased by one. In the case of primary servers the bits are set by operator intervention, while in the case of secondary servers the bits are set by the protocol. The two bits, bit 0 and bit 1, respectively, are coded as follows:

00	no warning
01	last minute has 61 seconds
10	last minute has 59 seconds
11	alarm condition (clock not synchronized)

In all except the alarm condition (11_2), NTP itself does nothing with these bits, except pass them on to the time-conversion routines that are not part of NTP. The alarm condition occurs when, for whatever reason, the local clock is not synchronized, such as when first coming up or after an extended period when no primary reference source is available.

Mode (peer.mode, pkt.mode): This is an integer indicating the association mode, with values coded as follows:

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

Stratum (sys.stratum, peer.stratum, pkt.stratum): This is an integer indicating the stratum of the local clock, with values defined as follows:

0	unspecified
1	primary reference (e.g., calibrated atomic clock, radio clock)
2-255	secondary reference (via NTP)

For comparison purposes a value of zero is considered greater than any other value. Note that the maximum value of the integer encoded as a packet variable is limited by the parameter NTP.MAXSTRATUM.

Poll Interval (sys.poll, peer.hostpoll, peer.peerpoll, pkt.poll): This is a signed integer indicating the minimum interval between transmitted messages, in seconds as a power of two. For instance, a value of six indicates a minimum interval of 64 seconds.

Precision (sys.precision, peer.precision, pkt.precision): This is a signed integer indicating the precision of the various clocks, in seconds to the nearest power of two. The value must be rounded to the next larger power of two; for instance, a 50-Hz (20 ms) or 60-Hz (16.67 ms) power-frequency clock would be assigned the value -5 (31.25 ms), while a 1000-Hz (1 ms) crystal-controlled clock would be assigned the value -9 (1.95 ms).

Root Delay (sys.rootdelay, peer.rootdelay, pkt.rootdelay): This is a signed fixed-point number indicating the total roundtrip delay to the primary reference source at the root of the synchronization subnet, in seconds. Note that this variable can take on both positive and negative values, depending on clock precision and skew.

Root Dispersion (sys.rootdispersion, peer.rootdispersion, pkt.rootdispersion): This is a signed fixed-point number indicating the maximum error relative to the primary reference source at the root of the synchronization subnet, in seconds. Only positive values greater than zero are possible.

Reference Clock Identifier (sys.refid, peer.refid, pkt.refid): This is a 32-bit code identifying the particular reference clock. In the case of stratum 0 (unspecified) or stratum 1 (primary reference source), this is a four-octet, left-justified, zero-padded ASCII string, for example (see Appendix A for comprehensive list):

Stratum	Code	Meaning
0	DCN	DCN routing protocol
0	TSP	TSP time protocol
1	ATOM	Atomic clock (calibrated)
1	WWVB	WWVB LF (band 5) radio
1	GOES	GOES UHF (band 9) satellite
1	WWV	WWV HF (band 7) radio

In the case of stratum 2 and greater (secondary reference) this is the four-octet Internet address of the peer selected for synchronization.

Reference Timestamp (sys.reftime, peer.reftime, pkt.reftime): This is the local time, in timestamp format, when the local clock was last updated. If the local clock has never been synchronized, the value is zero.

System Variables	Name	Procedure
Leap Indicator	sys.leap	clock update
Stratum	sys.stratum	clock update
Precision	sys.precision	system
Root Delay	sys.rootdelay	clock update
Root Dispersion	sys.rootdispersion	clock update
Reference Clock Ident	sys.refid	clock update
Reference Timestamp	sys.reftime	clock update
Local Clock	sys.clock	clock update
Clock Source	sys.peer	selection
Poll Interval	sys.poll	local clock
Cryptographic Keys	sys.keys	authentication

Table 1. System Variables

Originate Timestamp (peer.org, pkt.org): This is the local time, in timestamp format, at the peer when its latest NTP message was sent. If the peer becomes unreachable the value is set to zero.

Receive Timestamp (peer.rec, pkt.rec): This is the local time, in timestamp format, when the latest NTP message from the peer arrived. If the peer becomes unreachable the value is set to zero.

Transmit Timestamp (peer.xmt, pkt.xmt): This is the local time, in timestamp format, at which the NTP message departed the sender.

3.2.2. System Variables

Table 1 shows the complete set of system variables. In addition to the common variables described previously, the following variables are used by the operating system in order to synchronize the local clock.

Local Clock (sys.clock): This is the current local time, in timestamp format. Local time is derived from the hardware clock of the particular machine and increments at intervals depending on the design used. An appropriate design, including slewing and skew-Compensation mechanisms, is described in Section 5.

Clock Source (sys.peer): This is a selector identifying the current synchronization source. Usually this will be a pointer to a structure containing the peer variables. The special value NULL indicates there is no currently valid synchronization source.

3.2.3. Peer Variables

Table 2 shows the complete set of peer variables. In addition to the common variables described previously, the following variables are used by the peer management and measurement functions.

Configured Bit (peer.config): This is a bit indicating that the association was created from configuration information and should not be demobilized if the peer becomes unreachable.

Peer Variables	Name	Procedure
Configured Bit	peer.config	initialization
Peer Address	peer.peeraddress	receive
Peer Port	peer.peerport	receive
Host Address	peer.hostaddress	receive
Host Port	peer.hostport	receive
Leap Indicator	peer.leap	packet
Mode	peer.mode	packet
Stratum	peer.stratum	packet
Peer Poll Interval	peer.peerpoll	packet
Host Poll Interval	peer.hostpoll	poll update
Precision	peer.precision	packet
Root Delay	peer.rootdelay	packet
Root Dispersion	peer.rootdispersion	packet
Reference Clock Ident	peer.refid	packet
Reference Timestamp	peer.reftime	packet
Originate Timestamp	peer.org	packet, clear
Receive Timestamp	peer.rec	packet, clear
Transmit Timestamp	peer.xmt	transmit, clear
Update Timestamp	peer.update	filter, clear
Reachability Register	peer.reach	packet, transmit, clear
Peer Timer	peer.timer	receive, transmit, poll update
Filter Register	peer.filter	filter
Valid Data Counter	peer.valid	transmit
Delay	peer.delay	filter
Offset	peer.offset	filter
Dispersion	peer.dispersion	filter
Authentic Enable Bit	peer.authenable	authentication
Authenticated Bit	peer.authentic	authentication
Host Key Identifier	peer.hostkeyid	authentication
Peer Key Identifier	peer.peerkeyid	authentication

Table 2. Peer Variables

Update Timestamp (peer.update): This is the local time, in timestamp format, when the most recent NTP message was received. It is used in calculating the skew dispersion.

Reachability Register (peer.reach): This is a shift register of NTP.WINDOW bits used to determine the reachability status of the peer, with bits entering from the least significant (rightmost) end. A peer is considered reachable if at least one bit in this register is set to one.

Packet Variables	Name	Procedure
Peer Address	pkt.peeraddress	transmit
Peer Port	pkt.peerport	transmit
Host Address	pkt.hostaddress	transmit
Host Port	pkt.hostport	transmit
Leap Indicator	pkt.leap	transmit
Version Number	pkt.version	transmit
Mode	pkt.mode	transmit
Stratum	pkt.stratum	transmit
Poll Interval	pkt.poll	transmit
Precision	pkt.precision	transmit
Root Delay	pkt.rootdelay	transmit
Root Dispersion	pkt.rootdispersion	transmit
Reference Clock Ident	pkt.refid	transmit
Reference Timestamp	pkt.reftime	transmit
Originate Timestamp	pkt.org	transmit
Receive Timestamp	pkt.rec	transmit
Transmit Timestamp	pkt.xmt	transmit
Key Identifier	pkt.keyid	authentication
Crypto-Checksum	pkt.check	authentication

Table 3. Packet Variables

Peer Timer (peer.timer): This is an integer counter used to control the interval between transmitted NTP messages. Once set to a nonzero value, the counter decrements at one-second intervals until reaching zero, at which time the transmit procedure is called. Note that the operation of this timer is independent of local-clock updates, which implies that the timekeeping system and interval-timer system architecture must be independent of each other.

3.2.4. Packet Variables

Table 3 shows the complete set of packet variables. In addition to the common variables described previously, the following variables are defined.

Version Number (pkt.version): This is an integer indicating the version number of the sender. NTP messages will always be sent with the current version number NTP.VERSION and will always be accepted if the version number matches NTP.VERSION. Exceptions may be advised on a case-by-case basis at times when the version number is changed. Specific guidelines for interoperation between this version and previous versions of NTP are summarized in Appendix D.

3.2.5. Clock-Filter Variables

When the filter and selection algorithms suggested in Section 4 are used, the following state variables are defined in addition to the variables described previously.

Filter Register (peer.filter): This is a shift register of NTP.SHIFT stages, where each stage stores a 3-tuple consisting of the measured delay, measured offset and calculated dispersion associated with a single observation. These 3-tuples enter from the most significant (leftmost) right and are shifted towards the least significant (rightmost) end and eventually discarded as new observations arrive.

Valid Data Counter (peer.valid): This is an integer counter indicating the valid samples remaining in the filter register. It is used to determine the reachability state and when the poll interval should be increased or decreased.

Offset (peer.offset): This is a signed, fixed-point number indicating the offset of the peer clock relative to the local clock, in seconds.

Delay (peer.delay): This is a signed fixed-point number indicating the roundtrip delay of the peer clock relative to the local clock over the network path between them, in seconds. Note that this variable can take on both positive and negative values, depending on clock precision and skew-error accumulation.

Dispersion (peer.dispersion): This is a signed fixed-point number indicating the maximum error of the peer clock relative to the local clock over the network path between them, in seconds. Only positive values greater than zero are possible.

3.2.6. Authentication Variables

When the authentication mechanism suggested in Appendix C is used, the following state variables are defined in addition to the variables described previously. These variables are used only if the optional authentication mechanism described in Appendix C is implemented.

Authentication Enabled Bit (peer.authenable): This is a bit indicating that the association is to operate in the authenticated mode.

Authenticated Bit (peer.authentic): This is a bit indicating that the last message received from the peer has been correctly authenticated.

Key Identifier (peer.hostkeyid, peer.peerkeyid, pkt.keyid): This is an integer identifying the cryptographic key used to generate the message-authentication code.

Cryptographic Keys (sys.key): This is a set of 64-bit DES keys. Each key is constructed as in the Berkeley Unix distributions, which consists of eight octets, where the seven low-order bits of each octet correspond to the DES bits 1-7 and the high-order bit corresponds to the DES odd-parity bit 8.

Crypto-Checksum (pkt.check): This is a crypto-checksum computed by the encryption procedure.

