1. Appendix A. NTP Data Format - Version 3

The format of the NTP Message data area, which immediately follows the UDP header, is shown in Figure 4. Following is a description of its fields.

Leap Indicator (LI): This is a two-bit code warning of an impending leap second to be inserted/deleted in the last minute of the current day, with bit 0 and bit 1, respectively, coded as follows:

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

Version Number (VN): This is a three-bit integer indicating the NTP version number, currently three (3).

Mode: This is a three-bit integer indicating the mode, with values defined as follows:

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

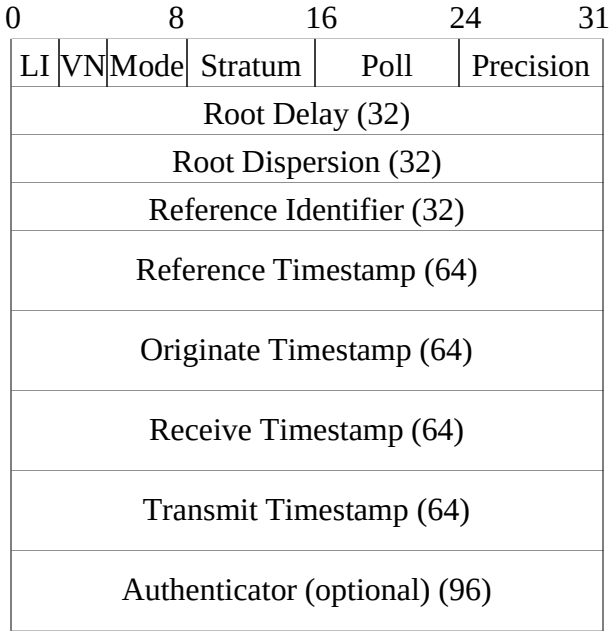

Figure 4. NTP Message Header

Stratum: This is a eight-bit integer indicating the stratum level of the local clock, with values defined as follows:

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

The values that can appear in this field range from zero to NTP.INFIN inclusive.

Poll Interval: This is an eight-bit signed integer indicating the maximum interval between successive messages, in seconds to the nearest power of two. The values that can appear in this field range from NTP.MINPOLL to NTP.MAXPOLL inclusive.

Precision: This is an eight-bit signed integer indicating the precision of the local clock, in seconds to the nearest power of two.

Root Delay: This is a 32-bit signed fixed-point number indicating the total roundtrip delay to the primary reference source, in seconds with fraction point between bits 15 and 16. Note that this variable can take on both positive and negative values, depending on clock precision and skew.

Root Dispersion: This is a 32-bit signed fixed-point number indicating the maximum error relative to the primary reference source, in seconds with fraction point between bits 15 and 16. Only positive values greater than zero are possible.

Reference Clock Identifier: This is a 32-bit code identifying the particular reference clock. In the case of stratum 0 (unspecified) or stratum 1 (primary reference), this is a four-octet, left-justified, zero-padded ASCII string. While not enumerated as part of the NTP specification, the following are suggested ASCII identifiers:

Stratum	Code	Meaning
0	DCN	DCN routing protocol
0	NIST	NIST public modem
0	TSP	TSP time protocol
0	DTS	Digital Time Service
1	ATOM	Atomic clock (calibrated)
1	VLF	VLF radio (OMEGA, etc.)
1	callsign	Generic radio
1	LORC	LORAN-C radionavigation
1	GOES	GOES UHF environment satellite
1	GPS	GPS UHF satellite positioning

In the case of stratum 2 and greater (secondary reference) this is the four-octet Internet address of the primary reference host.

Reference Timestamp: This is the local time at which the local clock was last set or corrected, in 64-bit timestamp format.

Originate Timestamp: This is the local time at which the request departed the client host for the service host, in 64-bit timestamp format.

Receive Timestamp: This is the local time at which the request arrived at the service host, in 64-bit timestamp format.

Transmit Timestamp: This is the local time at which the reply departed the service host for the client host, in 64-bit timestamp format.

Authenticator (optional): When the NTP authentication mechanism is implemented, this contains the authenticator information defined in Appendix C.

2. Appendix B. NTP Control Messages

In a comprehensive network-management environment, facilities are presumed available to perform routine NTP control and monitoring functions, such as setting the leap-indicator bits at the primary servers, adjusting the various system parameters and monitoring regular operations. Ordinarily, these functions can be implemented using a network-management protocol such as SNMP and suitable extensions to the MIB database. However, in those cases where such facilities are not available, these functions can be implemented using special NTP control messages described herein. These messages are intended for use only in systems where no other management facilities are available or appropriate, such as in dedicated-function bus peripherals. Support for these messages is not required in order to conform to this specification.

The NTP Control Message has the value 6 specified in the mode field of the first octet of the NTP header and is formatted as shown below. The format of the data field is specific to each command or response; however, in most cases the format is designed to be constructed and viewed by humans and so is coded in free-form ASCII. This facilitates the specification and implementation of simple management tools in the absence of fully evolved network-management facilities. As in ordinary NTP messages, the authenticator field follows the data field. If the authenticator is used the data field is zero-padded to a 32-bit boundary, but the padding bits are not considered part of the data field and are not included in the field count.

IP hosts are not required to reassemble datagrams larger than 576 octets; however, some commands or responses may involve more data than will fit into a single datagram. Accordingly, a simple reassembly feature is included in which each octet of the message data is numbered starting with zero. As each fragment is transmitted the number of its first octet is inserted in the offset field and the number of octets is inserted in the count field. The more-data (M) bit is set in all fragments except the last.

Most control functions involve sending a command and receiving a response, perhaps involving several fragments. The sender chooses a distinct, nonzero sequence number and sets the status field and R and E bits to zero. The responder interprets the opcode and additional information in the data field, updates the status field, sets the R bit to one and returns the three 32-bit words of the header along with additional information in the data field. In case of invalid message format or contents the responder inserts a code in the status field, sets the R and E bits to one and, optionally, inserts a diagnostic message in the data field.

Some commands read or write system variables and peer variables for an association identified in the command. Others read or write variables associated with a radio clock or other device directly connected to a source of primary synchronization information. To identify which type of variable and association a 16-bit association identifier is used. System variables are indicated by the identifier zero. As each association is mobilized a unique, nonzero identifier is created for it. These identifiers are used in a cyclic fashion, so that the chance of using an old identifier which matches a newly created association is remote. A management entity can request a list of current identifiers and subsequently use them to read and write variables for each association. An attempt to use an expired identifier results in an exception response, following which the list can be requested again.

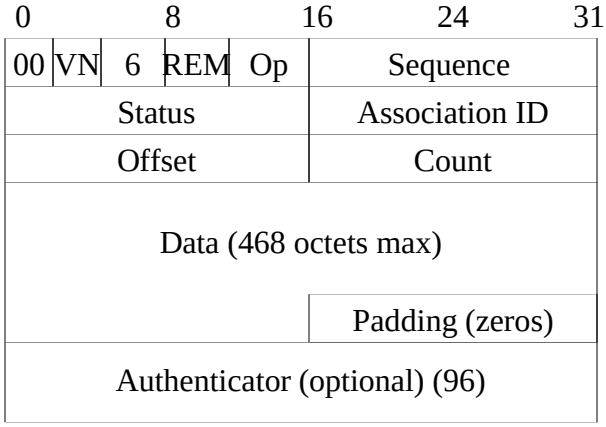

Figure 5. NTP Control Message Header

Some exception events, such as when a peer becomes reachable or unreachable, occur spontaneously and are not necessarily associated with a command. An implementation may elect to save the event information for later retrieval or to send an asynchronous response (called a trap) or both. In case of a trap the IP address and port number is determined by a previous command and the sequence field is set as described below. Current status and summary information for the latest exception event is returned in all normal responses. Bits in the status field indicate whether an exception has occurred since the last response and whether more than one exception has occurred.

Commands need not necessarily be sent by an NTP peer, so ordinary access-control procedures may not apply; however, the optional mask/match mechanism suggested elsewhere in this document provides the capability to control access by mode number, so this could be used to limit access for control messages (mode 6) to selected address ranges.

2.1. NTP Control Message Format

The format of the NTP Control Message header, which immediately follows the UDP header, is shown in Figure 5. Following is a description of its fields. Bit positions marked as zero are reserved and should always be transmitted as zero.

Version Number (VN): This is a three-bit integer indicating the NTP version number, currently three (3).

Mode: This is a three-bit integer indicating the mode. It must have the value 6, indicating an NTP control message.

Response Bit (R): Set to zero for commands, one for responses.

Error Bit (E): Set to zero for normal response, one for error response.

More Bit (M): Set to zero for last fragment, one for all others.

Operation Code (Op): This is a five-bit integer specifying the command function. Values currently defined include the following:

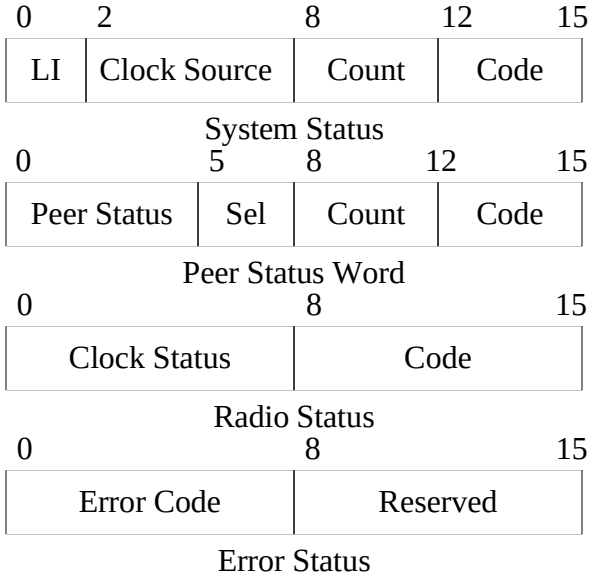

Figure 6. Status Word Formats

0	reserved
1	read status command/response
2	read variables command/response
3	write variables command/response
4	read clock variables command/response
5	write clock variables command/response
6	set trap address/port command/response
7	trap response
8-31	reserved

Sequence: This is a 16-bit integer indicating the sequence number of the command or response.

Status: This is a 16-bit code indicating the current status of the system, peer or clock, with values coded as described in following sections.

Association ID: This is a 16-bit integer identifying a valid association.

Offset: This is a 16-bit integer indicating the offset, in octets, of the first octet in the data area.

Count: This is a 16-bit integer indicating the length of the data field, in octets.

Data: This contains the message data for the command or response. The maximum number of data octets is 468.

Authenticator (optional): When the NTP authentication mechanism is implemented, this contains the authenticator information defined in Appendix C.

2.2. Status Words

Status words indicate the present status of the system, associations and clock. They are designed to be interpreted by network-monitoring programs and are in one of four 16-bit formats shown in Figure 6 and described in this section. System and peer status words are associated with responses for all commands except the read clock variables, write clock variables and set trap address/port commands. The association identifier zero specifies the system status word, while a nonzero identifier specifies a particular peer association. The status word returned in response to read clock variables and write clock variables commands indicates the state of the clock hardware and decoding software. A special error status word is used to report malformed command fields or invalid values.

2.2.1. System Status Word

The system status word appears in the status field of the response to a read status or read variables command with a zero association identifier. The format of the system status word is as follows:

Leap Indicator (LI): This is a two-bit code warning of an impending leap second to be inserted/deleted in the last minute of the current day, with bit 0 and bit 1, respectively, coded as follows:

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

Clock Source: This is a six-bit integer indicating the current synchronization source, with values coded as follows:

0	unspecified or unknown
1	Calibrated atomic clock (e.g., HP 5061)
2	VLF (band 4) or LF (band 5) radio (e.g., OMEGA, WWVB)
3	HF (band 7) radio (e.g., CHU, MSF, WWV/H)
4	UHF (band 9) satellite (e.g., GOES, GPS)
5	local net (e.g., DCN, TSP, DTS)
6	UDP/NTP
7	UDP/TIME
8	eyeball-and-wristwatch
9	telephone modem (e.g., NIST)
10-63	reserved

System Event Counter: This is a four-bit integer indicating the number of system exception events occurring since the last time the system status word was returned in a response or included in a trap message. The counter is cleared when returned in the status field of a response and freezes when it reaches the value 15.

System Event Code: This is a four-bit integer identifying the latest system exception event, with new values overwriting previous values, and coded as follows:

0	unspecified
1	system restart
2	system or hardware fault
3	system new status word (leap bits or synchronization change)
4	system new synchronization source or stratum (sys.peer or sys.stratum change)
5	system clock reset (offset correction exceeds CLOCK.MAX)
6	system invalid time or date (see NTP specification)
7	system clock exception (see system clock status word)
8-15	reserved

2.2.2. Peer Status Word

A peer status word is returned in the status field of a response to a read status, read variables or write variables command and appears also in the list of association identifiers and status words returned by a read status command with a zero association identifier. The format of a peer status word is as follows:

Peer Status: This is a five-bit code indicating the status of the peer determined by the packet procedure, with bits assigned as follows:

0	configured (peer.config)
1	authentication enabled (peer.authenable)
2	authentication okay (peer.authentic)
3	reachability okay (peer.reach $\neq$ 0)
4	reserved

Peer Selection (Sel): This is a three-bit integer indicating the status of the peer determined by the clock-selection procedure, with values coded as follows:

0	rejected
1	passed sanity checks (tests 1 through 8 in Section 3.4.3)
2	passed correctness checks (intersection algorithm in Section 4.2.1)
3	passed candidate checks (if limit check implemented)
4	passed outlyer checks (clustering algorithm in Section 4.2.2)
5	current synchronization source; max distance exceeded (if limit check implemented)
6	current synchronization source; max distance okay
7	reserved

Peer Event Counter: This is a four-bit integer indicating the number of peer exception events that occurred since the last time the peer status word was returned in a response or included in a trap message. The counter is cleared when returned in the status field of a response and freezes when it reaches the value 15.