3.2.7. Parameters

Table 4 shows the parameters assumed for all implementations operating in the Internet system. It is necessary to agree on the values for these parameters in order to avoid unnecessary network

Parameters	Name	Value
Version Number	NTP.VERSION	3
NTP Port	NTP.PORT	123
Max Stratum	NTP.MAXSTRATUM	15
Max Clock Age	NTP.MAXAGE	86,400 sec
Max Skew	NTP.MAXSKEW	1 sec
Max Distance	NTP.MAXDISTANCE	1 sec
Min Polling Interval	NTP.MINPOLL	6 (64 sec)
Max Polling Interval	NTP.MAXPOLL	10 (1024 sec)
Min Select Clocks	NTP.MINCLOCK	1
Max Select Clocks	NTP.MAXCLOCK	10
Min Dispersion	NTP.MINDISPERSE	.01 sec
Max Dispersion	NTP.MAXDISPERSE	16 sec
Reachability Reg Size	NTP.WINDOW	8
Filter Size	NTP.SHIFT	8
Filter Weight	NTP.FILTER	$^1/_2$
Select Weight	NTP.SELECT	$^3/_4$

Table 4. Parameters

overheads and stable peer associations. The following parameters are assumed fixed and applicable to all associations.

Version Number (NTP.VERSION): This is the current NTP version number (3).

NTP Port (NTP.PORT): This is the port number (123) assigned by the Internet Assigned Numbers Authority to NTP.

Maximum Stratum (NTP.MAXSTRATUM): This is the maximum stratum value that can be encoded as a packet variable, also interpreted as “infinity” or unreachable by the subnet routing algorithm.

Maximum Clock Age (NTP.MAXAGE): This is the maximum interval a reference clock will be considered valid after its last update, in seconds.

Maximum Skew (NTP.MAXSKEW): This is the maximum offset error due to skew of the local clock over the interval determined by NTP.MAXAGE, in seconds. The ratio $\varphi = \frac{\text{NTP.MAXSKEW}}{\text{NTP.MAXAGE}}$ is interpreted as the maximum possible skew rate due to all causes.

Maximum Distance (NTP.MAXDISTANCE): When the selection algorithm suggested in Section 4 is used, this is the maximum synchronization distance for peers acceptable for synchronization.

Minimum Poll Interval (NTP.MINPOLL): This is the minimum poll interval allowed by any peer of the Internet system, in seconds to a power of two.

Maximum Poll Interval (NTP.MAXPOLL): This is the maximum poll interval allowed by any peer of the Internet system, in seconds to a power of two.

Minimum Select Clocks (NTP.MINCLOCK): When the selection algorithm suggested in Section 4 is used, this is the minimum number of peers acceptable for synchronization.

Maximum Select Clocks (NTP.MAXCLOCK): When the selection algorithm suggested in Section 4 is used, this is the maximum number of peers considered for selection.

Minimum Dispersion (NTP.MINDISPERSE): When the filter algorithm suggested in Section 4 is used, this is the minimum dispersion increment for each stratum level, in seconds.

Maximum Dispersion (NTP.MAXDISPERSE): When the filter algorithm suggested in Section 4 is used, this is the maximum peer dispersion and the dispersion assumed for missing data, in seconds.

Reachability Register Size (NTP.WINDOW): This is the size of the reachability register (peer.reach), in bits.

Filter Size (NTP.SHIFT): When the filter algorithm suggested in Section 4 is used, this is the size of the clock filter (peer.filter) shift register, in stages.

Filter Weight (NTP.FILTER): When the filter algorithm suggested in Section 4 is used, this is the weight used to compute the filter dispersion.

Select Weight (NTP.SELECT): When the selection algorithm suggested in Section 4 is used, this is the weight used to compute the select dispersion.

3.3. Modes of Operation

Except in broadcast mode, an NTP association is formed when two peers exchange messages and one or both of them create and maintain an instantiation of the protocol machine, called an association. The association can operate in one of five modes as indicated by the host-mode variable (peer.mode): symmetric active, symmetric passive, client, server and broadcast, which are defined as follows:

Symmetric Active (1): A host operating in this mode sends periodic messages regardless of the reachability state or stratum of its peer. By operating in this mode the host announces its willingness to synchronize and be synchronized by the peer.

Symmetric Passive (2): This type of association is ordinarily created upon arrival of a message from a peer operating in the symmetric active mode and persists only as long as the peer is reachable and operating at a stratum level less than or equal to the host; otherwise, the association is dissolved. However, the association will always persist until at least one message has been sent in reply. By operating in this mode the host announces its willingness to synchronize and be synchronized by the peer.

Client (3): A host operating in this mode sends periodic messages regardless of the reachability state or stratum of its peer. By operating in this mode the host, usually a LAN workstation, announces its willingness to be synchronized by, but not to synchronize the peer.

Server (4): This type of association is ordinarily created upon arrival of a client request message and exists only in order to reply to that request, after which the association is dissolved. By operating in this mode the host, usually a LAN time server, announces its willingness to synchronize, but not to be synchronized by the peer.

Broadcast (5): A host operating in this mode sends periodic messages regardless of the reachability state or stratum of the peers. By operating in this mode the host, usually a LAN time server operating on a high-speed broadcast medium, announces its willingness to synchronize all of the peers, but not to be synchronized by any of them.

A host operating in client mode occasionally sends an NTP message to a host operating in server mode, perhaps right after rebooting and at periodic intervals thereafter. The server responds by simply interchanging addresses and ports, filling in the required information and returning the message to the client. Servers need retain no state information between client requests, while clients are free to manage the intervals between sending NTP messages to suit local conditions. In these modes the protocol machine described in this document can be considerably simplified to a simple remote-procedure-call mechanism without significant loss of accuracy or robustness, especially when operating over high-speed LANs.

In the symmetric modes the client/server distinction (almost) disappears. Symmetric passive mode is intended for use by time servers operating near the root nodes (lowest stratum) of the synchronization subnet and with a relatively large number of peers on an intermittent basis. In this mode the identity of the peer need not be known in advance, since the association with its state variables is created only when an NTP message arrives. Furthermore, the state storage can be reused when the peer becomes unreachable or is operating at a higher stratum level and thus ineligible as a synchronization source.

Symmetric active mode is intended for use by time servers operating near the end nodes (highest stratum) of the synchronization subnet. Reliable time service can usually be maintained with two peers at the next lower stratum level and one peer at the same stratum level, so the rate of ongoing polls is usually not significant, even when connectivity is lost and error messages are being returned for every poll.

Normally, one peer operates in an active mode (symmetric active, client or broadcast modes) as configured by a startup file, while the other operates in a passive mode (symmetric passive or server modes), often without prior configuration. However, both peers can be configured to operate in the symmetric active mode. An error condition results when both peers operate in the same mode, but not symmetric active mode. In such cases each peer will ignore messages from the other, so that prior associations, if any, will be demobilized due to reachability failure.

Broadcast mode is intended for operation on high-speed LANs with numerous workstations and where the highest accuracies are not required. In the typical scenario one or more time servers on

the LAN send periodic broadcasts to the workstations, which then determine the time on the basis of a preconfigured latency in the order of a few milliseconds. As in the client/server modes the protocol machine can be considerably simplified in this mode; however, a modified form of the clock selection algorithm may prove useful in cases where multiple time servers are used for enhanced reliability.

3.4. Event Processing

The significant events of interest in NTP occur upon expiration of a peer timer (peer.timer), one of which is dedicated to each peer with an active association, and upon arrival of an NTP message from the various peers. An event can also occur as the result of an operator command or detected system fault, such as a primary reference source failure. This section describes the procedures invoked when these events occur.

3.4.1. Notation Conventions

The NTP filtering and selection algorithms act upon a set of variables for clock offset (θ, Θ), roundtrip delay (δ, Δ) and dispersion (ε, E). When necessary to distinguish between them, lower-case Greek letters are used for variables relative to a peer, while upper-case Greek letters are used for variables relative to the primary reference source(s), i.e., via the peer to the root of the synchronization subnet. Subscripts will be used to identify the particular peer when this is not clear from context. The algorithms are based on a quantity called the synchronization distance (λ, Λ), which is computed from the roundtrip delay and dispersion as described below.

As described in Appendix H, the peer dispersion ε includes contributions due to measurement error $\rho = 1 \ll \text{sys.precision}$, skew-error accumulation $\varphi\tau$, where $\varphi = \frac{\text{NTP.MAXSKEW}}{\text{NTP.MAXAGE}}$ is the maximum skew rate and $\tau = \text{sys.clock} - \text{peer.update}$ is the interval since the last update, and filter (sample) dispersion ε_σ computed by the clock-filter algorithm. The root dispersion E includes contributions due to the selected peer dispersion ε and skew-error accumulation $\varphi\tau$, together with the root dispersion for the peer itself. The system dispersion includes the select (sample) dispersion ε_ξ computed by the clock-select algorithm and the absolute initial clock offset $|\Theta|$ provided to the local-clock algorithm. Both ε and E are dynamic quantities, since they depend on the elapsed time τ since the last update, as well as the sample dispersions calculated by the algorithms.

Each time the relevant peer variables are updated, all dispersions associated with that peer are updated to reflect the skew-error accumulation. The computations can be summarized as follows:

$$\theta \equiv \text{peer.offset},$$
$$\delta \equiv \text{peer.delay},$$
$$\varepsilon \equiv \text{peer.dispersion} = \rho + \varphi\tau + \varepsilon_\sigma,$$
$$\lambda \equiv \varepsilon + \frac{|\delta|}{2},$$

where τ is the interval since the original timestamp (from which θ and δ were determined) was transmitted to the present time and ε_σ is the filter dispersion (see clock-filter procedure below). The variables relative to the root of the synchronization subnet via peer *i* are determined as follows:

$$\Theta_i \equiv \theta_i ,$$
$$\Delta_i \equiv \text{peer.rootdelay} + \delta_i ,$$
$$E_i \equiv \text{peer.rootdispersion} + \varepsilon_i + \varphi\tau_i ,$$
$$\Lambda_i \equiv E_i + \frac{|\Delta_i|}{2} ,$$

where all variables are understood to pertain to the *i*th peer. Finally, assuming the *i*th peer is selected for synchronization, the system variables are determined as follows:

$$\Theta = \text{combined final offset} ,$$
$$\Delta = \Delta_i ,$$
$$E = E_i + \varepsilon\xi + |\Theta| ,$$
$$\Lambda = \Lambda_i ,$$

where $\varepsilon\xi$ is the select dispersion (see clock-selection procedure below).

Informal pseudo-code which accomplishes these computations is presented below. Note that the pseudo-code is represented in no particular language, although it has many similarities to the C language. Specific details on the important algorithms are further illustrated in the C-language routines in Appendix I.

3.4.2. Transmit Procedure

The transmit procedure is executed when the peer timer decrements to zero for all modes except client mode with a broadcast server and server mode in all cases. In client mode with a broadcast server messages are never sent. In server mode messages are sent only in response to received messages. This procedure is also called by the receive procedure when an NTP message arrives that does not result in a persistent association.

begin transmit procedure

The following initializes the packet buffer and copies the packet variables. The value *skew* is necessary to account for the skew-error accumulated over the interval since the local clock was last set.

```
pkt.peeraddr ← peer.hostaddr;          /* copy system and peer variables */
pkt.peerport ← peer.hostport;
pkt.hostaddr ← peer.peeraddr;
pkt.hostport ← peer.peerport;
pkt.leap ← sys.leap;
pkt.version ← NTP.VERSION;
pkt.mode ← peer.mode;
pkt.stratum ← sys.stratum;
```

```
pkt.poll ← peer.hostpoll;
pkt.precision ← sys.precision;
pkt.rootdelay ← sys.rootdelay;
if (sys.leap = 11₂ or (sys.clock – sys.reftime) > NTP.MAXAGE)
        skew ← NTP.MAXSKEW;
else
        skew ← φ(sys.clock – sys.reftime);
pkt.rootdispersion ← sys.rootdispersion+ (1 << sys.precision) + skew;
pkt.refid ← sys.refid;
pkt.reftime ← sys.reftime;
```

The transmit timestamp pkt.xmt will be used later in order to validate the reply; thus, implementations must save the exact value transmitted. In addition, the order of copying the timestamps should be designed so that the time to format and copy the data does not degrade accuracy.

```
pkt.org ← peer.org;                    /* copy timestamps */
pkt.rec ← peer.rec;
pkt.xmt ← sys.clock;
peer.xmt ← pkt.xmt;
```

The call to encrypt is implemented only if authentication is implemented. If authentication is enabled, the delay to encrypt the authenticator may degrade accuracy. Therefore, implementations should include a system state variable (not mentioned elsewhere in this specification) which contains an offset calculated to match the expected encryption delay and correct the transmit timestamp as obtained from the local clock.

```
#ifdef (authentication implemented)    /* see Appendix C */
        call encrypt;
        #endef
send packet;
```

The reachability register is shifted one position to the left, with zero replacing the vacated bit. If all bits of this register are zero, the clear procedure is called to purge the clock filter and reselect the synchronization source, if necessary. If the association was not configured by the initialization procedure, the association is demobilized.

```
peer.reach ← peer.reach<< 1;           /* update reachability */
if (peer.reach = 0 and peer.config = 0) begin
        demobilize association;
        exit;
        endif
```

If valid data have been shifted into the filter register at least once during the preceding two poll intervals (low-order bit of peer.reach set to one), the valid data counter is incremented. After eight such valid intervals the poll interval is incremented. Otherwise, the valid data counter and poll interval are both decremented and the clock-filter procedure called with zero values for offset and

peer.mode → *mode* ↓	sym act 1	sym pas 2	client 3	server 4	bcst 5
sym active	recv	pkt	recv[2]	xmit[2]	xmit[1,2]
sym passive	recv	error	recv[2]	error	error
client	xmit[2]	xmit[2]	error	xmit	xmit[1]
server	recv[2]	error	recv	error	error
broadcast	recv[1,2]	error	recv[1]	error	error

Notes:

1. A broadcast server responds directly to the client with pkt.org and pkt.rec containing correct values. At other times the server simply broadcasts the local time with pkt.org and pkt.rec set to zero.

2. Ordinarily, these mode combinations would not be used; however, within the limits of the specification, they would result in correct time.

Table 5. Modes and Actions

delay and NTP.MAXDISPERSE for dispersion. The clock-select procedure is called to reselect the synchronization source, if necessary.

```
if (peer.reach & 6 ≠ 0)                    /* test two low-order bits (shifted) */
        if (peer.valid < NTP.SHIFT)        /* valid data received */
                peer.valid ← peer.valid + 1;
                else peer.hostpoll ← peer.hostpoll + 1;
else begin
        peer.valid ← peer.valid – 1;       /* nothing heard */
        peer.hostpoll ← peer.hostpoll – 1);
        call clock-filter(0, 0, NTP.MAXDISPERSE);
        call clock-select;                 /* select clock source */
        endif
call poll-update;
end transmit procedure;
```

3.4.3. Receive Procedure

The receive procedure is executed upon arrival of an NTP message. It validates the message, interprets the various modes and calls other procedures to filter the data and select the synchronization source. If the version number in the packet does not match the current version, the message may be discarded; however, exceptions may be advised on a case-by-case basis at times when the version is changed. If the NTP control messages described in Appendix B are implemented and the packet mode is 6 (control), the control-message procedure is called. The source and destination Internet addresses and ports in the IP and UDP headers are matched to the correct peer. If there is no match a new instantiation of the protocol machine is created and the association mobilized.

```
begin receive procedure
        if (pkt.version ≠ NTP.VERSION) exit;
        #ifdef (control messages implemented)
                if (pkt.mode = 6) call control-message;
                #endef
        for (all associations)                  /* access control goes here */
                match addresses and ports to associations;
        if (no matching association)
                call receive-instantiation procedure;       /* create association */
```

The call to decrypt is implemented only if authentication is implemented.

```
        #ifdef (authentication implemented)     /* see Appendix C */
                call decrypt;
                #endef
```

If the packet mode is nonzero, this becomes the value of *mode* used in the following step; otherwise, the peer is an old NTP version and *mode* is determined from the port numbers as described in Section 3.3.

```
        if (pkt.mode = 0)                       /* for compatibility with old versions */
                mode ← (see Section 3.3);
        else
                mode ← pkt.mode;
```

Table 5 shows for each combination of peer.mode and *mode* the resulting case labels.

```
        case (mode, peer.hostmode)              /* see Table 5 */
```

If *error* the packet is simply ignored and the association demobilized, if not previously configured.

```
error:  if (peer.config = 0) demobilize association;       /* see no evil */
        break;
```

If *recv* the packet is processed and the association marked reachable if tests five through eight (valid header) enumerated in the packet procedure succeed. If, in addition, tests one through four succeed (valid data), the clock-update procedure is called to update the local clock. Otherwise, if the association was not previously configured, it is demobilized.

```
recv:   call packet;                            /* process packet */
        if (valid header) begin                 /* if valid header, update local clock */
                peer.reach ← peer.reach | 1;
                if (valid data) call clock-update;
                endif
        else
                if (peer.config = 0) demobilize association;
        break;
```

If *xmit* the packet is processed and an immediate reply is sent. The association is then demobilized if not previously configured.

```
xmit:       call packet;                        /* process packet */
            peer.hostpoll ← peer.peerpoll;      /* send immediate reply */
            call poll-update;
            call transmit;
            if (peer.config = 0) demobilize association;
            break;
```

If *pkt* the packet is processed and the association marked reachable if tests five through eight (valid header) enumerated in the packet procedure succeed. If, in addition, tests one through four succeed (valid data), the clock-update procedure is called to update the local clock. Otherwise, if the association was not previously configured, an immediate reply is sent and the association demobilized.

```
pkt:        call packet;                        /* process packet */
            if (valid header) begin             /* if valid header, update local clock */
                  peer.reach ← peer.reach | 1;
                  if (valid data) call clock-update;
                  endif
            else if (peer.config = 0) begin
                  peer.hostpoll ← peer.peerpoll;      /* send immediate reply */
                  call poll-update;
                  call transmit;
                  demobilize association;
                  endif
            endcase
      end receive procedure;
```

3.4.4. Packet Procedure

The packet procedure checks the message validity, computes delay/offset samples and calls other procedures to filter the data and select the synchronization source. Test 1 requires the transmit timestamp not match the last one received from the same peer; otherwise, the message might be an old duplicate. Test 2 requires the originate timestamp match the last one sent to the same peer; otherwise, the message might be out of order, bogus or worse. In case of broadcast mode (5) the apparent roundtrip delay will be zero and the full accuracy of the time-transfer operation may not be achievable. However, the accuracy achieved may be adequate for most purposes. The poll-update procedure is called with argument peer.hostpoll (peer.peerpoll may have changed).

```
begin packet procedure
      peer.rec ← sys.clock;                 /* capture receive timestamp */
      if (pkt.mode ≠ 5) begin
            test1 ← (pkt.xmt ≠ peer.org);   /* test 1 */
```

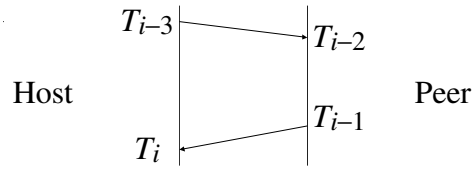

Figure 2. Calculating Delay and Offset

```
        test2 ← (pkt.org = peer.xmt);       /* test 2 */
        endif
    else begin
        pkt.org ← peer.rec;                 /* fudge missing timestamps */
        pkt.rec ← pkt.xmt;
        test1 ← true;                       /* fake tests */
        test2 ← true;
        endif
    peer.org ← pkt.xmt;                     /* update originate timestamp */
    peer.peerpoll ← pkt.poll;               /* adjust poll interval */
    call poll-update(peer.hostpoll);
```

Test 3 requires that both the originate and receive timestamps are nonzero. If either of the timestamps are zero, the association has not synchronized or has lost reachability in one or both directions.

```
    test3 ← (pkt.org ≠ 0 and pkt.rec ≠ 0);      /* test 3 */
```

The roundtrip delay and clock offset relative to the peer are calculated as follows. Number the times of sending and receiving NTP messages as shown in Figure 2 and let i be an even integer. Then T_{i-3}, T_{i-2}, T_{i-1} and T_i are the contents of the pkt.org, pkt.rec, pkt.xmt and peer.rec variables, respectively. The clock offset θ, roundtrip delay δ and dispersion ε of the host relative to the peer is:

$$\delta = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) ,$$
$$\theta = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2} ,$$
$$\varepsilon = (1 << \text{sys.precision}) + \varphi(T_i - T_{i-3}) ,$$

where, as before, $\varphi = \frac{\text{NTP.MAXSKEW}}{\text{NTP.MAXAGE}}$. The quantity ε represents the maximum error or dispersion due to measurement error at the host and local-clock skew accumulation over the interval since the last message was transmitted to the peer. Subsequently, the dispersion will be updated by the clock-filter procedure.

The above method amounts to a continuously sampled, returnable-time system, which is used in some digital telephone networks [BEL86]. Among the advantages are that the order and timing of the messages are unimportant and that reliable delivery is not required. Obviously, the accuracies

achievable depend upon the statistical properties of the outbound and inbound data paths. Further analysis and experimental results bearing on this issue can be found in [MIL90] and in Appendix H.

Test 4 requires that the calculated delay be within "reasonable" bounds:

```
test4 ← (|δ| < NTP.MAXDISPERSE and ε < NTP.MAXDISPERSE);  /* test 4 */
```

Test 5 is implemented only if the authentication mechanism described in Appendix C is implemented. It requires either that authentication be explicitly disabled or that the authenticator be present and correct as determined by the decrypt procedure.

```
#ifdef (authentication implemented)        /* test 5 */
        test5 ← ((peer.config = 1 and peer.authenable= 0) or peer.authentic = 1);
        #endef
```

Test 6 requires the peer clock be synchronized and the interval since the peer clock was last updated be positive and less than NTP.MAXAGE. Test 7 insures that the host will not synchronize on a peer with greater stratum. Test 8 requires that the header contains "reasonable" values for the pkt.rootdelay and pkt.rootdispersion fields.

```
test6 ← (pkt.leap ≠ 11₂ and                    /* test 6 */
        pkt.reftime ≤ pkt.xmt < pkt.reftime + NTP.MAXAGE)
test7 ← pkt.stratum ≤ sys.stratum and          /* test 7 */
         pkt.stratum < NTP.MAXSTRATUM;
test8 ← (|pkt.rootdelay| < NTP.MAXDISPERSE and     /* test 8 */
        pkt.rootdispersion < NTP.MAXDISPERSE);
```

With respect to further processing, the packet includes valid (synchronized) data if tests one through four succeed (*test*1 & *test*2 & *test*3 & *test*4 = 1), regardless of the remaining tests. Only packets with valid data can be used to calculate offset, delay and dispersion values. The packet includes a valid header if tests five through eight succeed (*test*5 & *test*6 & *test*7 & *test*8 = 1), regardless of the remaining tests. Only packets with valid headers can be used to determine whether a peer can be selected for synchronization. Note that *test*1 and *test*2 are not used in broadcast mode (forced to **true**), since the originate and receive timestamps are undefined.