Peer Event Code: This is a four-bit integer identifying the latest peer exception event, with new values overwriting previous values, and coded as follows:

0	unspecified
1	peer IP error
2	peer authentication failure (peer.authentic bit was one now zero)
3	peer unreachable (peer.reach was nonzero now zero)
4	peer reachable (peer.reach was zero now nonzero)
5	peer clock exception (see peer clock status word)
6-15	reserved

2.2.3. Clock Status Word

There are two ways a reference clock can be attached to a NTP service host, as an dedicated device managed by the operating system and as a synthetic peer managed by NTP. As in the read status command, the association identifier is used to identify which one, zero for the system clock and nonzero for a peer clock. Only one system clock is supported by the protocol, although many peer clocks can be supported. A system or peer clock status word appears in the status field of the response to a read clock variables or write clock variables command. This word can be considered an extension of the system status word or the peer status word as appropriate. The format of the clock status word is as follows:

Clock Status: This is an eight-bit integer indicating the current clock status, with values coded as follows:

0	clock operating within nominals
1	reply timeout
2	bad reply format
3	hardware or software fault
4	propagation failure
5	bad date format or value
6	bad time format or value
7-255	reserved

Clock Event Code: This is an eight-bit integer identifying the latest clock exception event, with new values overwriting previous values. When a change to any nonzero value occurs in the radio status field, the radio status field is copied to the clock event code field and a system or peer clock exception event is declared as appropriate.

2.2.4. Error Status Word

An error status word is returned in the status field of an error response as the result of invalid message format or contents. Its presence is indicated when the E (error) bit is set along with the response (R) bit in the response. It consists of an eight-bit integer coded as follows:

0	unspecified

1	authentication failure
2	invalid message length or format
3	invalid opcode
4	unknown association identifier
5	unknown variable name
6	invalid variable value
7	administratively prohibited
8-255	reserved

2.3. Commands

Commands consist of the header and optional data field shown in Figure 6. When present, the data field contains a list of identifiers or assignments in the form

<identifier>[=<value>],<identifier>[=<value>],...

where <identifier> is the ASCII name of a system or peer variable specified in Table 2 or Table 3 and <value> is expressed as a decimal, hexadecimal or string constant in the syntax of the C programming language. Where no ambiguity exists, the “sys.” or “peer.” prefixes shown in Table 2 or Table 4 can be suppressed. Whitespace (ASCII nonprinting format effectors) can be added to improve readability for simple monitoring programs that do not reformat the data field. Internet addresses are represented as four octets in the form [n.n.n.n], where n is in decimal notation and the brackets are optional. Timestamps, including reference, originate, receive and transmit values, as well as the logical clock, are represented in units of seconds and fractions, preferably in hexadecimal notation, while delay, offset, dispersion and distance values are represented in units of milliseconds and fractions, preferably in decimal notation. All other values are represented as-is, preferably in decimal notation.

Implementations may define variables other than those listed in Table 2 or Table 3. Called extramural variables, these are distinguished by the inclusion of some character type other than alphanumeric or “.” in the name. For those commands that return a list of assignments in the response data field, if the command data field is empty, it is expected that all available variables defined in Table 3 or Table 4 of the NTP specification will be included in the response. For the read commands, if the command data field is nonempty, an implementation may choose to process this field to individually select which variables are to be returned.

Commands are interpreted as follows:

Read Status (1): The command data field is empty or contains a list of identifiers separated by commas. The command operates in two ways depending on the value of the association identifier. If this identifier is nonzero, the response includes the peer identifier and status word. Optionally, the response data field may contain other information, such as described in the Read Variables command. If the association identifier is zero, the response includes the system identifier (0) and status word, while the data field contains a list of binary-coded pairs

<association identifier> <status word>,

one for each currently defined association.

Read Variables (2): The command data field is empty or contains a list of identifiers separated by commas. If the association identifier is nonzero, the response includes the requested peer identifier and status word, while the data field contains a list of peer variables and values as described above. If the association identifier is zero, the data field contains a list of system variables and values. If a peer has been selected as the synchronization source, the response includes the peer identifier and status word; otherwise, the response includes the system identifier (0) and status word.

Write Variables (3): The command data field contains a list of assignments as described above. The variables are updated as indicated. The response is as described for the Read Variables command.

Read Clock Variables (4): The command data field is empty or contains a list of identifiers separated by commas. The association identifier selects the system clock variables or peer clock variables in the same way as in the Read Variables command. The response includes the requested clock identifier and status word and the data field contains a list of clock variables and values, including the last timecode message received from the clock.

Write Clock Variables (5): The command data field contains a list of assignments as described above. The clock variables are updated as indicated. The response is as described for the Read Clock Variables command.

Set Trap Address/Port (6): The command association identifier, status and data fields are ignored. The address and port number for subsequent trap messages are taken from the source address and port of the control message itself. The initial trap counter for trap response messages is taken from the sequence field of the command. The response association identifier, status and data fields are not significant. Implementations should include sanity timeouts which prevent trap transmissions if the monitoring program does not renew this information after a lengthy interval.

Trap Response (7): This message is sent when a system, peer or clock exception event occurs. The opcode field is 7 and the R bit is set. The trap counter is incremented by one for each trap sent and the sequence field set to that value. The trap message is sent using the IP address and port fields established by the set trap address/port command. If a system trap the association identifier field is set to zero and the status field contains the system status word. If a peer trap the association identifier field is set to that peer and the status field contains the peer status word. Optional ASCII-coded information can be included in the data field.

3. Appendix C. Authentication Issues

NTP robustness requirements are similar to those of other multiple-peer distributed protocols used for network routing, management and file access. These include protection from faulty implementations, improper operation and possibly malicious replay attacks with or without data modification. These requirements are especially stringent with distributed protocols, since damage due to failures can propagate quickly throughout the network, devastating archives, routes and monitoring systems and even bring down major portions of the network in the fashion of the classic Internet Worm.

The access-control mechanism suggested in the NTP specification responds to these requirements by limiting access to trusted peers. The various sanity checks resist most replay and spoofing attacks by discarding old duplicates and using the originate timestamp as a one-time pad, since it is unlikely that even a synchronized peer can predict future timestamps with the precision required on the basis of past observations alone. In addition, the protocol environment resists jamming attacks by employing redundant time servers and diverse network paths. Resistance to stochastic disruptions, actual or manufactured, are minimized by careful design of the filtering and selection algorithms.

However, it is possible that a determined intruder can disrupt timekeeping operations between peers by subtle modifications of NTP message data, such as falsifying header fields or certain timestamps. In cases where protection from even these types of attacks is required, a specifically engineered message-authentication mechanism based on cryptographic techniques is necessary. Typical mechanisms involve the use of cryptographic certificates, algorithms and key media, together with secure media databases and key-management protocols. Ongoing research efforts in this area are directed toward developing a standard methodology that can be used with many protocols, including NTP. However, while it may eventually be the case that ubiquitous, widely applicable authentication methodology may be adopted by the Internet community and effectively overtake the mechanism described here, it does not appear that specific standards and implementations will happen within the lifetime of this particular version of NTP.

The NTP authentication mechanism described here is intended for interim use until specific standards and implementations operating at the network level or transport level are available. Support for this mechanism is not required in order to conform to the NTP specification itself. The mechanism, which operates at the application level, is designed to protect against unauthorized message-stream modification and misrepresentation of source by insuring that unbroken, authenticated paths exist between a trusted, stratum-one server in a particular synchronization subnet and all other servers in that subnet. It employs a crypto-checksum, computed by the sender and checked by the receiver, together with a set of predistributed algorithms, certificates and cryptographic keys indexed by a key identifier included in the message. However, there are no provisions in NTP itself to distribute or maintain the certificates, algorithms or keys. These quantities may occasionally be changed, which may result in inconsistent key information while rekeying is in progress. The nature of NTP itself is quite tolerant to such disruptions, so no particular provisions are included to deal with them.

The intent of the authentication mechanism is to provide a framework that can be used in conjunction with selected mode combinations to build specific plans to manage clockworking communities and

implement policy as necessary. It can be selectively enabled or disabled on a per-peer basis. There is no specific plan proposed to manage the use of such schemes; although several possibilities are immediately obvious. In one scenario a group of time servers peers among themselves using symmetric modes and shares one secret key, say key 1, while another group of servers peers among themselves using symmetric modes and shares another secret key, say key 2. Now, assume by policy it is decided that selected servers in group 1 can provide synchronization to group 2, but not the other way around. The selected servers in group 1 are given key 2, but operated only in server mode, so cannot accept synchronization from group 2; however, group 2 has authenticated access to group-1 servers. Many other scenarios are possible with suitable combinations of modes and keys.

A packet format and crypto-checksum procedure appropriate for NTP is specified in the following sections. The cryptographic information is carried in an authenticator which follows the (unmodified) NTP header fields. The crypto-checksum procedure uses the Data Encryption Standard (DES) [NBS77]; however, only the DES encryption algorithm is used and the decryption algorithm is not necessary. This feature is specifically targeted toward governmental sensitivities on the export of cryptographic technology, since the DES decryption algorithm need not be included in NTP software distributions and thus cannot be extracted and used in other applications to avoid message data disclosure.

3.1. NTP Authentication Mechanism

When it is created and possibly at other times, each association is allocated variables identifying the certificate authority, encryption algorithm, cryptographic key and possibly other data. The specific procedures to allocate and initialize these variables are beyond the scope of this specification, as are the association of the identifiers and keys and the management and distribution of the keys themselves. For example and consistency with the conventions of the NTP specification, a set of appropriate peer and packet variables might include the following:

Authentication Enabled Bit (peer.authenable): This is a bit indicating that the association is to operate in the authenticated mode. For configured peers this bit is determined from the startup environment. For non-configured peers, this bit is set to one if an arriving message includes the authenticator and set to zero otherwise.

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. The system variable peer.hostkeyid is used for active associations. The peer.peerkeyid variable is initialized at zero (unspecified) when the association is mobilized. For purposes of authentication an unassigned value is interpreted as zero (unspecified).

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

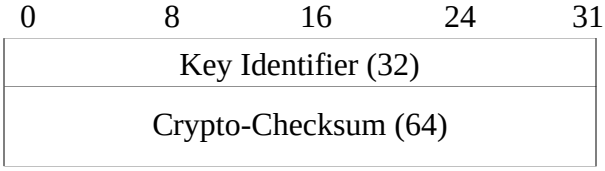

Figure 7. Authenticator Format

odd-parity bit 8. By convention, the unspecified key 0 (zero), consisting of eight odd-parity zero octets, is used for testing and presumed known throughout the NTP community. The remaining keys are distributed using methods outside the scope of NTP.

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

The authenticator field consists of two subfields, one consisting of the pkt.keyid variable and the other the pkt.check variable computed by the encrypt procedure, which is called by the transmit procedure described in the NTP specification, and by the decrypt procedure, which is called by the receive procedure described in the NTP specification. Its presence is revealed by the fact the total datagram length according to the UDP header is longer than the NTP message length, which includes the header plus the data field, if present. For authentication purposes, the NTP message is zero-padded if necessary to a 64-bit boundary, although the padding bits are not considered part of the NTP message itself. The authenticator format shown in Figure 7 has 96 bits, including a 32-bit key identifier and 64-bit crypto-checksum, and is aligned on a 32-bit boundary for efficient computation. Additional information required in some implementations, such as certificate authority and encryption algorithm, can be inserted between the (padded) NTP message and the key identifier, as long as the alignment conditions are met. Like the authenticator itself, this information is not included in the crypto-checksum. Use of these data are beyond the scope of this specification. These conventions may be changed in future as the result of other standardization activities.

3.2. NTP Authentication Procedures

When authentication is implemented there are two additional procedures added to those described in the NTP specification. One of these (encrypt) constructs the crypto-checksum in transmitted messages, while the other (decrypt) checks this quantity in received messages. The procedures use a variant of the cipher-block chaining method described in [NBS80] as applied to DES. In principal, the procedure is independent of DES and requires only that the encryption algorithm operate on 64-bit blocks. While the NTP authentication mechanism specifies the use of DES, other algorithms could be used by prior arrangement.

3.2.1. Encrypt Procedure

For ordinary NTP messages the encryption procedure operates as follows. If authentication is not enabled, the procedure simply exits. If the association is active (modes 1, 3, 5), the key is determined from the system key identifier. If the association is passive (modes 2, 4) the key is determined from the peer key identifier, if the authentic bit is set, or as the default key (zero) otherwise. These conventions allow further protection against replay attacks and keying errors, as well as facilitate

testing and migration to new versions. The crypto-checksum is calculated using the 64-bit NTP header and data words, but not the authenticator or padding bits.

```
begin encrypt procedure
        if (peer.authenable = 0) exit;              /* do nothing if not enabled */
        if (peer.hostmode = 1 or peer.hostmode = 3 or peer.hostmode = 5)
                keyid ← peer.hostkeyid;             /* active modes use system key */
        else
                if (peer.authentic = 1)             /* passive modes use peer key */
                        keyid ← peer.peerkeyid;
                else
                        keyid ← 0;                  /* unauthenticated use key 0 */
        temp ← 0;                                   /* calculate crypto-checksum */
        for (each 64-bit header and data word) begin
                temp ← temp xor word;
                temp ← DES(temp, keyid);
                endfor;
        pkt.keyid ← keyid;                          /* insert packet variables */
        pkt.check ← temp;
        end encrypt procedure;
```

3.2.2. Decrypt Procedure

For ordinary messages the decryption procedure operates as follows. If the peer is not configured, the data portion of the message is inspected to determine if the authenticator fields are present. If so, authentication is enabled; otherwise, it is disabled. If authentication is enabled and the authenticator fields are present and the crypto-checksum succeeds, the authentication bit is set to one; otherwise, it is set to zero.

```
begin decrypt procedure
        peer.authentic ← 0;
        if (peer.config = 0)                        /* if not configured, enable per packet */
                if (authenticator present)
                        peer.authenable ← 1;
                else
                        peer.authenable ← 0;
        if (peer.authenable = 0 or authenticator not present)) exit;
        peer.peerkeyid ← pkt.keyid;                 /* use peer key */
        temp ← 0;                                   /* calculate crypto-checksum */
        for (each 64-bit header and data word) begin
                temp ← temp xor word;
                temp ← DES(temp, peer.peerkeyid);
```

```
        endfor;
    if (temp == pkt.check) peer.authentic ← 1;      /* declare result */
    end decrypt procedure;
```

3.2.3. Control-Message Procedures

In anticipation that the functions provided by the NTP control messages will eventually be subsumed by a comprehensive network-managment function, the peer variables are not used for control message authentication. If an NTP command message is received with an authenticator field, the crypto-checksum is computed as in the decrypt procedure and the response message includes the authenticator field as computed by the encrypt procedure. If the received authenticator is correct, the key for the response is the same as in the command; otherwise, the default key (zero) is used. Commands causing a change to the peer data base, such as the write variables and set trap address/port commands, must be correctly authenticated; however, the remaining commands are normally not authenticated in order to minimize the encryption overhead.

4. Appendix D. Differences from Previous Versions.

The original NTP, later called NTP Version 0, was described in RFC-958 [MIL85c]. Subsequently, Version 0 was superseded by Version 1 (RFC-1059 [MIL88a]), and Version 2 (RFC-1119 [MIL89]. The Version-2 description was split into two documents, RFC-1119 defining the architecture and specifying the protocol and algorithms, and another [MIL90b] describing the service model, algorithmic analysis and operating experience. In previous versions these two objectives were combined in one document. While the architecture assumed in Version 3 is identical to Version 2, the protocols and algorithms differ in minor ways. Differences between NTP Version 3 and previous versions are described in this Appendix. Due to known bugs in very old implementations, continued support for Version-0 implementations is not recommended. It is recommended that new implementations follow the guidelines below when interoperating with older implementations.

Version 3 neither changes the protocol in any significant way nor obsoletes previous versions or existing implementations. The main motivation for the new version is to refine the analysis and implementation models for new applications at much higher network speeds to the gigabit-per-second regime and to provide for the enhanced stability, accuracy and precision required at such speeds. In particular, the sources of time and frequency errors have been rigorously examined and error bounds established in order to improve performance, provide a model for correctness assertions and indicate timekeeping quality to the user. Version 3 also incorporates two new optional features, (1) an algorithm to combine the offsets of a number of peer time servers in order to enhance accuracy and (2) improved local-clock algorithms which allow the poll intervals on all synchronization paths to be substantially increased in order to reduce network overhead. Following is a summary of previous versions of the protocol together with details of the Version 3 changes.

1. Version 1 supports no modes other than symmetric-active and symmetric-passive, which are determined by inspecting the port-number fields of the UDP packet header. The peer mode can be determined explicitly from the packet-mode variable (pkt.mode) if it is nonzero and implicitly from the source port (pkt.peerport) and destination port (pkt.hostport) variables if it is zero. For the case where pkt.mode is zero the mode is determined as follows:

pkt.peerport	pkt.hostport	Mode
NTP.PORT	NTP.PORT	symmetric active
NTP.PORT	not NTP.PORT	server
not NTP.PORT	NTP.PORT	client
not NTP.PORT	not NTP.PORT	not possible

   Note that it is not possible in this case to distinguish between symmetric active and symmetric passive modes. Use of the pkt.mode and NTP.PORT variables in this way is not recommended and may not be supported in future versions of the protocol. The low-order three bits of the first octet, specified as zero in Version 1, are used for the mode field in Version 2. Version-2 and Version-3 implementations interoperating with Version-1 implementations should operate in a passive mode only and use the value one in the version number (pkt.version) field and zero in the mode (pkt.mode) field in transmitted messages.

2. Version 1 does not support the NTP control message described in Appendix B. Certain old versions of the Unix NTP daemon *ntpd* use the high-order bits of the stratum field (pkt.stratum) for control and monitoring purposes. While these bits are never set during normal Version-1, Version-2 or Version-3 operations, new implementations may use the NTP reserved mode 6 described in Appendix B and/or private reserved mode 7 for special purposes, such as remote control and monitoring, and in such cases the format of the packet following the first octet can be arbitrary. While there is no guarantee that different implementations can interoperate using private reserved mode 7, it is recommended that vanilla ASCII format be used whenever possible.

3. Version 1 does not support authentication. The key identifiers, cryptographic keys and procedures described in Appendix C are new to Version 2 and continued in Version 3, along with the corresponding variables, procedures and authenticator fields. In the NTP message described in Appendix A and NTP control message described in Appendix B the format and contents of the header fields are independent of the authentication mechanism and the authenticator itself follows the header fields, so that previous versions will ignore the authenticator.

4. In Version 1 the total dispersion (pkt.rootdispersion) field of the NTP header was called the estimated drift rate, but not used in the protocol or timekeeping procedures. Implementations of the Version-1 protocol typically set this field to the current value of the skew-compensation register, which is a signed quantity. In a Version 2 implementation apparent large values in this field may affect the order considered in the clock-selection procedure. Version-2 and Version-3 implementations interoperating with older implementations should assume this field is zero, regardless of its actual contents.

5. Version 2 and Version 3 incorporate several sanity checks designed to avoid disruptions due to unsynchronized, duplicate or bogus timestamp information. The checks in Version 3 are specifically designed to detect lost or duplicate packets and resist invalid timestamps. The leap-indicator bits are set to show the unsynchronized state if updates are not received from a reference source for a considerable time or if the reference source has not received updates for a considerable time. Some Version-1 implementations could claim valid synchronization indefinitely following loss of the reference source.

6. The clock-selection procedure of Version 2 was considerably refined as the result of accumulated experience with the Version-1 implementation. Additional sanity checks are included for authentication, range bounds and to avoid use of very old data. The candidate list is sorted twice, once to select a relatively few robust candidates from a potentially large population of unruly peers and again to order the resulting list by measurement quality. As in Version 1, The final selection procedure repeatedly casts out outlyers on the basis of weighted dispersion.

7. The local-clock procedure of Version 2 were considerably improved over Version 1 as the result of analysis, simulation and experience. Checks have been added to warn that the oscillator has gone too long without update from a reference source. The compliance register has been added to improve frequency stability to the order of a millisecond per day. The various parameters were retuned for optimum loop stability using measured data over typical Internet paths and

with typical local-clock hardware. In version 3 the phase-lock loop model was further refined to provide an adaptive-bandwidth feature that automatically adjusts for the inherent stabilities of the reference clock and local clock while providing optimum loop stability in each case.

8. Problems in the timekeeping calculations of Version 1 with high-speed LANs were found and corrected in Version 2. These were caused by jitter due to small differences in clock rates and different precisions between the peers. Subtle bugs in the Version-1 reachability and polling-rate control were found and corrected. The peer.valid and sys.hold variables were added to avoid instabilities when the reference source changes rapidly due to large dispersive delays under conditions of severe network congestion. The peer.config, peer.authenable and peer.authentic bits were added to control special features and simplify configuration.

9. In Version 3 The local-clock algorithm has been overhauled to improve stability and accuracy. Appendix G presents a detailed mathematical model and design example which has been refined with the aid of feedback-control analysis and extensive simulation using data collected over ordinary Internet paths. Section 5 of RFC-1119 on the NTP local clock has been completely rewritten to describe the new algorithm. Since the new algorithm can result in message rates far below the old ones, it is highly recommended that they be used in new implementations. Note that this algorithm is not integral to the NTP protocol specification itself and its use does not affect interoperability with previous versions or existing implementations; however, in order to insure overall NTP subnet stability in the Internet, it is essential that the local-clock characteristics of all NTP time servers conform to the analytical models presented previously and in this document.

10. In Version 3 a new algorithm to combine the offsets of a number of peer time servers is presented in Appendix F. This algorithm is modelled on those used by national standards laboratories to combine the weighted offsets from a number of standard clocks to construct a synthetic laboratory timescale more accurate than that of any clock separately. It can be used in an NTP implementation to improve accuracy and stability and reduce errors due to asymmetric paths in the Internet. The new algorithm has been simulated using data collected over ordinary Internet paths and, along with the new local-clock algorithm, implemented and tested in the Fuzzball time servers now running in the Internet. Note that this algorithm is not integral to the NTP protocol specification itself and its use does not affect interoperability with previous versions or existing implementations.

11. Several inconsistencies and minor errors in previous versions have been corrected in Version 3. The description of the procedures has been rewritten in pseudo-code augmented by English commentary for clarity and to avoid ambiguity. Appendix I has been added to illustrate C-language implementations of the various filtering and selection algorithms suggested for NTP. Additional information is included in Section 5 and in Appendix E, which includes the tutorial material formerly included in Section 2 of RFC-1119, as well as much new material clarifying the interpretation of timescales and leap seconds.

12. Minor changes have been made in the Version-3 local-clock algorithms to avoid problems observed when leap seconds are introduced in the UTC timescale and also to support an auxiliary

precision oscillator, such as a cesium clock or timing receiver, as a precision timebase. In addition, changes were made to some procedures described in Section 3 and in the clock-filter and clock-selection procedures described in Section 4. While these changes were made to correct minor bugs found as the result of experience and are recommended for new implementations, they do not affect interoperability with previous versions or existing implementations in other than minor ways (at least until the next leap second).

13. In Version 3 changes were made to the way delay, offset and dispersion are defined, calculated and processed in order to reliably bound the errors inherent in the time-transfer procedures. In particular, the error accumulations were moved from the delay computation to the dispersion computation and both included in the clock filter and selection procedures. The clock-selection procedure was modified to remove the first of the two sorting/discarding steps and replace with an algorithm first proposed by Marzullo and later incorporated in the Digital Time Service. These changes do not significantly affect the ordinary operation of or compatibility with various versions of NTP, but they do provide the basis for formal statements of correctness as described in Appendix H.

5. Appendix E. The NTP Timescale and its Chronometry

5.1. Introduction

Following is an extended discussion on *computer network chronometry*, which is the precise determination of computer time and frequency relative to international standards and the determination of conventional civil time and date according to the modern calendar. It describes the methods conventionally used to establish civil time and date and the various timescales now in use. In particular, it characterizes the Network Time Protocol (NTP) timescale relative to the Coordinated Universal Time (UTC) timescale, and establishes the precise interpretation of UTC leap seconds in NTP.

In the following discussion the terms *time*, *oscillator*, *clock*, *epoch*, *calendar*, *date* and *timescale* are used in a technical sense. Strictly speaking, the time of an event is an abstraction which determines the ordering of events in some given frame of reference. An oscillator is a generator capable of precise frequency (relative to the given frame of reference) to a specified tolerance. A clock is an oscillator together with a counter which records the (fractional) number of cycles since being initialized with a given value at a given time. The value of the counter at any given time is called its epoch at that time. In general, epoches are not continuous and depend on the precision of the counter.

A calendar is a mapping from epoch in some frame of reference to the times and dates used in everyday life. Since multiple calendars are in use today and sometimes disagree on the dating of the same events in the past, the chronometry of past and present events is an art practiced by historians. One of the goals of this discussion is to provide a standard chronometry for precision dating of present and future events in a global networking community. To *synchronize frequency* means to adjust the oscillators in the network to run at the same frequency, to *synchronize time* means to set the clocks so that all agree at a particular epoch with respect to UTC, as provided by international standards, and to *synchronize clocks* means to synchronize them in both frequency and time.

In order to synchronize clocks, there must be some way to directly or indirectly compare them in time and frequency. The ultimate frame of reference for our world consists of the cosmic oscillators: the Sun, Moon and other galactic orbiters. Since the frequencies of these oscillators are relatively unstable and not known exactly, the ultimate reference standard oscillator has been chosen by international agreement as a synthesis of many observations of an atomic transition of exquisite stability. The epoches of each heavenly and Earthbound oscillator defines a distinctive timescale, not necessarily always continuous, relative to the standard oscillator. Another goal of this presentation is to describe a standard chronometry to rationalize conventional computer time and UTC; in particular, how to handle leap seconds.

5.2. Primary Frequency and Time Standards

A primary frequency standard is an oscillator that can maintain extremely precise frequency relative to a physical phenomenon, such as a transition in the orbital states of an electron. Presently available atomic oscillators are based on the transitions of the hydrogen, cesium and rubidium atoms. Table

Oscillator type	Stability (per day)	Drift /Aging (per day)
Hydrogen maser	2×10^{-14}	1×10^{-12}/yr
Cesium beam	3×10^{-13}	3×10^{-12}/yr
Rubidium gas cell	5×10^{-12}	3×10^{-11}/mo
Oven-controlled crystal	1×10^{-9} 0-50 deg C	1×10^{-10}
Digital-comp crystal	5×10^{-8} 0-60 deg C	1×10^{-9}
Temp-compensated crystal	5×10^{-7} 0-60 deg C	3×10^{-9}
Uncompensated crystal	~1×10^{-6} per deg C	don't ask

Table 7. Characteristics of Standard Oscillators

7 shows the characteristics for typical oscillators of these types compared with those for various types of quartz-crystal oscillators found in electronic equipment. For reasons of cost and robustness cesium oscillators are used worldwide for national primary frequency standards. On the other hand, local clocks used in computing equipment almost always are designed with uncompensated crystal oscillators.