The clock-filter procedure is called to produce the delay (peer.delay), offset (peer.offset) and dispersion (peer.dispersion) for the peer. Specification of the clock-filter algorithm is not an integral part of the NTP specification, since there may be other algorithms that work well in practice. However, one found to work well in the Internet environment is described in Section 4 and its use is recommended.

```
if (not valid header) exit;
peer.leap ← pkt.leap;                          /* copy packet variables */
peer.stratum ← pkt.stratum;
peer.precision ← pkt.precision;
peer.rootdelay ← pkt.rootdelay;
```

```
peer.rootdispersion ← pkt.rootdispersion;
peer.refid ← pkt.refid;
peer.reftime ← pkt.reftime;
if (valid data) call clock-filter(θ, δ, ε);      /* process sample */
end packet procedure;
```

3.4.5. Clock-Update Procedure

The clock-update procedure is called from the receive procedure when valid clock offset, delay and dispersion data have been determined by the clock-filter procedure for the current *peer*. The result of the clock-selection and clock-combining procedures is the final clock correction Θ, which is used by the local-clock procedure to update the local clock. If no candidates survive these procedures, the clock-update procedure exits without doing anything further.

```
begin clock-update procedure
    call clock-select;                          /* select clock source */
    if (sys.peer ≠ peer) exit;
```

It may happen that the local clock may be reset, rather than slewed to its final value. In this case the clear procedure is called for every peer to purge the clock filter, reset the poll interval and reselect the synchronization source, if necessary. Note that the local-clock procedure sets the leap bits sys.leap to "unsynchronized" 11_2 in this case, so that no other peer will attempt to synchronize to the host until the host once again selects a peer for synchronization.

The distance procedure calculates the root delay Δ, root dispersion E and root synchronization distance Λ via the peer to the root of the synchronization subnet. The host will not synchronize to the selected peer if the distance is greater than NTP.MAXDISTANCE. The reason for the minimum clamp at NTP.MINDISPERSE is to discourage subnet route flaps that can happen with Bellman-Ford algorithms and small roundtrip delays.

```
    Λ andistance(peer);                         /* update system variables */
    if (Λ ≥ NTP.MAXDISTANCE) exit;
    sys.leap ← peer.leap;
    sys.stratum ← peer.stratum + 1;
    sys.refid ← peer.peeraddr;
    call local-clock;
    if (local clock reset) begin                /* if reset, clear state variables */
        sys.leap ← 11₂;
        for (all peers) call clear;
        endif
    else begin
        sys.peer ← peer;                        /* if not, adjust local clock */
        sys.rootdelay ← Δ;
        sys.rootdispersion ← E + max(εξ + |Θ|, NTP.MINDISPERSE);
        endif
```

```
    sys.reftime ← sys.clock;
    end clock-update procedure;
```

In some system configurations a precise source of timing information is available in the form of a train of timing pulses spaced at one-second intervals. Usually, this is in addition to a source of timecode information, such as a radio clock or even NTP itself, to number the seconds, minutes, hours and days. In these configurations the system variables are set to refer to the source from which the pulses are derived. For those configurations which support a primary reference source, such as a radio clock or calibrated atomic clock, the stratum is set at one as long as this is the actual synchronization source and whether or not the primary-clock procedure is used.

Specification of the clock-selection and local-clock algorithms is not an integral part of the NTP specification, since there may be other algorithms which provide equivalent performance. However, a clock-selection algorithm found to work well in the Internet environment is described in Section 4, while a local-clock algorithm is described in Section 5 and their use is recommended. The clock-selection algorithm described in Section 4 usually picks the peer at the lowest stratum and minimum synchronization distance among all those available, unless that peer appears to be a falseticker. The result is that the algorithms all work to build a minimum-weight spanning tree relative to the primary reference time servers and thus a hierarchical-master-slave synchronization subnet.

3.4.6. Primary-Clock Procedure

When a primary reference source such as a radio clock is connected to the host, it is convenient to incorporate its information into the data base as if the clock were represented as an ordinary peer. In the primary-clock procedure the clock is polled once a minute or so and the returned timecode used to produce a new update for the local clock. When peer.timer decrements to zero for a primary clock peer, the transmit procedure is not called; rather, the radio clock is polled, usually using an ASCII string specified for this purpose. When a valid timecode is received from the radio clock, it is converted to NTP timestamp format and the peer variables updated. The value of peer.leap is set depending on the status of the leap-warning bit in the timecode, if available, or manually by the operator. The value for peer.peeraddr, which will become the value of sys.refid when the clock-update procedure is called, is set to an ASCII string describing the clock type (see Appendix A).

```
begin primary-clock-update procedure
    peer.leap ← from radio or operator;       /* copy variables */
    peer.peeraddr ← ASCII identifier;
    peer.rec ← radio timestamp;
    peer.reach ← 1;
    call clock-filter(sys.clock – peer.rec, 0, 1 << peer.precision);  /* process sample */
    call clock-update;                        /* update local clock */
    end primary-clock-update procedure;
```

3.4.7. Initialization Procedures

The initialization procedures are used to set up and initialize the system, its peers and associations.

3.4.7.1. Initialization Procedure

The initialization procedure is called upon reboot or restart of the NTP daemon. The local clock is presumably undefined at reboot; however, in some equipment an estimate is available from the reboot environment, such as a battery-backed clock/calendar. The precision variable is determined by the intrinsic architecture of the local hardware clock. The authentication variables are used only if the authentication mechanism described in Appendix C is implemented. The values of these variables are determined using procedures beyond the scope of NTP itself.

```
begin initialization procedure
        #ifdef (authentication implemented)       / * see Appendix C */
                sys.keys ← as required;
                #endef;
        sys.leap ← 11₂;                           /* copy variables */
        sys.stratum ← 0 (undefined);
        sys.precision ← host precision;
        sys.rootdelay ← 0 (undefined);
        sys.rootdispersion ← 0 (undefined);
        sys.refid ← 0 (undefined);
        sys.reftime ← 0 (undefined);
        sys.clock ← external reference;
        sys.peer ← NULL;
        sys.poll ← NTP.MINPOLL;
        for (all configured peers)                /* create configured associations */
                call initialization-instantiation procedure;
        end initialization procedure;
```

3.4.7.2. Initialization-Instantiation Procedure

This implementation-specific procedure is called from the initialization procedure to define an association. The addresses and modes of the peers are determined using information read during the reboot procedure or as the result of operator commands. The authentication variables are used only if the authentication mechanism described in Appendix C is implemented. The values of these variables are determined using procedures beyond the scope of NTP itself. With the authentication bits set as suggested, only properly authenticated peers can become the synchronization source.

```
begin initialization-instantiation procedure
        peer.config ← 1;
        #ifdef (authentication implemented)       /* see Appendix C */
                peer.authenable ← 1 (suggested);
                peer.authentic ← 0;
                peer.hostkeyid ← as required;
                peer.peerkeyid ← 0;
                #endef;
        peer.peeraddr ← peer IP address;          /* copy variables */
```

```
        peer.peerport ← NTP.PORT;
        peer.hostaddr ← host IP address;
        peer.hostport ← NTP.PORT;
        peer.mode ← host mode;
        peer.peerpoll ← 0 (undefined);
        peer.timer ← 0;
        peer.delay ← 0 (undefined);
        peer.offset ← 0 (undefined);
        call clear;                                  /* initialize association */
        end initialization-instantiation procedure;
```

3.4.7.3. Receive-Instantiation Procedure

The receive-instantiation procedure is called from the receive procedure when a new peer is discovered. It initializes the peer variables and mobilizes the association. If the message is from a peer operating in client mode (3), the host mode is set to server mode (4); otherwise, it is set to symmetric passive mode (2). The authentication variables are used only if the authentication mechanism described in Appendix C is implemented. If implemented, only properly authenticated non-configured peers can become the synchronization source.

```
begin receive-instantiation procedure
        #ifdef (authentication implemented)      /* see Appendix C */
                peer.authenable ← 0;
                peer.authentic ← 0;
                peer.hostkeyid ← as required;
                peer.peerkeyid ← 0;
                #endef
        peer.config ← 0;                         /* copy variables */
        peer.peeraddr ← pkt.peeraddr;
        peer.peerport ← pkt.peerport;
        peer.hostaddr ← pkt.hostaddr;
        peer.hostport ← pkt.hostport;
        if (pkt.mode = 3)                        /* determine mode */
                peer.mode ← 4;
                else
                peer.mode ← 2;
        peer.peerpoll ← 0 (undefined);
        peer.timer ← 0;
        peer.delay ← 0 (undefined);
        peer.offset ← 0 (undefined);
        call clear;                              /* initialize association */
        end receive-instantiation procedure;
```

3.4.7.4. Primary Clock-Instantiation Procedure

This procedure is called from the initialization procedure in order to set up the state variables for the primary clock. The value for peer.precision is determined from the radio clock specification and hardware interface. The value for peer.rootdispersion is nominally ten times the inherent maximum error of the radio clock; for instance, 10 μs for a calibrated atomic clock, 10 ms for a WWVB or GOES radio clock and 100 ms for a less accurate WWV radio clock.

```
begin clock-instantiation procedure
        peer.config ← 1;                        /* copy variables */
        peer.peeraddr ← 0 undefined;
        peer.peerport ← 0 (not used);
        peer.hostaddr ← 0 (not used);
        peer.hostport ← 0 (not used);
        peer.leap ← 11₂;
        peer.mode ← 0 (not used);
        peer.stratum ← 0;
        peer.peerpoll ← 0 (undefined);
        peer.precision ← clock precision;
        peer.rootdelay ← 0;
        peer.rootdispersion ← clock dispersion;
        peer.refid ← 0 (not used);
        peer.reftime ← 0 (undefined);
        peer.timer ← 0;
        peer.delay ← 0 (undefined);
        peer.offset ← 0 (undefined);
        call clear;                             /* initialize association */
        end clock-instantiation procedure;
```

In some configurations involving a calibrated atomic clock or LORAN-C receiver, the primary reference source may provide only a seconds pulse, but lack a complete timecode from which the numbering of the seconds, etc., can be derived. In these configurations seconds numbering can be derived from other sources, such as a radio clock or even other NTP peers. In these configurations the primary clock variables should reflect the primary reference source, not the seconds-numbering source; however, if the seconds-numbering source fails or is known to be operating incorrectly, updates from the primary reference source should be suppressed as if it had failed.