For the three atomic oscillators listed in Table 7 the drift/aging column shows the maximum offset per day from nominal standard frequency due to systematic mechanical and electrical characteristics. In the case of crystal oscillators this offset is not constant, which results in a gradual change in frequency with time, called aging. Even if a crystal oscillator is temperature compensated by some means, it must be periodically compared to a primary standard in order to maintain the highest accuracy. For all types of oscillators the stability column shows the maximum variation in frequency per day due to circuit noise and environmental factors.

As the telephone networks of the world are evolving rapidly to digital technology, consideration should be given to the methods used for frequency synchronization in digital networks. A network of clocks in which each oscillator is phase-locked to a single frequency standard is called *isochronous*, while a network in which some oscillators are phase-locked to different master oscillators, but with the master oscillators closely synchronized in frequency (not necessarily phase locked), to a single frequency standard is called plesiochronous. In plesiochronous systems the phase of some oscillators can slip relative to others and cause occasional data errors in synchronous transmission systems.

The industry has agreed on a classification of clock oscillators as a function of minimum accuracy, minimum stability and other factors [ALL74a]. There are three factors which determine the classification: stability, jitter and wander. Stability refers to the systematic variation of frequency with time and is synonymous with aging, drift, trends, etc. Jitter (also called timing jitter) refers to short-term variations in frequency with components greater than 10 Hz, while wander refers to long-term variations in frequency with components less than 10 Hz. The classification determines the oscillator stratum (not to be confused with the NTP stratum), with the more accurate oscillators assigned the lower strata and less accurate oscillators the higher strata:

Stratum	Min Accuracy (per day)	Min Stability (per day)
1	1×10^{-11}	not specified
2	1.6×10^{-8}	1×10^{-10}
3	4.6×10^{-6}	3.7×10^{-7}
4	3.2×10^{-5}	not specified

The construction, operation and maintenance of stratum-one oscillators is assumed to be consistent with national standards and often includes cesium oscillators or precision crystal oscillators synchronized via LORAN-C to national standards. Stratum-two oscillators represent the stability required for interexchange toll switches such as the AT&T 4ESS and interexchange digital cross-connect systems, while stratum-three oscillators represent the stability required for exchange switches such as the AT&T 5ESS and local cross-connect systems. Stratum-four oscillators represent the stability required for digital channel-banks and PBX systems.

5.3. Time and Frequency Dissemination

In order that atomic and civil time can be coordinated throughout the world, national administrations operate primary time and frequency standards and coordinate them cooperatively by observing various radio broadcasts and through occasional use of portable atomic clocks. Most seafaring nations of the world operate some sort of broadcast time service for the purpose of calibrating chronographs, which are used in conjunction with ephemeris data to determine navigational position. In many countries the service is primitive and limited to seconds-pips broadcast by marine communication stations at certain hours. For instance, a chronograph error of one second represents a longitudinal position error of about 0.23 nautical mile at the Equator.

The U.S. National Institute of Standards and Technology (NIST - formerly National Bureau of Standards) operates three radio services for the dissemination of primary time and frequency information. One of these uses high-frequency (HF or CCIR band 7) transmissions on frequencies of 2.5, 5, 10, 15 and 20 MHz from Fort Collins, CO (WWV), and Kauai, HI (WWVH). Signal propagation is usually by reflection from the upper ionospheric layers, which vary in height and composition throughout the day and season and result in unpredictable delay variations at the receiver. The timecode is transmitted over a 60-second interval at a data rate of 1 bps using a 100-Hz subcarrier on the broadcast signal. The timecode information includes UTC time-day information, but does not currently include year or leap-second warning. While these transmissions and those of Canada from Ottawa, Ontario (CHU), and other countries can be received over large areas in the western hemisphere, reliable frequency comparisons can be made only to the order of 10^{-7} and time accuracies are limited to the order of a millisecond [BLA74]. Radio clocks which operate with these transmissions include the Traconex 1020, which provides accuracies to about ten milliseconds and is priced in the $1,500 range.

A second service operated by NIST uses low-frequency (LF or CCIR band 5) transmissions on 60 kHz from Boulder, CO (WWVB), and can be received over the continental U.S. and adjacent coastal areas. Signal propagation is via the lower ionospheric layers, which are relatively stable and have predictable diurnal variations in height. The timecode is transmitted over a 60-second interval at a

rate of 1 pps using periodic reductions in carrier power. With appropriate receiving and averaging techniques and corrections for diurnal and seasonal propagation effects, frequency comparisons to within 10^{-11} are possible and time accuracies of from a few to 50 microseconds can be obtained [BLA74]. Some countries in western Europe operate similar services which use transmissions on 60 kHz from Rugby, U.K. (MSF), and on 77.5 kHz from Mainflingen, West Germany (DCF77). The timecode information includes UTC time-day-year information and leap-second warning. Radio clocks which operate with these transmissions include the Spectracom 8170 and Kinemetrics/TrueTime 60-DC and LF-DC, which provide accuracies to a millisecond or less and are priced in the $2,500 range. However, these receivers do not extract the year information and leap-second warning.

The third service operated by NIST uses ultra-high frequency (UHF or CCIR band 9) transmissions on about 468 MHz from the Geosynchronous Orbit Environmental Satellites (GOES), three of which cover the western hemisphere. The timecode is interleaved with messages used to interrogate remote sensors and consists of 60 4-bit binary-coded decimal words transmitted over an interval of 30 seconds. The timecode information includes UTC time-day-year information and leap-second warning. Radio clocks which operate with these transmissions include the Kinemetrics/TrueTime 468-DC, which provides accuracies to 0.5 ms and is priced in the $6,000 range. However, this receiver does not extract the year information and leap-second warning.

The U.S. Department of Defense is developing the Global Positioning System (GPS) for worldwide precision navigation. This system will eventually provide 24-hour worldwide coverage using a constellation of 24 satellites in 12-hour orbits. For time-transfer applications GPS has a potential accuracy in the order of a few nanoseconds; however, various considerations of defense policy may limit accuracy to hundreds of nanoseconds [VAN84]. The timecode information includes GPS time and UTC correction; however, there appears to be no leap-second warning. Radio clocks which operate with these transmissions include the Kinemetrics/TrueTime GPS-DC, which provides accuracies to 200 μs and is priced in the $12,000 range. However, since only about half the satellites have been launched, expensive rubidium or quartz oscillators are necessary to preserve accuracy during outages. Also, since this is a single-channel receiver, it must be supplied with geographic coordinates within a degree from an external source before operation begins.

The U.S. Coast Guard, along with agencies of other countries, has operated the LORAN-C [FRA82] radionavigation system for many years. It currently provides time-transfer accuracies of less than a microsecond and eventually may achieve 100 ns within the ground-wave coverage area of a few hundred kilometers from the transmitter. Beyond the ground wave area signal propagation is via the lower ionospheric layers, which decreases accuracies to the order of 50 us. With the recent addition of the Mid-Continent Chain, the deployment of LORAN-C transmitters now provides complete coverage of the U.S. LORAN-C timing receivers, such as the Austron 2000, are specialized and extremely expensive (up to $20,000). They are used primarily to monitor local cesium clocks and are not suited for unattended, automatic operation. While the LORAN-C system provides a highly accurate frequency and time reference within the ground wave area, there is no timecode modulation, so the receiver must be supplied with UTC time to within a few tens of seconds from an external source before operation begins.

The OMEGA [VAS78] radionavigation system operated by the U.S. Navy and other countries consists of eight very-low-frequency (VLF or CCIR band 4) transmitters operating on frequencies from 10.2 to 13.1 kHz and providing 24-hour worldwide coverage. With appropriate receiving and averaging techniques and corrections for propagation effects, frequency comparisons and time accuracies are comparable to the LF systems, but with worldwide coverage [BLA74]. Radio clocks which operate with these transmissions include the Kinemetrics/TrueTime OM-DC, which provides accuracies to 1 ms and is priced in the $3,500 range. While the OMEGA system provides a highly accurate frequency reference, there is no timecode modulation, so the receiver must be supplied with geographic coordinates within a degree and UTC time within five seconds from an external source before operation begins. There are several other VLF services intended primarily for worldwide data communications with characteristics similar to OMEGA. These services can be used in a manner similar to OMEGA, but this requires specialized techniques not suited for unattended, automatic operation.

Note that not all transmission formats used by NIST radio broadcast services [NBS79] and no currently available radio clocks include provisions for year information and leap-second warning. This information must be determined from other sources. NTP includes provisions to distribute advance warnings of leap seconds using the leap-indicator bits described in the NTP specification. The protocol is designed so that these bits can be set manually or by the radio timecode at the primary time servers and then automatically distributed throughout the synchronization subnet to all other time servers.

5.4. Calendar Systems

The calendar systems used in the ancient world reflect the agricultural, political and ritual needs characteristic of the societies in which they flourished. Astronomical observations to establish the winter and summer solstices were in use three to four millennia ago. By the 14th century BC the Shang Chinese had established the solar year as 365.25 days and the lunar month as 29.5 days. The lunisolar calendar, in which the ritual month is based on the Moon and the agricultural year on the Sun, was used throughout the ancient Near East (except Egypt) and Greece from the third millennium BC. Early calendars used either thirteen lunar months of 28 days or twelve alternating lunar months of 29 and 30 days and haphazard means to reconcile the 354/364-day lunar year with the 365-day vague solar year.

The ancient Egyptian lunisolar calendar had twelve 30-day lunar months, but was guided by the seasonal appearance of the star Sirius (Sothis). In order to reconcile this calendar with the solar year, a civil calendar was invented by adding five intercalary days for a total of 365 days. However, in time it was observed that the civil year was about one-fourth day shorter than the actual solar year and thus would precess relative to it over a 1460-year cycle called the Sothic cycle. Along with the Shang Chinese, the ancient Egyptians had thus established the solar year at 365.25 days, or within about 11 minutes of the present measured value. In 432 BC, about a century after the Chinese had done so, the Greek astronomer Meton calculated there were 110 lunar months of 29 days and 125 lunar months of 30 days for a total of 235 lunar months in 6940 solar days, or just over 19 years.

The 19-year cycle, called the Metonic cycle, established the lunar month at 29.532 solar days, or within about two minutes of the present measured value.

The Roman republican calendar was based on a lunar year and by 50 BC was eight weeks out of step with the solar year. Julius Caesar invited the Alexandrian astronomer Sosigenes to redesign the calendar, which led to the adoption in 46 BC of the Julian calendar. This calendar is based on a year of 365 days with an intercalary day inserted every four years. However, for the first 36 years an intercalary day was mistakenly inserted every three years instead of every four. The result was 12 intercalary days instead of nine, and a series of corrections that was not complete until 8 AD.

The seven-day Sumerian week was introduced only in the fourth century AD by Emperor Constantine I. During the Roman era a 15-year census cycle, called the Indiction cycle, was instituted for taxation purposes. The sequence of day-names for consecutive occurrences of a particular day of the year does not recur for 28 years, called the solar cycle. Thus, the least common multiple of the 28-year solar cycle, 19-year Metonic cycle and 15-year Indiction cycle results in a grand 7980-year supercycle called the Julian Era, which began in 4713 BC. A particular combination of the day of the week, day of the year, phase of the Moon and round of the census will recur beginning in 3268 AD.

By 1545 the discrepancy in the Julian year relative to the solar year had accumulated to ten days. In 1582, following suggestions by the astronomers Christopher Clavius and Luigi Lilio, Pope Gregory XIII issued a papal bull which decreed, among other things, that the solar year would consist of 365.2422 days. In order to more closely approximate the new value, only those centennial years divisible by 400 would be leap years, while the remaining centennial years would not, making the actual value 365.2425, or within about 26 seconds of the current measured value. Since the beginning of the Common Era and prior to 1990 there were 474 intercalary days inserted in the Julian calendar, but 14 of these were removed in the Gregorian calendar. While the Gregorian calendar is in use throughout most of the world today, some countries did not adopt it until early in the twentieth century.

While it remains a fascinating field for time historians, the above narrative provides conclusive evidence that conjugating calendar dates of significant events and assigning NTP timestamps to them is approximate at best. In principle, reliable dating of such events requires only an accurate count of the days relative to some globally alarming event, such as a comet passage or supernova explosion; however, only historically persistent and politically stable societies, such as the ancient Chinese and Egyptian, and especially the classic Maya, possessed the means and will to do so.

5.5. The Modified Julian Day System

In order to measure the span of the universe or the decay of the proton, it is necessary to have a standard day-numbering plan. Accordingly, the International Astronomical Union has adopted the use of the standard second and Julian Day Number (JDN) to date cosmological events and related phenomena. The standard day consists of 86,400 standard seconds, where time is expressed as a fraction of the whole day, and the standard year consists of 365.25 standard days.