3.4.8. Clear Procedure

The clear procedure is called when some event occurs that results in a significant change in reachability state or potential disruption of the local clock.

```
begin clear procedure
        peer.org ← 0 (undefined);               /* mark timestamps undefined */
        peer.rec ← 0 (undefined);
```

```
        peer.xmt ← 0 (undefined);
        peer.reach ← 0;                         /* reset state variables */
        peer.filter ← [0, ,0, NTP.MAXDISPERSE];  /* all stages */
        peer.valid ← 0;
        peer.dispersion ← NTP.MAXDISPERSE;
        peer.hostpoll ← NTP.MINPOLL;            /* reset poll interval */
        call poll-update;
        call clock-select;                      /* select clock source */
        end clear procedure;
```

3.4.9. Poll-Update Procedure

The poll-update procedure is called when a significant event occurs that may result in a change of the poll interval or peer timer. It checks the values of the host poll interval (peer.hostpoll) and peer poll interval (peer.peerpoll) and clamps each within the valid range. If the peer is selected for synchronization, the value is further clamped as a function of the computed compliance (see Section 5).

```
begin poll-update procedure
        temp ← peer.hostpoll;                   /* determine host poll interval */
        if (peer = sys.peer)
                temp ← min(temp, sys.poll, NTP.MAXPOLL);
        else
                temp ← min(temp, NTP.MAXPOLL);
        peer.hostpoll ← max(temp, NTP.MINPOLL);
        temp ← 1 << min(peer.hostpoll, max(peer.peerpoll, NTP.MINPOLL));
```

If the poll interval is unchanged and the peer timer is zero, the timer is simply reset. If the poll interval is changed and the new timer value is greater than the present value, no additional action is necessary; otherwise, the peer timer must be reduced. When the peer timer must be reduced it is important to discourage tendencies to synchronize transmissions between the peers. A prudent precaution is to randomize the first transmission after the timer is reduced, for instance by the sneaky technique illustrated.

```
        if (peer.timer = 0)                     /* reset peer timer */
                peer.timer ← temp;
        else if (peer.timer > temp)
                peer.timer ← (sys.clock & (temp – 1)) + 1;
        end poll-update procedure;
```

3.5. Synchronization Distance Procedure

The distance procedure calculates the synchronization distance from the peer variables for the peer *peer*.

begin distance(*peer*) procedure;

$\Delta \leftarrow$ peer.rootdelay + |peer.delay|;

$E \leftarrow$ peer.rootdispersion + peer.dispersion + φ(sys.clock – peer.update);

$\Lambda \leftarrow E + \frac{|\Delta|}{2}$;

end distance procedure;

Note that, while Δ may be negative in some cases, both E and Λ are always positive.

3.6. Access Control Issues

The NTP design is such that accidental or malicious data modification (tampering) or destruction (jamming) at a time server should not in general result in timekeeping errors elsewhere in the synchronization subnet. However, the success of this approach depends on redundant time servers and diverse network paths, together with the assumption that tampering or jamming will not occur at many time servers throughout the synchronization subnet at the same time. In principle, the subnet vulnerability can be engineered through the selection of time servers known to be trusted and allowing only those time servers to become the synchronization source. The authentication procedures described in Appendix C represent one mechanism to enforce this; however, the encryption algorithms can be quite CPU-intensive and can seriously degrade accuracy, unless precautions such as mentioned in the description of the transmit procedure are taken.

While not a required feature of NTP itself, some implementations may include an access-control feature that prevents unauthorized access and controls which peers are allowed to update the local clock. For this purpose it is useful to distinguish between three categories of access: those that are preauthorized as trusted, preauthorized as friendly and all other (non-preauthorized) accesses. Presumably, preauthorization is accomplished by entries in the configuration file or some kind of ticket-management system such as Kerberos [STE88]. In this model only trusted accesses can result in the peer becoming the synchronization source. While friendly accesses cannot result in the peer becoming the synchronization source, NTP messages and timestamps are returned as specified.

It does not seem useful to maintain a secret clock, as would result from restricting non-preauthorized accesses, unless the intent is to hide the existence of the time server itself. Well-behaved Internet hosts are expected to return an ICMP service-unavailable error message if a service is not implemented or resources are not available; however, in the case of NTP the resources required are minimal, so there is little need to restrict requests intended only to read the clock. A simple but effective access-control mechanism is then to consider all associations preconfigured in a symmetric mode or client mode (modes 1, 2 and 3) as trusted and all other associations, preconfigured or not, as friendly.

If a more comprehensive trust model is required, the design can be based on an access-control list with each entry consisting of a 32-bit Internet address, 32-bit mask and three-bit mode. If the logical AND of the source address (pkt.peeraddr) and the mask in an entry matches the corresponding address in the entry and the mode (pkt.mode) matches the mode in the entry, the access is allowed; otherwise an ICMP error message is returned to the requestor. Through appropriate choice of mask,

it is possible to restrict requests by mode to individual addresses, a particular subnet or net addresses, or have no restriction at all. The access-control list would then serve as a filter controlling which peers could create associations.

4. Filtering and Selection Algorithms

A most important factor affecting the accuracy and reliability of time distribution is the complex of algorithms used to reduce the effect of statistical errors and falsetickers due to failure of various subnet components, reference sources or propagation media. The algorithms suggested in this section were developed and refined over several years of operation in the Internet under widely varying topologies, speeds and traffic regimes. While these algorithms are believed the best available at the present time, they are not an integral part of the NTP specification, since other algorithms with similar or superior performance may be devised in future.

However, it is important to observe that not all time servers or clients in an NTP synchronization subnet must implement these algorithms. For instance, simple workstations may dispense with one or both of them in the interests of simplicity if accuracy and reliability requirements justify. Nevertheless, it would be expected that an NTP server providing synchronization to a sizable community, such as a university campus or research laboratory, would be expected to implement these algorithms or others proved to have equivalent functionality. A comprehensive discussion of the design principles and performance is given in [MIL91a].

In order for the NTP filter and selection algorithms to operate effectively, it is useful to have a measure of recent sample variance recorded for each peer. The measure adopted is based on first-order differences, which are easy to compute and effective for the purposes intended. There are two measures, one called the *filter dispersion* ε_σ and the other the *select dispersion* ε_ξ. Both are computed as the weighted sum of the clock offsets in a temporary list sorted by synchronization distance. If θ_i $(0 \le i < n)$ is the offset of the ith entry, then the sample difference ε_{ij} of the ith entry relative to the jth entry is defined $\varepsilon_{ij} = |\theta_i - \theta_j|$. The dispersion relative to the jth entry is defined ε_j and computed as the weighted sum

$$\varepsilon_j = \sum_{i=0}^{n-1} \varepsilon_{ij}\, w^{\,i+1} ,$$

where w is a weighting factor chosen to control the influence of synchronization distance in the dispersion budget. In the NTP algorithms w is chosen less than ½: w = NTP.FILTER for filter dispersion and w = NTP.SELECT for select dispersion. The (absolute) dispersion ε_σ and ε_ξ as used in the NTP algorithms are defined relative to the 0th entry ε_0.

There are two procedures described in the following, the clock-filter procedure, which is used to select the best offset samples from a given clock, and the clock-selection procedure, which is used to select the best clock among a hierarchical set of clocks.

4.1. Clock-Filter Procedure

The clock-filter procedure is executed upon arrival of an NTP message or other event that results in new data samples. It takes arguments of the form (θ, δ, ε), where θ is a sample clock offset measurement and δ and ε are the associated roundtrip delay and dispersion. It determines the filtered clock offset (peer.offset), roundtrip delay (peer.delay) and dispersion (peer.dispersion). It also updates the dispersion of samples already recorded and saves the current time (peer.update).

The basis of the clock-filter procedure is the filter shift register (peer.filter), which consists of NTP.SHIFT stages, each stage containing a 3-tuple [θ_i, δ_i, ε_i], with indices numbered from zero on the left. The filter is initialized with the value [0, 0, NTP.MAXDISPERSE] in all stages by the clear procedure. New data samples are shifted into the filter at the left end, causing first NULLs then old samples to fall off the right end. The packet procedure provides samples of the form (θ, δ, ε) as new updates arrive, while the transmit procedure provides samples of the form [0, 0, NTP.MAXDISPERSE] when two poll intervals elapse without a fresh update. While the same symbols (θ, δ, ε) are used here for the arguments, clock-filter contents and peer variables, the meaning will be clear from context. The following pseudo-code describes this procedure.

begin clock-filter procedure (θ, δ, ε)

The dispersion ε_i for all valid samples in the filter register must be updated to account for the skew-error accumulation since the last update. These samples are also inserted on a temporary list with entry format [*distance*,*index*]. The samples in the register are shifted right one stage, with the overflow sample discarded and the new sample inserted at the leftmost stage. The temporary list is then sorted by increasing *distance*. If no samples remain in the list, the procedure exits without updating the peer variables.

```
for (i from NTP.SIZE – 1 to 1) begin      /* update dispersion */
    [θi, δi, εi] ← [θi–1, δi–1, εi–1];    /* shift stage right */
    εi = εi + φτ;
    add [λi ≡ εi + |δi|/2, i] to temporary list;
    endfor;
[θ0, δ0, ε0] ← [θ, δ, ε];                 /* insert new sample */
add [λ ≡ ε + |δ|/2, 0] to temporary list;
peer.update ← sys.clock;                  /* reset base time */
sort temporary list by increasing [distance||index];
```

where [*distance* ||*index*] represents the concatenation of the *distance* and *index* fields and *distance* is the high-order field. The filter dispersion ε_σ is computed and included in the peer dispersion. Note that for this purpose the temporary list is already sorted.

```
εσ ← 0;
for (i from NTP.SHIFT–1 to 0)             /* compute filter dispersion */
```

```
        if (peer.dispersion_index[i] ≥ NTP.MAXDISPERSE or
                |θ_i − θ_0| > NTP.MAXDISPERSE)
                ε_σ ← (ε_σ + NTP.MAXDISPERSE) × NTP.FILTER;
        else
                ε_σ ← (ε_σ + |θ_i − θ_0|) × NTP.FILTER;
```

The peer offset θ_0, delay δ_0 and dispersion ε_0 are chosen as the values corresponding to the minimum-distance sample; in other words, the sample corresponding to the first entry on the temporary list, here represented as the 0th subscript.

```
        peer.offset ← θ_0;                      /* update peer variables */
        peer.delay ← δ_0;
        peer.dispersion ← min(ε_0 + ε_σ, NTP.MAXDISPERSE);
        end clock-filter procedure
```

The peer.offset and peer.delay variables represent the clock offset and roundtrip delay of the local clock relative to the peer clock. Both of these are precision quantities and can usually be averaged over long intervals in order to improve accuracy and stability without bias accumulation (see Appendix H). The peer.dispersion variable represents the maximum error due to measurement error, skew-error accumulation and sample variance. All three variables are used in the clock-selection and clock-combining procedures to select the peer clock(s) used for synchronization and to maximize the accuracy and stability of the indications.