In the scheme devised in 1583 by the French scholar Joseph Julius Scaliger and named after his father, Julius Caesar Scaliger, JDN 0.0 corresponds to 12^h (noon) on the first day of the Julian Era, 1 January 4713 BC. The years prior to the Common Era (BC) are reckoned according to the Julian calendar, while the years of the Common Era (AD) are reckoned according to the Gregorian calendar. Since 1 January 1 AD in the Gregorian calendar corresponds to 3 January 1 in the Julian calendar [DER90], JDN 1,721,426.0 corresponds to 12^h on the first day of the Common Era, 1 January 1 AD. The Modified Julian Date (MJD), which is sometimes used to represent dates near our own era in conventional time and with fewer digits, is defined as MJD = JD – 2,400,000.5. Following the convention that our century began at 0^h on 1 January 1900, at which time the tropical year was already 12^h old, that eclectic instant corresponds to MJD 15,020.0. Thus, the Julian timescale ticks in standard (atomic) 365.25-day centuries and was set to a given value at the approximate epoch of a cosmic event which apparently synchronized the entire human community, the origin of the Common Era.

5.6. Determination of Frequency

For many years the most important use of time and frequency information was for worldwide navigation and space science, which depend on astronomical observations of the Sun, Moon and stars [JOR85]. Sidereal time is based on the transit of stars across the celestial meridian of an observer. The mean sidereal day is 23 hours, 56 minutes and 4.09 seconds, but varies about ±30 ms throughout the year due to polar wandering and orbit variations. Ephemeris time is based on tables with which a standard time interval such as the tropical year - one complete revolution of the Earth around the Sun - can be determined through observations of the Sun, Moon and planets. In 1958 the standard second was defined as 1/31,556,925.9747 of the tropical year that began this century. On this scale the tropical year is 365.2421987 days and the lunar month - one complete revolution of the Moon around the Earth - is 29.53059 days; however, the actual tropical year can be determined only to an accuracy of about 50 ms and has been increasing by about 5.3 ms per year.

Of the three heavenly oscillators readily apparent to ancient mariners and astronomers - the Earth rotation about its axis, the Earth revolution around the Sun and the Moon revolution around the Earth - none of the three have the intrinsic stability, relative to modern technology, to serve as a standard reference oscillator. In 1967 the standard second was redefined as “9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium-133 atom.” Since 1972 the time and frequency standards of the world have been based on International Atomic Time (TAI), which is defined and maintained using multiple cesium-beam oscillators to an accuracy of a few parts in 10^{13}, or better than a microsecond per day. Note that, while this provides an extraordinarily precise timescale, it does not necessarily agree with conventional solar time and may not in fact even be absolutely uniform, unless subtle atomic conspiracies can be ruled out.

5.7. Determination of Time and Leap Seconds

The International Bureau of Weights and Measures (IBWM) uses astronomical observations provided by the U.S. Naval Observatory and other observatories to determine UTC. Starting from apparent mean solar time as observed, the UT0 timescale is determined using corrections for Earth

UTC Date	MJD	NTP Time	Offset
01 Jan 72	41,317	2,272,060,800	0
30 Jun 72	41,498	2,287,785,600	1
31 Dec 72	41,682	2,303,683,200	2
31 Dec 73	42,047	2,335,219,200	3
31 Dec 74	42,412	2,366,755,200	4
31 Dec 75	42,777	2,398,291,200	5
31 Dec 76	43,143	2,429,913,600	6
31 Dec 77	43,508	2,461,449,600	7
31 Dec 78	43,873	2,492,985,600	8
31 Dec 79	44,238	2,524,521,600	9
30 Jun 81	44,785	2,571,782,400	10
30 Jun 82	45,150	2,603,318,400	11
30 Jun 83	45,515	2,634,854,400	12
30 Jun 85	46,246	2,698,012,800	13
31 Dec 87	47,160	2,776,982,400	14
31 Dec 89	47,891	2,840,140,800	15
31 Dec 90	48,256	2,871,676,800	16

Table 8. Table of Leap-Second Insertions

orbit and inclination (the Equation of Time, as used by sundials), the UT1 (navigator's) timescale by adding corrections for polar migration and the UT2 timescale by adding corrections for known periodicity variations. While standard frequencies are based on TAI, conventional civil time is based on UT1, which is presently slowing relative to TAI by a fraction of a second per year. When the magnitude of correction approaches 0.7 second, a leap second is inserted or deleted in the TAI timescale on the last day of June or December.

For the most precise coordination and timestamping of events since 1972, it is necessary to know when leap seconds are implemented in UTC and how the seconds are numbered. As specified in CCIR Report 517, which is reproduced in [BLA74], a leap second is inserted following second 23:59:59 on the last day of June or December and becomes second 23:59:60 of that day. A leap second would be deleted by omitting second 23:59:59 on one of these days, although this has never happened. Leap seconds were inserted prior to 1 January 1991 on the occasions listed in Table 8 (courtesy U.S. Naval Observatory). Published IBWM corrections consist not only of leap seconds, which result in step discontinuities relative to TAI, but 100-ms UT1 adjustments called DUT1, which provide increased accuracy for navigation and space science.

Note that the NTP time column actually shows the epoch following the last second of the day given in the UTC date and MJD columns (except for the first line), which is the precise epoch of insertion. The offset column shows the cumulative seconds offset between the uncoordinated (Julian) timescale and the UTC timescale; that is, the number of seconds to add to the Julian clock in order to maintain nominal agreement with the UTC clock. Finally, note that the epoch of insertion is relative to the timescale immediately prior to that epoch; e.g., the epoch of the 31 December 90

insertion is determined on the timescale in effect following the 31 December 1990 insertion, which means the actual insertion relative to the Julian clock is fourteen seconds later than the apparent time on the UTC timescale.

The UTC timescale thus ticks in standard (atomic) seconds and was set to the value 0^h MJD 41,317.0 at the epoch determined by astronomical observation to be 0^h on 1 January 1972 according to the Gregorian calendar; that is, the inaugural tick of the UTC Era. In fact, the inaugural tick which synchronized the cosmic oscillators, Julian clock, UTC clock and Gregorian calendar forevermore was displaced about ten seconds from the civil clock then in use, while the GPS clock is ahead of the UTC clock by six seconds in late 1990. Subsequently, the UTC clock has marched backward relative to the Julian timescale exactly one second on scheduled occasions at monumental epoches embedded in the institutional memory of our civilization. Note in passing that leap-second adjustments affect the number of seconds per day and thus the number of seconds per year. Apparently, should we choose to worry about it, the UTC clock, Julian clock and various cosmic clocks will inexorably drift apart with time until rationalized by some future papal bull.

5.8. The NTP Timescale and Reckoning with UTC

The NTP timescale is based on the UTC timescale, but not necessarily always coincident with it. At 0^h on 1 January 1972 (MJD 41,317.0), the first tick of the UTC Era, the NTP clock was set to 2,272,060,800, representing the number of standard seconds since 0^h on 1 January 1900 (MJD 15,020.0). The insertion of leap seconds in UTC and subsequently into NTP does not affect the UTC or NTP oscillator, only the conversion to conventional civil UTC time. However, since the only institutional memory available to NTP are the UTC timecode broadcast services, the NTP timescale is in effect reset to UTC as each timecode is received. Thus, when a leap second is inserted in UTC and subsequently in NTP, knowledge of all previous leap seconds is lost.

Another way to describe this is to say there are as many NTP timescales as historic leap seconds. In effect, a new timescale is established after each new leap second. Thus, all previous leap seconds, not to mention the apparent origin of the timescale itself, lurch backward one second as each new timescale is established. If a clock synchronized to NTP in 1990 was used to establish the UTC epoch of an event that occurred in early 1972 without correction, the event would appear fifteen seconds late relative to UTC. However, NTP primary time servers resolve the epoch using the broadcast timecode, so that the NTP clock is set to the broadcast value on the current timescale. As a result, for the most precise determination of epoch relative to the historic UTC clock, the user must subtract from the apparent NTP epoch the offsets shown in Table 8 at the relative epoches shown. This is a feature of almost all present day time-distribution mechanisms.

The chronometry involved can be illustrated with the help of Figure 8, which shows the details of seconds numbering just before, during and after the last scheduled leap insertion at 23:59:59 on 31 December 1989. Notice the NTP leap bits are set on the day prior to insertion, as indicated by the "+" symbols on the figure. Since this makes the day one second longer than usual, the NTP day rollover will not occur until the end of the first occurrence of second 800. The UTC time conversion routines must notice the apparent time and the leap bits and handle the timescale conversions accordingly. Immediately after the leap insertion both timescales resume ticking the seconds as if

	UTC		NTP	
	hours	seconds	kiloseconds	seconds
31 Dec 90	23:59	:59	2,871,590	,399 +
(leap)	23:59	:60	2,871,590	,400 +
1 Jan 91	00:00	:00	2,871.590	,400
	00:00	:01	2,871,590	,401

Figure 8. Comparison of UTC and NTP Timescales at Leap

the leap had never happened. The chronometric correspondence between the UTC and NTP timescales continues, but NTP has forgotten about all past leap insertions. In NTP chronometric determination of UTC time intervals spanning leap seconds will thus be in error, unless the exact times of insertion are known.

It is possible that individual systems may use internal data formats other than the NTP timestamp format, which is represented in seconds to a precision of about 200 picoseconds; however, a persuasive argument exists to use a two-part representation, one part for whole days (MJD or some fixed offset from it) and the other for the seconds (or some scaled value, such as milliseconds). This not only facilitates conversion between NTP and conventional civil time, but makes the insertion of leap seconds much easier. All that is required is to change the modulus of the seconds counter, which on overflow increments the day counter. This design insures that continuity of the timescale is assured, even if outside synchronization is lost before, during or after leap-second insertion. Since timestamp data are unaffected, synchronization is assured, even if timestamp data are in flight at the instant and originated before or at that instant.

6. Appendix F. The NTP Clock-Combining Algorithm

6.1. Introduction

A common problem in synchronization subnets is systematic time-offset errors resulting from asymmetric transmission paths, where the networks or transmission media in one direction are substantially different from the other. The errors can range from microseconds on high-speed ring networks to large fractions of a second on satellite/landline paths. It has been found experimentally that these errors can be considerably reduced by combining the apparent offsets of a number of time servers to produce a more accurate working offset. Following is a description of the combining method used in the NTP implementation for the Fuzzball [MIL88b]. The method is similar to that used by national standards laboratories to determine a synthetic laboratory timescale from an ensemble of cesium clocks [ALL74b]. These procedures are optional and not required in a conforming NTP implementation.

In the following description 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*.

6.2. Determining Time and Frequency

Figure 9 shows the overall organization of the NTP time-server model. Timestamps exchanged with possibly many other subnet peers are used to determine individual roundtrip delays and clock offsets relative to each peer as described in the NTP specification. As shown in the figure, the computed delays and offsets are processed by the clock filter to reduce incidental timing noise and the most accurate and reliable subset determined by the clock-selection algorithm. The resulting offsets of this subset are first combined as described below and then processed by the phase-locked loop (PLL). In the PLL the combined effects of the filtering, selection and combining operations is to produce a phase-correction term. This is processed by the loop filter to control the local clock, which functions as a voltage-controlled oscillator (VCO). The VCO furnishes the timing (phase) reference to produce the timestamps used in all calculations.

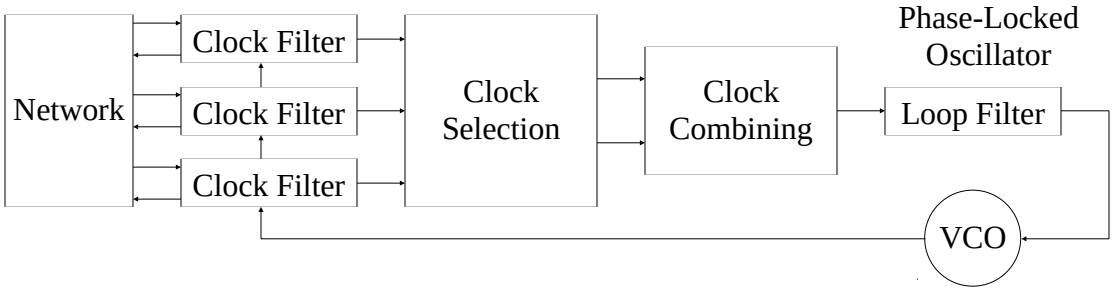

Figure 9. Network Time Protocol

6.3. Clock Modelling

The International Standard (SI) definition of *time interval* is in terms of the standard second: "the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium-133 atom." Let u represent the standard unit of time interval so defined and $v = \frac{1}{u}$ be the standard unit of frequency. The *epoch*, denoted by t, is defined as the reading of a counter that runs at frequency v and began counting at some agreed initial epoch t_0, which defines the *standard* or *absolute timescale*. For the purposes of the following analysis, the epoch of the standard timescale, as well as the *time* indicated by a clock will be considered continuous. In practice, time is determined relative to a clock constructed from an atomic oscillator and system of counter/dividers, which defines a timescale associated with that particular oscillator. Standard time and frequency are then determined from an ensemble of such timescales and algorithms designed to combine them to produce a composite timescale approximating the standard timescale.

Let $T(t)$ be the time displayed by a clock at epoch t relative to the standard timescale:

$$T(t) = \tfrac{1}{2}D(t_0)[t - t_0]^2 + R(t_0)[t - t_0] + T(t_0) + x(t) ,$$

where $D(t_0)$ is the fractional frequency drift per unit time, $R(t_0)$ the frequency and $T(t_0)$ the time at some previous epoch t_0. In the usual stationary model these quantities can be assumed constant or changing slowly with epoch. The random nature of the clock is characterized by $x(t)$, which represents the random noise (jitter) relative to the standard timescale. In the usual analysis the second-order term $D(t_0)$ is ignored and the noise term $x(t)$ modelled as a normal distribution with predictable spectral density or autocorrelation function.

The probability density function of time offset $p(t - T(t))$ usually appears as a bell-shaped curve centered somewhere near zero. The width and general shape of the curve are determined by $x(t)$, which depends on the oscillator precision and jitter characteristics, as well as the measurement system and its transmission paths. Beginning at epoch t_0 the offset is set to zero, following which the bell creeps either to the left or right, depending on the value of $R(t_0)$ and accelerates depending on the value of $D(t_0)$.

6.4. Development of a Composite Timescale

Now consider the time offsets of a number of real clocks connected by real networks. A display of the offsets of all clocks relative to the standard timescale will appear as a system of bell-shaped curves slowly precessing relative to each other, but with some further away from nominal zero than others. The bells will normally be scattered over the offset space, more or less close to each other, with some overlapping and some not. The problem is to estimate the true offset relative to the standard timescale from a system of offsets collected routinely between the clocks.

A composite timescale can be determined from a sequence of offsets measured between the n clocks of an ensemble at nominal intervals τ. Let $R_i(t_0)$ be the frequency and $T_i(t_0)$ the time of the ith clock

at epoch t_0 relative to the standard timescale and let "^" designate the associated estimates. Then, an estimator for T_i computed at t_0 for epoch $t_0 + \tau$ is

$$\hat{T}_i(t_0 + \tau) = \hat{R}_i(t_0)\tau + T_i(t_0) ,$$

neglecting second-order terms. Consider a set of n independent time-offset measurements made between the clocks at epoch $t_0 + \tau$ and let the offset between clock i and clock j at that epoch be $T_{ij}(t_0 + \tau)$, defined as

$$T_{ij}(t_0 + \tau) \equiv T_i(t_0 + \tau) - T_j(t_0 + \tau) .$$

Note that $T_{ij} = -T_{ji}$ and $T_{ii} = 0$. Let $w_i(\tau)$ be a previously determined weight factor associated with the ith clock for the nominal interval τ. The basis for new estimates at epoch $t_0 + \tau$ is

$$T_j(t_0 + \tau) = \sum_{i=1}^{n} w_i(\tau)[\hat{T}_i(t_0 + \tau) + T_{ji}(t_0 + \tau)].$$

That is, the apparent time indicated by the jth clock is a weighted average of the estimated time of each clock at epoch $t_0 + \tau$ plus the time offset measured between the jth clock and that clock at epoch $t_0 + \tau$.

An intuitive grasp of the behavior of this algorithm can be gained with the aid of a few examples. For instance, if $w_i(\tau)$ is unity for the ith clock and zero for all others, the apparent time for each of the other clocks is simply the estimated time $\hat{T}_i(t_0 + \tau)$. If $w_i(\tau)$ is zero for the ith clock, that clock can never affect any other clock and its apparent time is determined entirely from the other clocks. If $w_i(\tau) = 1/n$ for all i, the apparent time of the ith clock is equal to the average of the time estimates computed at t_0 plus the average of the time offsets measured to all other clocks. Finally, in a system with two clocks and $w_i(\tau) = 1/2$ for each, and if the estimated time at epoch $t_0 + \tau$ is fast by 1 s for one clock and slow by 1 s for the other, the apparent time for both clocks will coincide with the standard timescale.

In order to establish a basis for the next interval τ, it is necessary to update the frequency estimate $\hat{R}_i(t_0 + \tau)$ and weight factor $w_i(\tau)$. The average frequency assumed for the ith clock during the previous interval τ is simply the difference between the times at the beginning and end of the interval divided by τ. A good estimator for $R_i(t_0 + \tau)$ has been found to be the exponential average of these differences, which is given by

$$\hat{R}_i(t_0 + \tau) = \hat{R}_i(t_0) + \alpha_i[\hat{R}_i(t_0) - \frac{T_i(t_0 + \tau) - T_i(t_0)}{\tau}] ,$$

where α_i is an experimentally determined weight factor which depends on the estimated frequency error of the ith clock. In order to calculate the weight factor $w_i(\tau)$, it is necessary to determine the

expected error $\varepsilon_i(\tau)$ for each clock. In the following, braces "|" indicate absolute value and brackets "<>" indicate the infinite time average. In practice, the infinite averages are computed as exponential time averages. An estimate of the magnitude of the unbiased error of the *i*th clock accumulated over the nominal interval τ is

$$\varepsilon_i(\tau) = |\hat{T}_i(t_0 + \tau) - T_i(t_0 + \tau)| + \frac{0.8 < \varepsilon_e^2(\tau) >}{\sqrt{< \varepsilon_i^2(\tau) >}},$$

where $\varepsilon_i(\tau)$ and $\varepsilon_e(\tau)$ are the accumulated error of the *i*th clock and entire clock ensemble, respectively. The accumulated error of the entire ensemble is

$$< \varepsilon_e^2(\tau) > = \left[\sum_{i=1}^{n} \frac{1}{< \varepsilon_i^2(\tau) >} \right]^{-1}.$$

Finally, the weight factor for the *i*th clock is calculated as

$$w_i(\tau) = \frac{< \varepsilon_e^2(\tau) >}{< \varepsilon_i^2(\tau) >}.$$

When all estimators and weight factors have been updated, the origin of the estimation interval is shifted and the new value of t_0 becomes the old value of $t_0 + \tau$.

While not entering into the above calculations, it is useful to estimate the frequency error, since the ensemble clocks can be located some distance from each other and become isolated for some time due to network failures. The frequency-offset error in R_i is equivalent to the fractional frequency y_i,

$$y_i = \frac{\nu_i - \nu_I}{\nu_I}$$

measured between the *i*th timescale and the standard timescale *I*. Temporarily dropping the subscript *i* for clarity, consider a sequence of *N* independent frequency-offset samples $y(j)$ ($j = 1, 2, \ldots, N$) where the interval between samples is uniform and equal to *T*. Let τ be the nominal interval over which these samples are averaged. The Allan variance $\sigma_y^2(N, T, \tau)$ [ALL74a] is defined as

$$<\sigma_y^2(N, T, \tau)> = < \frac{1}{N-1} \left[\sum_{j=1}^{N} y(j)^2 - \frac{1}{N} \left(\sum_{j=1}^{N} y(j) \right)^2 \right] >,$$

A particularly useful formulation is $N = 2$ and $T = \tau$:

$$<\sigma_y^2(N = 2,\ T = \tau,\ \tau)> \equiv \sigma_y^2(\tau) = <\frac{[y(j + 1) - y(j)]^2}{2}> ,$$

so that

$$\sigma_y^2(\tau) = \frac{1}{2(N - 1)} \sum_{j=1}^{n-1} [y(j + 1) - y(j)]^2 .$$

While the Allan variance has found application when estimating errors in ensembles of cesium clocks, its application to NTP is limited due to the computation and storage burden. As described in the next section, it is possible to estimate errors with some degree of confidence using normal byproducts of NTP processing algorithms.

6.5. Application to NTP

The NTP clock model is somewhat less complex than the general model described above. For instance, at the present level of development it is not necessary to separately estimate the time and frequency of all peer clocks, only the time and frequency of the local clock. If the timekeeping reference is the local clock itself, then the offsets available in the peer.offset peer variables can be used directly for the T_{ij} quantities above. In addition, the NTP local-clock model incorporates a type-II phase-locked loop, which itself reliably estimates frequency errors and corrects accordingly. Thus, the requirement for estimating frequency is entirely eliminated.

There remains the problem of how to determine a robust and easily computable error estimate ε_i. The method described above, although analytically justified, is most difficult to implement. Happily, as a byproduct of the NTP clock-filter algorithm, a useful error estimate is available in the form of the dispersion. As described in the NTP specification, the dispersion includes the absolute value of the weighted average of the offsets between the chosen offset sample and the $n - 1$ other samples retained for selection. The effectiveness of this estimator was compared with the above estimator by simulation using observed timekeeping data and found to give quite acceptable results.

The NTP clock-combining algorithm can be implemented with only minor modifications to the algorithms as described in the NTP specification. Although elsewhere in the NTP specification the use of general-purpose multiply/divide routines has been successfully avoided, there seems to be no way to avoid them in the clock-combining algorithm. However, for best performance the local-clock algorithm described elsewhere in this document should be implemented as well, since the combining algorithms result in a modest increase in phase noise which the revised local-clock algorithm is designed to suppress.

6.6. Clock-Combining Procedure

The result of the NTP clock-selection procedure is a set of survivors (there must be at least one) that represent truechimers, or correct clocks. When clock combining is not implemented, one of these peers, chosen as the most likely candidate, becomes the synchronization source and its computed offset becomes the final clock correction. Subsequently, the system variables are adjusted

as described in the NTP clock-update procedure. When clock combining is implemented, these actions are unchanged, except that the final clock correction is computed by the clock-combining procedure.

The clock-combining procedure is called from the clock-select procedure. It constructs from the variables of all surviving peers the final clock correction Θ. The estimated error required by the algorithms previously described is based on the synchronization distance Λ computed by the distance procedure, as defined in the NTP specification. The reciprocal of Λ is the weight of each clock-offset contribution to the final clock correction. The following pseudo-code describes the procedure.

begin clock-combining procedure
    *temp*1 $\leftarrow$ 0;
    *temp*2 $\leftarrow$ 0;
    for (each *peer* remaining on the candidate list) /* scan all survivors */
        $\Lambda \leftarrow$ distance(*peer*);
        $temp \leftarrow \frac{1}{\text{peer.stratum} \times \text{NTP.MAXDISPERSE} + \Lambda}$;
        *temp*1 $\leftarrow$ *temp*1 + *temp*; /* update weight and offset */
        *temp*2 $\leftarrow$ *temp*2 + *temp* $\times$ peer.offset;
        endif;
    $\Theta \leftarrow \frac{temp2}{temp1}$; /* compute final correction */
    end clock-combining procedure;

The value Θ is the final clock correction used by the local-clock procedure to adjust the clock.

7. Appendix G. Computer Clock Modelling and Analysis

A computer clock includes some kind of reference oscillator, which is stabilized by a quartz crystal or some other means, such as the power grid. Usually, the clock includes a prescaler, which divides the oscillator frequency to a standard value, such as 1 MHz or 100 Hz, and a counter, implemented in hardware, software or some combination of the two, which can be read by the processor. For systems intended to be synchronized to an external source of standard time, there must be some means to correct the phase and frequency by occasional vernier adjustments produced by the timekeeping protocol. Special care is necessary in all timekeeping system designs to insure that the clock indications are always monotonically increasing; that is, system time never “runs backwards.”

7.1. Computer Clock Models

The simplest computer clock consists of a hardware latch which is set by overflow of a hardware counter or prescaler, and causes a processor interrupt or *tick*. The latch is reset when acknowledged by the processor, which then increments the value of a software clock counter. The phase of the clock is adjusted by adding periodic corrections to the counter as necessary. The frequency of the clock can be adjusted by changing the value of the increment itself, in order to make the clock run faster or slower. The precision of this simple clock model is limited to the tick interval, usually in the order of 10 ms; although in some systems the tick interval can be changed using a kernel variable.

This software clock model requires a processor interrupt on every tick, which can cause significant overhead if the tick interval is small, say in the order less 1 ms with the newer RISC processors. Thus, in order to achieve timekeeping precisions less than 1 ms, some kind of hardware assist is required. A straightforward design consists of a voltage-controlled oscillator (VCO), in which the

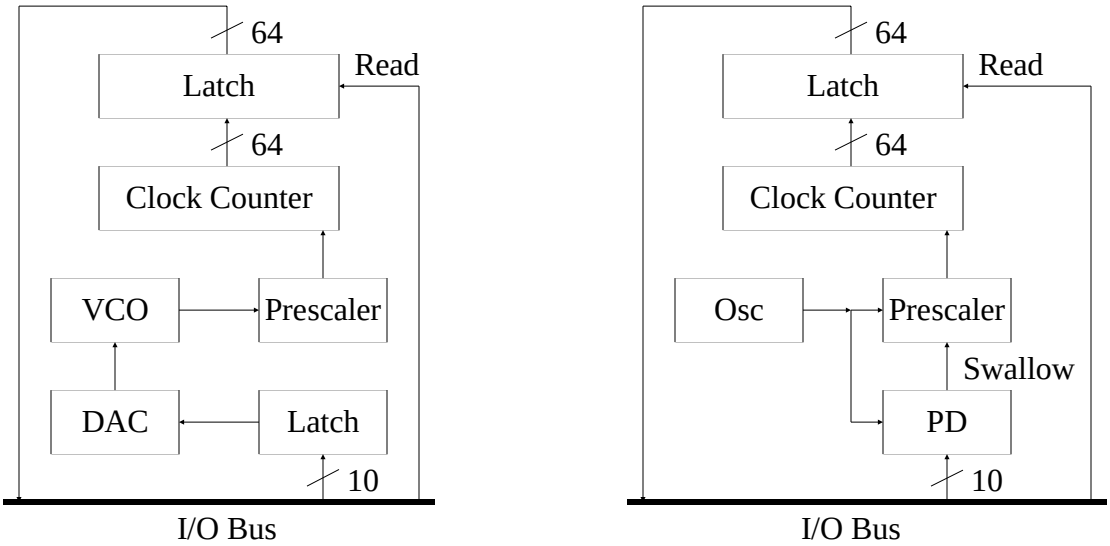

Figure 10. Hardware Clock Models

frequency is controlled by a buffered, digital/analog converter (DAC). Under the assumption that the VCO tolerance is 10^{-4} or 100 parts-per-million (ppm) (a reasonable value for inexpensive crystals) and the precision required is 100 μs (a reasonable goal for a RISC processor), the DAC must include at least ten bits.