4.2. Clock-Selection Procedure

The clock-selection procedure uses the peer variables Θ, Δ, E and τ and is called when these variables change or when the reachability status changes. It consists of two algorithms, the intersection algorithm and the clustering algorithm. The intersection algorithm constructs a list of candidate peers eligible to become the synchronization source, computes a confidence interval for each and casts out falsetickers using a technique adapted from Marzullo and Owicki [MAR85]. The clustering algorithm sorts the list of surviving candidates in order of stratum and synchronization distance and repeatedly casts out outlyers on the basis of select dispersion until only the most accurate, precise and stable survivors are left. A bit is set for each survivor to indicate the outcome of the selection process. The system variable sys.peer is set as a pointer to the most likely survivor, if there is one, or to the NULL value if not.

4.2.1. Intersection Algorithm

```
        begin clock-selection procedure
```

Each peer is examined in turn and added to an endpoint list only if it passes several sanity checks designed to avoid loops and use of exceptionally noisy data. If no peers survive the sanity checks, the procedure exits without finding a synchronization source. For each of *m* survivors three entries of the form [*endpoint*, *type*] are added to the endpoint list: $[\Theta - \Lambda, -1]$, $[\Theta, 0]$ and $[\Theta + \Lambda, 1]$. There will be $3m$ entries on the list, which is then sorted by increasing *endpoint*.

```
m ← 0;
for (each peer)                          /* calling all peers */
        if (peer.reach ≠ 0 and peer.dispersion < NTP.MAXDISPERSE and
                not (peer.stratum > 1 and peer.refid = peer.hostaddr)) begin
                Λ andistance(peer);              /* make list entry */
                add [Θ − Λ, −1] to endpoint list;
                add [Θ, 0] to endpoint list;
                add [Θ + Λ, 1] to endpoint list;
                m ← m + 1;
                endif
        endfor
if (m = 0) begin                         /* skedaddle if no candidates */
        sys.peer ← NULL;
        sys.stratum ← 0 (undefined);
        exit;
        endif
sort endpoint list by increasing endpoint||type;
```

The following algorithm is adapted from DTS [DEC89] and is designed to produce the largest single intersection containing only truechimers. The algorithm begins by initializing a value *f* and counters *i* and *c* to zero. Then, starting from the lowest endpoint of the sorted endpoint list, for each entry [*endpoint*, *type*] the value of *type* is subtracted from the counter *i*, which is the number of intersections. If *type* is zero, increment the value of *c*, which is the number of falsetickers (see Appendix H). If $i \ge m - f$ for some entry, *endpoint* of that entry becomes the lower endpoint of the intersection; otherwise, *f* is increased by one and the above procedure is repeated. Without resetting *f* or *c*, a similar procedure is used to find the upper endpoint, except that the value of *type* is added to the counter.. If after both endpoints have been determined $c \le f$, the procedure continues having found $m - f$ truechimers; otherwise, *f* is increased by one and the entire procedure is repeated.

```
for (f from 0 to f ≥ m/2) begin          /* calling all truechimers */
        c ← 0;
        i ← 0;
        for (each [endpoint, type] from lowest) begin    /* find low endpoint */
                i ← i − type;
                low ← endpoint;
                if (i ≥ m − f) break;
                if (type = 0) c ← c + 1;
                endfor;
        i ← 0;

        for (each [endpoint, type] from highest) begin   /* find high endpoint */
                i ← i + type;
                high ← endpoint;
```

```
                if (i ≥ m – f) break;
                if (type = 0) c ← c + 1;
                endfor;
        if (c ≤ f) break;                   /* continue until all falsetickers found */
        endfor;
if (low > high) begin                       /* quit if no intersection found */
        sys.peer ← NULL;
        exit;
        endif;
```

Note that processing continues past this point only if there are more than $\frac{m}{2}$ intersections. However, it is possible, but not highly likely, that there may be fewer than $\frac{m}{2}$ truechimers remaining in the intersection.

4.2.2. Clustering Algorithm

In the original DTS algorithm the clock-selection procedure exits at this point with the presumed correct time set midway in the computed intersection [*low*, *high*]. However, this can lead to a considerable loss in accuracy and stability, since the individual peer statistics are lost. Therefore, in NTP the candidates that survived the preceding steps are processed further. The candidate list is rebuilt with entries of the form [*distance*, *index*], where *distance* is computed from the (scaled) peer stratum and synchronization distance Λ. The scaling factor provides a mechanism to weight the combination of stratum and distance. Ordinarily, the stratum will dominate, unless one or more of the survivors has an exceptionally high distance. The list is then sorted by increasing *distance*.

```
m ← 0;
for (each peer) begin                       /* calling all peers */
        if (low ≤ θ ≤ high) begin
                Λ ← distance(peer);         /* make list entry */
                dist ← peer.stratum × NTP.MAXDISPERSE + Λ
                add [dist, peer] to candidate list;
                m ← m + 1;
                endif;
        endfor;
sort candidate list by increasing dist;
```

The next steps are designed to cast out outlyers which exhibit significant dispersions relative to the other members of the candidate list while minimizing wander, especially on high-speed LANs with many time servers. Wander causes needless network overhead, since the poll interval is clamped at sys.poll as each new peer is selected for synchronization and only slowly increases when the peer is no longer selected. It has been the practical experience that the number of candidates surviving to this point can become quite large and can result in significant processor cycles without materially

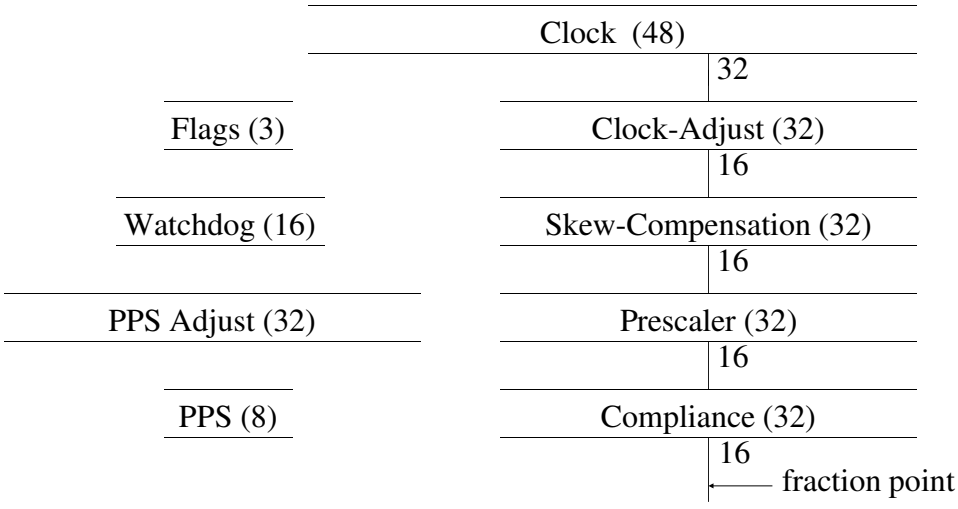

Figure 3. Clock Registers

enhancing stability and accuracy. Accordingly, the candidate list is truncated at NTP.MAXCLOCK entries.

Note $\varepsilon\xi_i$ is the select (sample) dispersion relative to the *i*th peer represented on the candidate list, which can be calculated in a manner similar to the filter dispersion described previously. The E_j is the dispersion of the *j*th peer represented on the list and includes components due to measurement error, skew-error accumulation and filter dispersion. If the maximum $\varepsilon\xi_i$ is greater than the minimum E_j and the number of survivors is greater than NTP.MINCLOCK, the *i*th peer is discarded from the list and the procedure is repeated. If the current synchronization source is one of the survivors and there is no other survivor of lower stratum, then the procedure exits without doing anything further. Otherwise, the synchronization source is set to the first survivor on the candidate list. In the following *i*, *j*, *k*, *l* are peer indices, with *k* the index of the current synchronization source (NULL if none) and *l* the index of the first survivor on the candidate list.

```
while begin
        for (each survivor [distance, index]) begin       /* compute dispersions */
                find index i for max εξi;
                find index j for min Ej;
                endfor
        if (εξj ≤ Ej or m ≤ NTP.MINCLOCK) break;
        peer.survivor[i] ← 0 ;                  /* discard ith peer */
        if (i = k) sys.peer ← NULL;
        delete the ith peer from the candidate list;
        m ← m – 1;
        endwhile
if (peer.survivor[k] = 0 or peer.stratum[k] > peer.stratum[l]) begin
        sys.peer ← l;                           /* new clock source */
```

Parameter	Name	Crystal	Mains
Adjustment Interval	CLOCK.ADJ	4 sec	1 sec
PPS Timeout	CLOCK.PPS	60 sec	60 sec
Step Timeout	CLOCK.MINSTEP	900 sec	900 sec
Maximum Aperture	CLOCK.MAX	± 128 ms	± 512 ms
Frequency Weight	CLOCK.FREQ	16	16
Phase Weight	CLOCK.PHASE	8	9
Compliance Weight	CLOCK.WEIGHT	13	13
Compliance Maximum	CLOCK.COMP	4	4
Compliance Multiplier	CLOCK.MULT	4	4

Table 6. Clock Parameters

```
        call poll-update;
        endif
    end clock-select procedure;
```

The algorithm is designed to favor those peers near the head of the candidate list, which are at the lowest stratum and distance and presumably can provide the most accurate and stable time. With proper selection of weight factor *v* (also called NTP.SELECT), entries will be trimmed from the tail of the list, unless a few outlyers disagree significantly with respect to the remaining entries, in which case the outlyers are discarded first. The termination condition is designed to avoid needless switching between synchronization sources when not statistically justified, yet maintain a bias toward the low-stratum, low-distance peers.

5. Local Clocks

In order to implement a precise and accurate local clock, the host must be equipped with a hardware clock consisting of an oscillator and interface and capable of the required precision and stability. A logical clock is then constructed using these components plus software components that adjust the apparent time and frequency in response to periodic updates computed by NTP or some other time-synchronization protocol such as Hellospeak [MIL83b] or the Unix 4.3bsd TSP [GUS85a]. This section describes the Fuzzball local-clock model and implementation, which includes provisions for precise time and frequency adjustment and can maintain time to within 15 ns and frequency to within 0.3 ms per day. The model is suitable for use with both compensated and uncompensated quartz oscillators and can be adapted to power-frequency oscillators. A summary of the characteristics of these and other types of oscillators can be found in Appendix E, while a comprehensive mathematical analysis of the NTP local-clock model can be found in Appendix G.

It is important to note that the particular implementation described is only one of possibly many implementations that provide equivalent functionality. However, it is equally important to note that the clock model described in Appendix G and which is the basis of the implementation involves a particular kind of control-feedback loop that is potentially unstable if the design rules are broken. The model and parameter described in Appendix G are designed to provide accurate and stable time under typical operating conditions using conventional hardware and in the face of disruptions in

hardware or network connectivity. The parameters have been engineered for reliable operation in a multi-level hierarchical subnet where unstable operation at one level can disrupt possibly many other levels.