A design sketch of a computer clock constructed entirely of hardware logic components is shown in Figure 10a. The clock is read by first pulsing the read signal, which latches the current value of the clock counter, then adding the contents of the clock-counter latch and a 64-bit clock-offset variable, which is maintained in processor memory. The clock phase is adjusted by adding a correction to the clock-offset variable, while the clock frequency is adjusted by loading a correction to the DAC latch. In principle, this clock model can be adapted to any precision by changing the number of bits of the prescaler or clock counter or changing the VCO frequency. However, it does not seem useful to reduce precision much below the minimum interrupt latency, which is in the low microseconds for a modern RISC processor.

If it is not possible to vary the oscillator frequency, which might be the case if the oscillator is an external frequency standard, a design such as shown in Figure 10b may be used. It includes a fixed-frequency oscillator and prescaler which includes a dual-modulus *swallow counter* that can be operated in either divide-by-10 or divide-by-11 modes as controlled by a pulse produced by a programmable divider (PD). The PD is loaded with a value representing the frequency offset. Each time the divider overflows a pulse is produced which switches the swallow counter from the divide-by-10 mode to the divide-by-11 mode and then back again, which in effect “swallows” or deletes a single pulse of the prescaler pulse train.

The pulse train produced by the prescaler is controlled precisely over a small range by the contents of the PD. If programmed to emit pulses at a low rate, relatively few pulses are swallowed per second and the frequency counted is near the upper limit of its range; while, if programmed to emit pulses at a high rate, relatively many pulses are swallowed and the frequency counted is near the lower limit. Assuming some degree of freedom in the choice of oscillator frequency and prescaler ratios, this design can compensate for a wide range of oscillator frequency tolerances.

In all of the above designs it is necessary to limit the amount of adjustment incorporated in any step to insure that the system clock indications are always monotonically increasing. With the software clock model this is assured as long as the increment is never negative. When the magnitude of a phase adjustment exceeds the tick interval (as corrected for the frequency adjustment), it is necessary to spread the adjustments over mulitple tick intervals. This strategy amounts to a deliberate frequency offset sustained for an interval equal to the total number of ticks required and, in fact, is a feature of the Unix clock model discussed below.

In the hardware clock models the same considerations apply; however, in these designs the tick interval amounts to a single pulse at the prescaler output, which may be in the order of 1 ms. In order to avoid decreasing the indicated time when a negative phase correction occurs, it is necessary to avoid modifying the clock-offset variable in processor memory and to confine all adjustments to the VCO or prescaler. Thus, all phase adjustments must be performed by means of programmed frequency adjustments in much the same way as with the software clock model described previously.

It is interesting to conjecture on the design of a processor assist that could provide all of the above functions in a compact, general-purpose hardware interface. The interface might consist of a multifunction timer chip such as the AMD 9513A, which includes five 16-bit counters, each with programmable load and hold registers, plus an onboard crystal oscillator, prescaler and control circuitry. A 48-bit hardware clock counter would utilize three of the 16-bit counters, while the fourth would be used as the swallow counter and the fifth as the programmable divider. With the addition of a programmable-array logic device and architecture-specific host interface, this compact design could provide all the functions necessary for a comprehensive timekeeping system.

7.1.1. The Fuzzball Clock Model

The Fuzzball clock model uses a combination of hardware and software to provide precision timing with a minimum of software and processor overhead. The model includes an oscillator, prescaler and hardware counter; however, the oscillator frequency remains constant and the hardware counter produces only a fraction of the total number of bits required by the clock counter. A typical design uses a 64-bit software clock counter and a 16-bit hardware counter which counts the prescaler output. A hardware-counter overflow causes the processor to increment the software counter at the bit corresponding to the frequency $2^N f_p$, where N is the number of bits of the hardware counter and f_p is the counted frequency at the prescaler output. The processor reads the clock counter by first generating a read pulse, which latches the hardware counter, and then adding its contents, suitably aligned, to the software counter.

The Fuzzball clock can be corrected in phase by adding a (signed) adjustment to the software clock counter. In practice, this is done only when the local time is substantially different from the time indicated by the clock and may violate the monotonicity requirement. Vernier phase adjustments determined in normal system operation must be limited to no more than the period of the counted frequency, which is 1 kHz for LSI-11 Fuzzballs. In the Fuzzball model these adjustments are performed at intervals of 4 s, called the *adjustment interval*, which provides a maximum frequency adjustment range of 250 ppm. The adjustment opportunities are created using the interval-timer facility, which is a feature of most operating systems and independent of the time-of-day clock. However, if the counted frequency is increased from 1 kHz to 1 MHz for enhanced precision, the adjustment frequency must be increased to 250 Hz, which substantially increases processor overhead. A modified design suitable for high precision clocks is presented in the next section.

In some applications involving the Fuzzball model, an external pulse-per-second (pps) signal is available from a reference source such as a cesium clock or GPS receiver. Such a signal generally provides much higher accuracy than the serial character string produced by a radio timecode receiver, typically in the low nanoseconds. In the Fuzzball model this signal is processed by an interface which produces a hardware interrupt coincident with the arrival of the pps pulse. The processor then reads the clock counter and computes the residual modulo 1 s of the clock counter. This represents the local-clock error relative to the pps signal.

Assuming the seconds numbering of the clock counter has been determined by a reliable source, such as a timecode receiver, the offset within the second is determined by the residual computed above. In the NTP local-clock model the timecode receiver or NTP establishes the time to within

±128 ms, called the aperture, which guarantees the seconds numbering to within the second. Then, the pps residual can be used directly to correct the oscillator, since the offset must be less than the aperture for a correctly operating timecode receiver and pps signal.

The above technique has an inherent error equal to the latency of the interrupt system, which in modern RISC processors is in the low tens of microseconds. It is possible to improve accuracy by latching the hardware time-of-day counter directly by the pps pulse and then reading the counter in the same way as usual. This requires additional circuitry to prioritize the pps signal relative to the pulse generated by the program to latch the counter.

7.1.2. The Unix Clock Model

The Unix 4.3bsd clock model is based on two system calls, *settimeofday* and *adjtime*, together with two kernel variables *tick* and *tickadj*. The *settimeofday* call unceremoniously resets the kernel clock to the value given, while the *adjtime* call slews the kernel clock to a new value numerically equal to the sum of the present time of day and the (signed) argument given in the *adjtime* call. In order to understand the behavior of the Unix clock as controlled by the Fuzzball clock model described above, it is helpful to explore the operations of *adjtime* in more detail.

The Unix clock model assumes an interrupt produced by an onboard frequency source, such as the clock counter and prescaler described previously, to deliver a pulse train in the 100-Hz range. In priniciple, the power grid frequency can be used, although it is much less stable than a crystal oscillator. Each interrupt causes an increment called *tick* to be added to the clock counter. The value of the increment is chosen so that the clock counter, plus an initial offset established by the *settimeofday* call, is equal to the time of day in microseconds.

The Unix clock can actually run at three different rates, one corresponding to *tick*, which is related to the intrinsic frequency of the particular oscillator used as the clock source, one to *tick* + *tickadj* and the third to *tick* – *tickadj*. Normally the rate corresponding to *tick* is used; but, if *adjtime* is called, the argument δ given is used to calculate an interval $\Delta t = \delta \frac{tick}{tickadj}$ during which one or the other of the two rates are used, depending on the sign of δ. The effect is to slew the clock to a new value at a small, constant rate, rather than incorporate the adjustment all at once, which could cause the clock to be set backward. With common values of *tick* = 10 ms and *tickadj* = 5 μs, the maximum frequency adjustment range is $\pm\frac{tickadj}{tick} = \pm\frac{5 \text{ x } 10^{-6}}{10^{-2}}$ or ±500 ppm. Even larger ranges may be required in the case of some workstations (e.g., SPARCstations) with extremely poor component tolerances.

When precisions not less than about 1 ms are required, the Fuzzball clock model can be adapted to the Unix model by software simulation, as described in Section 5 of the NTP specification, and calling *adjtime* at each adjustment interval. When precisions substantially better than this are required, the hardware microsecond clock provided in some workstations can be used together with certain refinements of the Fuzzball and Unix clock models. The particular design described below

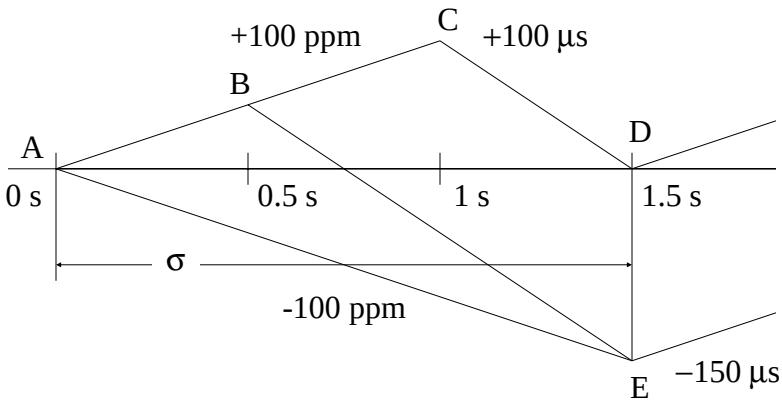

Figure 11. Clock Adjustment Process

is appropriate for a maximum oscillator frequency tolerance of 100 ppm (.01%), which can be obtained using a relatively inexpensive quartz crystal oscillator, but is readily scalable for other assumed tolerances.

The clock model requires the capability to slew the clock frequency over the range ±100 ppm with an intrinsic oscillator frequency error as great as ±100 ppm. Figure 11 shows the timing relationships at the extremes of the requirements envelope. Starting from an assumed offset of nominal zero and an assumed error of +100 ppm at time 0 s, the line AC shows how the uncorrected offset grows with time. Let σ represent the adjustment interval and *a* the interval AB, in seconds, and let *r* be the slew, or rate at which corrections are introduced, in ppm. For an accuracy specification of 100 μs, then

$$\sigma \le \frac{100\ \mu s}{100\ ppm} + \frac{100\ \mu s}{(r - 100)\ ppm} = \frac{r}{r - 100}\ .$$

The line AE represents the extreme case where the clock is to be steered −100 ppm. Since the slew must be complete at the end of the adjustment interval,

$$a \le \frac{(r - 200)\ \sigma}{r}.$$

These relationships are satisfied only if $r > 200$ ppm and $\sigma < 2$ s. Using $r = 300$ ppm for convenience, $\sigma = 1.5$ s and $a \le 0.5$ s. For the Unix clock model with *tick* = 10 ms, this results in the value of *tickadj* = 3 μs.

One of the assumptions made in the Unix clock model is that the period of adjustment computed in the *adjtime* call must be completed before the next call is made. If not, this results in an error message to the system log. However, in order to correct for the intrinsic frequency offset of the clock oscillator, the NTP clock model requires *adjtime* to be called at regular adjustment intervals of σ s.

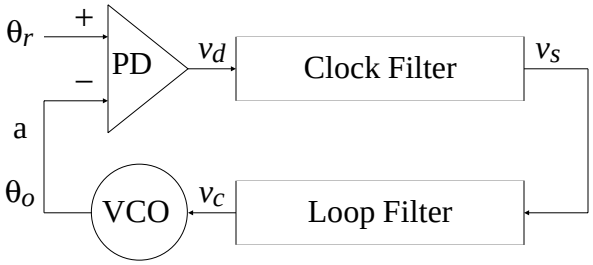

Figure 12. NTP Phase-Lock Loop (PLL) Model

Variable	Description
v_d	phase detector output
v_s	clock filter output
v_c	loop filter output
θ_r	reference phase
θ_o	VCO phase
ω_c	PLL crossover frequency

Table 9. Notation Used in PLL Analysis

Parameter	Value	Description
α	2^{-2}	VCO gain
σ	2^{2}	adjustment interval
τ	2^{6}	PLL time constant
T	2^{3}	clock-filter delay
K_f	2^{22}	frequency weight

Table 10. PLL Parameters

Using the algorithms described here and the architecture constants in the NTP specification, these adjustments will always complete.

7.2. Mathematical Model of the NTP Logical Clock

The NTP logical clock can be represented by the feedback-control model shown in Figure 12. The model consists of an adaptive-parameter, phase-lock loop (PLL), which continuously adjusts the phase and frequency of an oscillator to compensate for its intrinsic jitter, wander and drift. A mathematical analysis of this model developed along the lines of [SMI86] is presented in following sections, along with a design example useful for implementation guidance in operating-systems

environments such as Unix and Fuzzball. Table 9 summarizes the quantities ordinarily treated as variables in the model. By convention, *v* is used for internal loop variables, θ for phase, ω for frequency and τ for time. Table 10 summarizes those quantities ordinarily fixed as constants in the model. Note that these are all expressed as a power of two in order to simplify the implementation.

In Figure 12 the variable θ_r represents the phase of the reference signal and θ_o the phase of the voltage-controlled oscillator (VCO). The phase detector (PD) produces a voltage v_d representing the phase difference $\theta_r - \theta_o$. The clock filter functions as a tapped delay line, with the output v_s taken at the tap selected by the clock-filter algorithm described in the NTP specification. The loop filter, represented by the equations given below, produces a VCO correction voltage v_c, which controls the oscillator frequency and thus the phase θ_o.

The PLL behavior is completely determined by its open-loop, Laplace transfer function $G(s)$ in the s domain. Since both frequency and phase corrections are required, an appropriate design consists of a type-II PLL, which is defined by the function

$$G(s) = \frac{\omega_c^2}{\tau^2 s^2}\left(1 + \frac{\tau s}{\omega_z}\right),$$

where ω_c is the crossover frequency (also called loop gain), ω_z is the corner frequency (required for loop stability) and τ determines the PLL time constant and thus the bandwidth. While this is a first-order function and some improvement in phase noise might be gained from a higher-order function, in practice the improvement is lost due to the effects of the clock-filter delay, as described below.