5.1. Fuzzball Implementation

The Fuzzball local clock consists of a collection of hardware and software registers, together with a set of algorithms, which implement a logical clock that functions as a disciplined oscillator and synchronizes to an external source. Following is a description of its components and manner of operation. Note that all arithmetic is two's complement integer and all shifts "<<" and ">>" are arithmetic (sign-fill for right shifts and zero-fill for left shifts). Also note that $x << n$ is equivalent to $x >> -n$.

The principal components of the local clock are shown in Figure 3, in which the fraction points shown are relative to whole milliseconds. The 48-bit Clock register and 32-bit Prescaler function as a disciplined oscillator which increments in milliseconds relative to midnight at the fraction point. The 32-bit Clock-Adjust register is used to adjust the oscillator phase in gradual steps to avoid discontinuities in the indicated timescale. Its contents are designated x in the following. The 32-bit Skew-Compensation register is used to trim the oscillator frequency by adding small phase increments at periodic adjustment intervals and can compensate for frequency errors as much as .01% or ±100 ppm. Its contents are designated y in the following. The 16-bit Watchdog counter and 32-bit Compliance register are used to determine validity, as well as establish the PLL bandwidth and poll interval (see Appendix G). The contents of the Compliance register are designated z in the following. The 32-bit PPS-Adjust register is used to hold a precision time adjustment when a source of 1-pps pulses is available, while the 8-bit PPS counter is used to verify presence of these pulses. The two-bit Flags register contains the two leap bits described elsewhere (leap).

All registers except the Prescaler register are ordinarily implemented in memory. In typical clock interface designs such as the DEC KWV11-C, the Prescaler register is implemented as a 16-bit buffered counter driven by a quartz-controlled oscillator at some multiple of 1000 Hz. A counter overflow is signalled by an interrupt, which results in an increment of the Clock register at the bit corresponding to the overflow. The time of day is determined by reading the Prescaler register, which does not disturb the counting process, and adding its value to that of the Clock register with fraction points aligned as shown and with unimplemented low-order bits set to zero. In other interface designs, such as the LSI-11 event-line mechanism, each tick of the clock is signalled by an interrupt at intervals of $16\text{-}^{2}/_{3}$ ms or 20 ms, depending on interface and mains frequency. When this occurs the appropriate increment in fractional milliseconds is added to the Clock register.

The various parameters used are summarized in Table 6, in which certain parameters have been rescaled from those given in Appendix G due to the units here being in milliseconds. When the system is initialized, all registers and counters are cleared and the leap bits set to 11_2 (unsynchronized). At adjustment intervals of CLOCK.ADJ seconds CLOCK.ADJ is subtracted from the PPS counter, but only if the previous contents of the PPS counter are greater than zero. Also, CLOCK.ADJ is added to the Watchdog counter, but the latter is clamped not to exceed

NTP.MAXAGE divided by CLOCK.ADJ (one full day). In addition, if the Watchdog counter reaches this value, the leap bits are set to 11_2 (unsynchronized).

In some system configurations a precise source of timing information is available in the form of a train of timing pulses spaced at one-second intervals. Usually, this is in addition to a source of timecode information, such as a radio clock or even NTP itself, to number the seconds, minutes, hours and days. In typical clock interface designs such as the DEC KWV11-C, a special input is provided which can trigger an interrupt as each pulse is received. When this happens the PPS counter is set to CLOCK.PPS and the current time offset is determined in the usual way. Then, the PPS-Adjust register is set to the time offset scaled to milliseconds. Finally, if the PPS-Adjust register is greater than or equal to 500, 1000 is subtracted from its contents. As described below, the PPS-Adjust register and PPS counters can be used in conjunction with an ordinary timecode to produce an extremely accurate local clock.

5.2. Gradual Phase Adjustments

Left uncorrected, the local clock runs at the offset and frequency resulting from its last update. An update is produced by an event that results in a valid clock selection. It consists of a signed 48-bit integer in whole milliseconds and fraction, with fraction point to the left of bit 32. If the magnitude is greater than the maximum aperture CLOCK.MAX, a step adjustment is required, in which case proceed as described later. Otherwise, a gradual phase adjustment is performed. Normally, the update is computed by the NTP algorithms described previously; however, if the PPS counter is greater than zero, the value of the PPS-Adjust register is used instead. Let u be a 32-bit quantity with bits 0-31 set as bits 16-47 of the update. If some of the low-order bits of the update are unimplemented, they are set as the value of the sign bit. These operations move the fraction point of u to the left of bit 16 and minimize the effects of truncation and roundoff errors. Let b be the number of leading zeros of the absolute value of the Compliance register and let c be the number of leading zeros of the Watchdog counter, both of which are easily computed by compact loops. Then, set b to

$$b = b - 16 + \text{CLOCK.COMP}$$

and clamp it to be not less than zero. This represents the logarithm of the loop time constant. Then, set c to

$$c = 10 - c$$

and clamp it to be not greater than NTP.MAXPOLL – NTP.MINPOLL. This represents the logarithm of the integration interval since the last update. The clamps insure stable operation under typical conditions encountered in the Internet. Then, compute new values for the Clock-Adjust and Skew-Compensation registers

$$x = u >> b\ ,$$
$$y = y + (u >> (b + b - c))\ .$$

Finally, compute the exponential average

$$z = z + (u << (b + \text{CLOCK.MULT}) - z) >> \text{CLOCK.WEIGHT},$$

where the left shift realigns the fraction point for greater precision and ease of computation.

At each adjustment interval the final clock correction consisting of two components is determined. The first (phase) component consists of the quantity

$$x >> \text{CLOCK.PHASE} ,$$

which is then subtracted from the previous contents of the Clock-Adjust register to form the new contents of that register. The second (frequency) component consists of the quantity

$$y >> \text{CLOCK.FREQ} .$$

The sum of the phase and frequency components is the final clock correction, which is then added to the Clock register. FInally, the Watchdog counter is set to zero. Operation continues in this way until a new correction is introduced.

The value of *b* computed above can be used to update the poll interval system variable (sys.poll). This functions as an adaptive parameter that provides a very valuable feature which reduces the polling overhead, especially if the clock-combining algorithm described in Appendix F is used:

$$\text{sys.poll} \leftarrow b + \text{NTP.MINPOLL}.$$

Under conditions when update noise is high or the hardware oscillator frequency is changing relatively rapidly due to environmental conditions, the magnitude of the compliance increases. With the parameters specified, this causes the loop bandwidth (reciprocal of time constant) to increase and the poll interval to decrease, eventually to NTP.MINPOLL seconds. When noise is low and the hardware oscillator very stable, the compliance decreases, which causes the loop bandwidth to decrease and the poll interval to increase, eventually to NTP.MAXPOLL seconds.

The parameters in Table 6 have been selected so that, under good conditions with updates in the order of a few milliseconds, a precision of a millisecond per day (about .01 ppm or 10^{-8}), can be achieved. Care is required in the implementation to insure monotonicity of the Clock register and to preserve the highest precision while minimizing the propagation of roundoff errors. Since all of the multiply/divide operations (except those involved with the 1-pps pulses) computed in real time can be approximated by bitwise-shift operations, it is not necessary to implement a full multiply/divide capability in hardware or software.

In the various implementations of NTP for many Unix-based systems it has been the common experience that the single most important factor affecting local-clock stability is the matching of the phase and frequency coefficients to the particular kernel implementation. It is vital that these coefficients be engineered according to the model values, for otherwise the PLL can fail to track normal oscillator variations and can even become unstable.

5.3. Step Phase Adjustments

When the magnitude of a correction exceeds the maximum aperture CLOCK.MAX, the possibility exists that the clock is so far out of synchronization with the reference source that the best action is

an immediate and wholesale replacement of Clock register contents, rather than in gradual adjustments as described above. However, in cases where the sample variance is extremely high, it is prudent to disbelieve a step change, unless a significant interval has elapsed since the last gradual adjustment. Therefore, if a step change is indicated and the Watchdog counter is less than the preconfigured value CLOCK.MINSTEP, the update is ignored and the local-clock procedure exits. These safeguards are especially useful in those system configurations using a calibrated atomic clock or LORAN-C receiver in conjunction with a separate source of seconds-numbering information, such as a radio clock or NTP peer.

If a step change is indicated the update is added directly to the Clock register and the Clock-Adjust register and Watchdog counter both set to zero, but the other registers are left undisturbed. Since a step change invalidates data currently in the clock filters, the leap bits are set to 11_2 (unsynchronized) and, as described elsewhere, the clear procedure is called to purge the clock filters and state variables for all peers. In practice, the necessity to perform a step change is rare and usually occurs when the local host or reference source is rebooted, for example. This is fortunate, since step changes can result in the local clock apparently running backward, as well as incorrect delay and offset measurements of the synchronization mechanism itself.

Considerable experience with the Internet environment suggests the values of CLOCK.MAX tabulated in Table 6 as appropriate. In practice, these values are exceeded with a single time-server source only under conditions of the most extreme congestion or when multiple failures of nodes or links have occurred. The most common case when the maximum is exceeded is when the time-server source is changed and the time indicated by the new and old sources exceeds the maximum due to systematic errors in the primary reference source or large differences in path delays. It is recommended that implementations include provisions to tailor CLOCK.MAX for specific situations. The amount that CLOCK.MAX can be increased without violating the monotonicity requirement depends on the Clock register increment. For an increment of 10 ms, as used in many workstations, the value shown in Table 6 can be increased by a factor of five.

5.4. Implementation Issues

The basic NTP robustness model is that a host has no other means to verify time other than NTP itself. In some equipment a battery-backed clock/calendar is available for a sanity check. If such a device is available, it should be used only to confirm sanity of the timekeeping system, not as the source of system time. In the common assumption (not always justified) that the clock/calendar is more reliable, but less accurate, than the NTP synchronization subnet, the recommended approach at initialization is to set the Clock register as determined from the clock/calendar and the other registers, counters and flags as described above. On subsequent updates if the time offset is greater than a configuration parameter (e.g., 1000 seconds), then the update should be discarded and an error condition reported. Some implementations periodically record the contents of the Skew-Compensation register in stable storage such as a system file or NVRAM and retrieve this value at initialization. This can significantly reduce the time to converge to the nominal stability and accuracy regime.

Conversion from NTP format to the common date and time formats used by application programs is simplified if the internal local-clock format uses separate date and time variables. The time variable is designed to roll over at 24 hours, give or take a leap second as determined by the leap-indicator bits, with its overflows (underflows) incrementing (decrementing) the date variable. The date and time variables then indicate the number of days and seconds since some previous reference time, but uncorrected for intervening leap seconds.

On the day prior to the insertion of a leap second the leap bits (sys.leap) are set at the primary servers, presumably by manual means. Subsequently, these bits show up at the local host and are passed to the local-clock procedure. This causes the modulus of the time variable, which is the length of the current day, to be increased or decreased by one second as appropriate. Immediately following insertion the leap bits are reset. Additional discussion on this issue can be found in Appendix E.

Lack of a comprehensive mechanism to administer the leap bits in the primary servers is presently an awkward problem better suited to a comprehensive network-management mechanism yet to be developed. As a practical matter and unless specific provisions have been made otherwise, currently manufactured radio clocks have no provisions for leap seconds, either automatic or manual. Thus, when a leap actually occurs, the radio must resynchronize to the broadcast timecode, which may take from a few minutes to some hours. Unless special provisions are made, a primary server might leap to the new timescale, only to be yanked back to the previous timescale when it next synchronizes to the radio. Subsequently, the server will be yanked forward again when the radio itself resynchronizes to the broadcast timecode.

This problem can not be reliably avoided using any selection algorithm, since there will always exist an interval of at least a couple of minutes and possibly as much as some hours when some or all radios will be out of synchronization with the broadcast timecode and only after the majority of them have resynchronized will the subnet settle down. The CLOCK.MINSTEP delay is designed to cope with this problem by forcing a minimum interval since the last gradual adjustment was made before allowing a step change to occur. Therefore, until the radio resynchronizes, it will continue on the old timescale, which is one second off the local clock after the leap and outside the maximum aperture CLOCK.MAX permitted for gradual phase adjustments. When the radio eventually resynchronizes, it will almost certainly come up within the aperture and again become the reference source. Thus, even in the unlikely case when the local clock incorrectly leaps, the server will go no longer than CLOCK.MINSTEP seconds before resynchronizing.

6. Acknowledgments

Many people contributed to the contents of this document, which was thoroughly debated by electronic mail and debugged using two different prototype implementations for the Unix 4.3bsd operating system, one written by Louis Mamakos and Michael Petry of the University of Maryland and the other by Dennis Ferguson of the University of Toronto. Another implementation for the Fuzzball operating system [MIL88b] was written by the author. Many individuals to numerous to mention meticulously tested the several beta-test prototype versions and ruthlessly smoked out the bugs, both in the code and the specification. Especially useful were comments from Dennis Ferguson

and Bill Sommerfeld, as well as discussions with Joe Comuzzi and others at Digital Equipment Corporation.

7. References

[ABA89] Abate, et al. AT&T's new approach to the synchronization of telecommunication networks. *IEEE Communications Magazine* (April 1989), 35-45.

[ALL74a] Allan, D.W., J.H. Shoaf and D. Halford. Statistics of time and frequency data analysis. In: Blair, B.E. (Ed.). *Time and Frequency Theory and Fundamentals*. National Bureau of Standards Monograph 140, U.S. Department of Commerce, 1974, 151-204.

[ALL74b] Allan, D.W., J.E. Gray and H.E. Machlan. The National Bureau of Standards atomic time scale: generation, stability, accuracy and accessibility. In: Blair, B.E. (Ed.). *Time and Frequency Theory and Fundamentals*. National Bureau of Standards Monograph 140, U.S. Department of Commerce, 1974, 205-231.

[BEL86] Bell Communications Research. Digital Synchronization Network Plan. Technical Advisory TA-NPL-000436, 1 November 1986.

[BER87] Bertsekas, D., and R. Gallager. *Data Networks*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[BLA74] Blair, B.E. Time and frequency dissemination: an overview of principles and techniques. In: Blair, B.E. (Ed.). *Time and Frequency Theory and Fundamentals*. National Bureau of Standards Monograph 140, U.S. Department of Commerce, 1974, 233-314.

[BRA80] Braun, W.B. Short term frequency effects in networks of coupled oscillators. *IEEE Trans. Communications COM-28, 8* (August 1980), 1269-1275.

[COL88] Cole, R., and C. Foxcroft. An experiment in clock synchronisation. *The Computer Journal 31, 6* (1988), 496-502.

[DAR81a] Defense Advanced Research Projects Agency. Internet Protocol. DARPA Network Working Group Report RFC-791, USC Information Sciences Institute, September 1981.

[DAR81b] Defense Advanced Research Projects Agency. Internet Control Message Protocol. DARPA Network Working Group Report RFC-792, USC Information Sciences Institute, September 1981.

[DEC89] Digital Time Service Functional Specification Version T.1.0.5. Digital Equipment Corporation, 1989.

[DER90] Dershowitz, N., and E.M. Reingold. Calendrical Calculations. *Software Practice and Experience 20, 9* (September 1990), 899-928.

[FRA82] Frank, R.L. History of LORAN-C. *Navigation 29, 1* (Spring 1982).

[GUS84] Gusella, R., and S. Zatti. TEMPO - A network time controller for a distributed Berkeley UNIX system. *IEEE Distributed Processing Technical Committee Newsletter 6, NoSI-2* (June 1984), 7-15. Also in: *Proc. Summer USENIX Conference* (June 1984, Salt Lake City).

[GUS85a] Gusella, R., and S. Zatti. The Berkeley UNIX 4.3BSD time synchronization protocol: protocol specification. Technical Report UCB/CSD 85/250, University of California, Berkeley, June 1985.

[GUS85b] Gusella, R., and S. Zatti. An election algorithm for a distributed clock synchronization program. Technical Report UCB/CSD 86/275, University of California, Berkeley, December 1985.

[HAL84] Halpern, J.Y., B. Simons, R. Strong and D. Dolly. Fault-tolerant clock synchronization. *Proc. Third Annual ACM Symposium on Principles of Distributed Computing* (August 1984), 89-102.

[JOR85] Jordan, E.C. (Ed). *Reference Data for Engineers, Seventh Edition.* H.W. Sams & Co., New York, 1985.

[KOP87] Kopetz, H., and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Trans. Computers C-36, 8* (August 1987), 933-939.

[LAM78] Lamport, L., Time, clocks and the ordering of events in a distributed system. *Comm. ACM 21, 7* (July 1978), 558-565.

[LAM85] Lamport, L., and P.M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM 32, 1* (January 1985), 52-78.

[LIN80] Lindsay, W.C., and A.V. Kantak. Network synchronization of random signals. *IEEE Trans. Communications COM-28, 8* (August 1980), 1260-1266.

[LUN84] Lundelius, J., and N.A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Proc. Third Annual ACM Symposium on Principles of Distributed Computing* (August 1984), 75-88.

[MAR85] Marzullo, K., and S. Owicki. Maintaining the time in a distributed system. *ACM Operating Systems Review 19, 3* (July 1985), 44-54.

[MIL81a] Mills, D.L. Time Synchronization in DCNET Hosts. DARPA Internet Project Report IEN-173, COMSAT Laboratories, February 1981.

[MIL81b] Mills, D.L. DCNET Internet Clock Service. DARPA Network Working Group Report RFC-778, COMSAT Laboratories, April 1981.

[MIL83a] Mills, D.L. Internet Delay Experiments. DARPA Network Working Group Report RFC-889, M/A-COM Linkabit, December 1983.

[MIL83b] Mills, D.L. DCN local-network protocols. DARPA Network Working Group Report RFC-891, M/A-COM Linkabit, December 1983.

[MIL85a] Mills, D.L. Algorithms for synchronizing network clocks. DARPA Network Working Group Report RFC-956, M/A-COM Linkabit, September 1985.

[MIL85b] Mills, D.L. Experiments in network clock synchronization. DARPA Network Working Group Report RFC-957, M/A-COM Linkabit, September 1985.

[MIL85c] Mills, D.L. Network Time Protocol (NTP). DARPA Network Working Group Report RFC-958, M/A-COM Linkabit, September 1985.

[MIL88a] Mills, D.L. Network Time Protocol (version 1) - specification and implementation. DARPA Network Working Group Report RFC-1059, University of Delaware, July 1988.

[MIL88b] Mills, D.L. The Fuzzball. *Proc. ACM SIGCOMM 88 Symposium* (Palo Alto, CA, August 1988), 115-122.

[MIL89] Mills, D.L. Network Time Protocol (version 2) - specification and implementation. DARPA Network Working Group Report RFC-1119, University of Delaware, September 1989.

[MIL90] Mills, D.L. Measured performance of the Network Time Protocol in the Internet system. *ACM Computer Communication Review 20, 1* (January 1990), 65-75.

[MIL91a]Mills, D.L. Internet time synchronization: the Network Time Protocol. *IEEE Trans. Communications 39, 10* (October 1991), 1482-1493.

[MIL91b]Mills, D.L. On the chronology and metrology of computer network timescales and their application to the Network Time Protocol. *ACM Computer Communications Review 21, 5* (October 1991), 8-17.

[MIT80] Mitra, D. Network synchronization: analysis of a hybrid of master-slave and mutual synchronization. *IEEE Trans. Communications COM-28, 8* (August 1980), 1245-1259.

[NBS77] *Data Encryption Standard*. Federal Information Processing Standards Publication 46. National Bureau of Standards, U.S. Department of Commerce, 1977.

[NBS79] *Time and Frequency Dissemination Services*. NBS Special Publication 432, U.S. Department of Commerce, 1979.

[NBS80] *DES Modes of Operation*. Federal Information Processing Standards Publication 81. National Bureau of Standards, U.S. Department of Commerce, December 1980.

[POS80] Postel, J. User Datagram Protocol. DARPA Network Working Group Report RFC-768, USC Information Sciences Institute, August 1980.

[POS83a] Postel, J. Daytime protocol. DARPA Network Working Group Report RFC-867, USC Information Sciences Institute, May 1983.

[POS83b] Postel, J. Time protocol. DARPA Network Working Group Report RFC-868, USC Information Sciences Institute, May 1983.

[RIC88] Rickert, N.W. Non Byzantine clock synchronization - a programming experiment. *ACM Operating Systems Review 22, 1* (January 1988), 73-78.

[SCH86] Schneider, F.B. A paradigm for reliable clock synchronization. Department of Computer Science Technical Report TR 86-735, Cornell University, February 1986.

[SMI86] Smith, J. *Modern Communications Circuits*. McGraw-Hill, New York, NY, 1986.

[SRI87] Srikanth, T.K., and S. Toueg. Optimal clock synchronization. *J. ACM 34, 3* (July 1987), 626-645.

[STE88] Steiner, J.G., C. Neuman, and J.I. Schiller. Kerberos: an authentication service for open network systems. *Proc. Winter USENIX Conference* (February 1988).

[SU81] Su, Z. A specification of the Internet protocol (IP) timestamp option. DARPA Network Working Group Report RFC-781. SRI International, May 1981.

[TRI86] Tripathi, S.K., and S.H. Chang. ETempo: a clock synchronization algorithm for hierarchical LANs - implementation and measurements. Systems Research Center Technical Report TR-86-48, University of Maryland, 1986.

[VAN84] Van Dierendonck, A.J., and W.C. Melton. Applications of time transfer using NAVSTAR GPS. In: *Global Positioning System, Papers Published in Navigation, Vol. II*, Institute of Navigation, Washington, DC, 1984.

[VAS78] Vass, E.R. OMEGA navigation system: present status and plans 1977-1980. *Navigation 25, 1* (Spring 1978).