The open-loop transfer function $G(s)$ is constructed by breaking the loop at point *a* on Figure 12 and computing the ratio of the output phase $\theta_o(s)$ to the reference phase $\theta_r(s)$. This function is the product of the individual transfer functions for the phase detector, clock filter, loop filter and VCO. The phase detector delivers a voltage $v_d(t) = \theta_r(t)$, so its transfer function is simply $F_d(s) = 1$, expressed in V/rad. The VCO delivers a frequency change $\Delta\omega = \frac{d\,\theta_o(t)}{dt} = \alpha v_c(t)$, where α is the VCO gain in rad/V-sec and $\theta_o(t) = \alpha \int v_c(t)\, dt$. Its transfer function is the Laplace transform of the integral, $F_o(s) = \frac{\alpha}{s}$, expressed in rad/V. The clock filter contributes a stochastic delay due to the clock-filter algorithm; but, for present purposes, this delay will be assumed a constant *T*, so its transfer function is the Laplace transform of the delay, $F_s(s) = e^{-Ts}$. Let $F(s)$ be the transfer function of the loop filter, which has yet to be determined. The open-loop transfer function $G(s)$ is the product of these four individual transfer functions:

$$G(s) = \frac{\omega_c^2}{\tau^2 s^2}\left(1 + \frac{\tau s}{\omega_z}\right) = F_d(s)F_s(s)F(s)F_o(s) = 1e^{-Ts}\,F(s)\,\frac{\alpha}{s}\,.$$

For the moment, assume that the product Ts is small, so that $e^{-Ts} \approx 1$. Making the following substitutions,

$$\omega_c^2 = \frac{\alpha}{K_f} \quad \text{and} \quad \omega_z = \frac{K_g}{K_f}$$

and rearranging yields

$$F(s) = \frac{1}{K_g\,\tau} + \frac{1}{K_f\,\tau^2 s}\,,$$

which corresponds to a constant term plus an integrating term scaled by the PLL time constant τ. This form is convenient for implementation as a sampled-data system, as described later.

With the parameter values given in Table 10, the Bode plot of the open-loop transfer function $G(s)$ consists of a –12 dB/octave line which intersects the 0-dB baseline at $\omega_c = 2^{-12}$ rad/s, together with a +6 dB/octave line at the corner frequency $\omega_z = 2^{-14}$ rad/s. The damping factor $\zeta = \frac{\omega_c}{2\omega_z} = 2$ suggests the PLL will be stable and have a large phase margin together with a low overshoot. However, if the clock-filter delay T is not small compared to the loop delay, which is approximately equal to $\frac{1}{\omega_c}$, the above analysis becomes unreliable and the loop can become unstable. With the values determined as above, T is ordinarily small enough to be neglected.

Assuming the output is taken at v_s, the closed-loop transfer function $H(s)$ is

$$H(s) \equiv \frac{v_s(s)}{\theta_r(s)} = \frac{F_d(s)e^{-Ts}}{1 + G(s)}\,.$$

If only the relative response is needed and the clock-filter delay can be neglected, $H(s)$ can be written

$$H(s) = \frac{1}{1 + G(s)} = \frac{s^2}{s^2 + \frac{\omega_c^2}{\omega_z\,\tau}s + \frac{\omega_c^2}{\tau^2}}\,.$$

For some input function $I(s)$ the output function $I(s)H(s)$ can be inverted to find the time response. Using a unit-step input $I(s) = \frac{1}{s}$ and the values determined as above, This yields a PLL risetime of about 52 minutes, a maximum overshoot of about 4.8 percent in about 1.7 hours and a settling time to within one percent of the initial offset in about 8.7 hours.

7.3. Parameter Management

A very important feature of the NTP PLL design is the ability to adapt its behavior to match the prevailing stability of the local oscillator and transmission conditions in the network. This is done using the α and τ parameters shown in Table 10. Mechanisms for doing this are described in following sections.

7.4. Adjusting VCO Gain (α)

The α parameter is determined by the maximum frequency tolerance of the local oscillator and the maximum jitter requirements of the timekeeping system. This parameter is usually an architecture constant and fixed during system operation. In the implementation model described below, the reciprocal of α, called the adjustment interval σ, determines the time between corrections of the local clock, and thus the value of α. The value of σ can be determined by the following procedure.

The maximum frequency tolerance for board-mounted, uncompensated quartz-crystal oscillators is probably in the range of 10^{-4} (100 ppm). Many if not most Internet timekeeping systems can tolerate jitter to at least the order of the intrinsic local-clock resolution, called *precision* in the NTP specification, which is commonly in the range from one to 20 ms. Assuming 10^{-3} s peak-to-peak as the most demanding case, the interval between clock corrections must be no more than $\sigma = \frac{10^{-3}}{2 \; x \; 10^{-4}} = 5$ sec. For the NTP reference model $\sigma = 4$ sec in order to allow for known features of the Unix operating-system kernel. However, in order to support future anticipated improvements in accuracy possible with faster workstations, it may be useful to decrease σ to as little as one-tenth the present value.

Note that if σ is changed, it is necessary to adjust the parameters K_f and K_g in order to retain the same loop bandwidth; in particular, the same ω_c and ω_z. Since α varies as the reciprocal of σ, if σ is changed to something other than 2^2, as in Table 10, it is necessary to divide both K_f and K_g by $\frac{\sigma}{4}$ to obtain the new values.

7.5. Adjusting PLL Bandwidth (τ)

A key feature of the type-II PLL design is its capability to compensate for the intrinsic frequency errors of the local oscillator. This requires a initial period of adaptation in order to refine the frequency estimate (see later sections of this appendix). The τ parameter determines the PLL time constant and thus the loop bandwidth, which is approximately equal to $\frac{\omega_c}{\tau}$. When operated with a relatively large bandwidth (small τ), as in the analysis above, the PLL adapts quickly to changes in the input reference signal, but has poor long term stability. Thus, it is possible to accumulate substantial errors if the system is deprived of the reference signal for an extended period. When operated with a relatively small bandwidth (large τ), the PLL adapts slowly to changes in the input

Variable	Value	Description
μ		update interval
ρ		poll interval
f		frequency error
g		phase error
h		compliance
K_h	2^{13}	compliance weight
K_s	2^4	compliance maximum
K_t	2^{14}	compliance multiplier
K_u	2^0	poll-interval factor

Table 11. Notation Used in PLL Analysis

reference signal, and may even fail to lock onto it. Assuming the frequency estimate has stabilized, it is possible for the PLL to coast for an extended period without external corrections and without accumulating significant error.

In order to achieve the best performance without requiring individual tailoring of the loop bandwidth, it is necessary to compute each value of τ based on the measured values of offset, delay and dispersion, as produced by the NTP protocol itself. The traditional way of doing this in precision timekeeping systems based on cesium clocks, is to relate τ to the Allan variance, which is defined as the mean of the first-order differences of sequential samples measured during a specified interval τ,

$$\sigma_y^2(\tau) = \frac{1}{2(N-1)} \sum_{i=1}^{N-1} [y(i+1) - y(i)]^2 ,$$

where y is the fractional frequency measured with respect to the local timescale and N is the number of samples.

In the NTP local-clock model the Allan variance (called the compliance, h in Table 11) is approximated on a continuous basis by exponentially averaging the first-order differences of the offset samples using an empirically determined averaging constant. Using somewhat ad-hoc mapping functions determined from simulation and experience, the compliance is manipulated to produce the loop time constant and update interval.

7.6. The NTP Clock Model

The PLL behavior can also be described by a set of recurrence equations, which depend upon several variables and constants. The variables and parameters used in these equations are shown in Tables 9, 10 and 11. Note the use of powers of two, which facilitates implementation using arithmetic shifts and avoids the requirement for a multiply/divide capability.

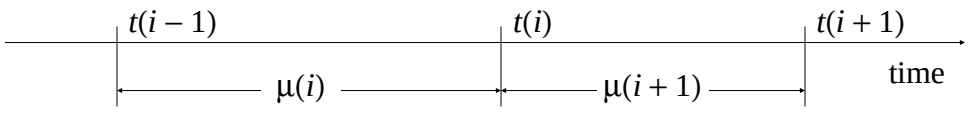

Figure 13. Timing Intervals

A capsule overview of the design may be helpful in understanding how it operates. The logical clock is continuously adjusted in small increments at fixed intervals of σ. The increments are determined while updating the variables shown in Tables 9 and 11, which are computed from received NTP messages as described in the NTP specification. Updates computed from these messages occur at discrete times as each is received. The intervals μ between updates are variable and can range up to about 17 minutes. As part of update processing the compliance h is computed and used to adjust the PLL time constant τ. Finally, the update interval ρ for transmitted NTP messages is determined as a fixed multiple of τ.

Updates are numbered from zero, with those in the neighborhood of the ith update shown in Figure 13. All variables are initialized at $i = 0$ to zero, except the time constant $\tau(0) = \tau$, poll interval $\mu(0) = \tau$ (from Table 10) and compliance $h(0) = K_S$. After an interval $\mu(i)$ ($i > 0$) from the previous update the ith update arrives at time $t(i)$ including the time offset $v_S(i)$. Then, after an interval $\mu(i + 1)$ the i+1th update arrives at time $t(i + 1)$ including the time offset $v_S(i + 1)$. When the update $v_S(i)$ is received, the frequency error $f(i + 1)$ and phase error $g(i + 1)$ are computed:

$$f(i+1) = f(i) + \frac{\mu(i) v_S(i)}{\tau(i)^2}, \quad g(i+1) = \frac{v_S(i)}{\tau(i)} .$$

Note that these computations depend on the value of the time constant $\tau(i)$ and poll interval $\mu(i)$ previously computed from the i–1th update. Then, the time constant for the next interval is computed from the current value of the compliance $h(i)$

$$\tau(i+1) = \max[K_S - |h(i)|, 1] .$$

Next, using the new value of τ, called τ' to avoid confusion, the poll interval is computed

$$\rho(i+1) = K_u \tau' .$$

Finally, the compliance $h(i + 1)$ is recomputed for use in the i+1th update:

$$h(i+1) = h(i) + \frac{K_t \tau' v_S(i) - h(i)}{K_h} .$$

The factor τ' in the above has the effect of adjusting the bandwidth of the PLL as a function of compliance. When the compliance has been low over some relatively long period, τ' is increased and the bandwidth is decreased. In this mode small timing fluctuations due to jitter in the network

are suppressed and the PLL attains the most accurate frequency estimate. On the other hand, if the compliance becomes high due to greatly increased jitter or a systematic frequency offset, τ' is decreased and the bandwidth is increased. In this mode the PLL is most adaptive to transients which can occur due to reboot of the system or a major timing error. In order to maintain optimum stability, the poll interval ρ is varied directly with τ.

A model suitable for simulation and parameter refinement can be constructed from the above recurrence relations. It is convenient to set the temporary variable $a = g(i + 1)$. At each adjustment interval σ the quantity $\frac{a}{K_g} + \frac{f(i+1)}{K_f}$ is added to the local-clock phase and the quantity $\frac{a}{K_g}$ is subtracted from a. For convenience, let n be the greatest integer in $\frac{\mu(i)}{\sigma}$; that is, the number of adjustments that occur in the ith interval. Thus, at the end of the ith interval just before the i+1th update, the VCO control voltage is:

$$v_c(i+1) = v_c(i) + [1 - (1 - \frac{1}{K_g})^n]\, g(i+1) + \frac{n}{K_f} f(i+1) \; .$$

Detailed simulation of the NTP PLL with the values specified in Tables 9, 10 and 11 and the clock filter described in the NTP specification results in the following characteristics: For a 100-ms phase change the loop reaches zero error in 39 minutes, overshoots 7 ms at 54 minutes and settles to less than 1 ms in about six hours. For a 50-ppm frequency change the loop reaches 1 ppm in about 16 hours and 0.1 ppm in about 26 hours. When the magnitude of correction exceeds a few milliseconds or a few ppm for more than a few updates, the compliance begins to increase, which causes the loop time constant and update interval to decrease. When the magnitude of correction falls below about 0.1 ppm for a few hours, the compliance begins to decrease, which causes the loop time constant and update interval to increase. The effect is to provide a broad capture range exceeding 4 s per day, yet the capability to resolve oscillator skew well below 1 ms per day. These characteristics are appropriate for typical crystal-controlled oscillators with or without temperature compensation or oven control.

8. Appendix H. Analysis of Errors and Correctness Principles

8.1. Introduction

This appendix contains an analysis of errors arising in the generation and processing of NTP timestamps and the determination of delays and offsets. It establishes error bounds as a function of measured roundtrip delay and dispersion to the root (primary reference source) of the synchronization subnet. It also discusses correctness assertions about these error bounds and the time-transfer, filtering and selection algorithms used in NTP.

The notation $w = [u, v]$ in the following describes the interval in which u is the lower limit and v the upper limit, inclusive. Thus, $u = \min(w) \le v = \max(w)$, and for scalar a, $w + a = [u + a, v + a]$. Table 12 shows a summary of other notation used in the analysis. The notation $< x >$ designates the (infinite) average of x, which is usually approximated by an exponential average, while the notation $\hat{x}$ designates an estimator for x. The lower-case Greek letters θ, δ and ε are used to designate measurement data for the local clock to a peer clock, while the upper-case Greek letters Θ, Δ and E are used to designate measurement data for the local clock relative to the primary reference source at the root of the synchronization subnet. Exceptions will be noted as they arise.

8.2. Timestamp Errors

The standard second (1 s) is defined as "9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the cesium-133 atom" [ALL74b], which implies a granularity of about 1.1×10^{-10} s. Other intervals can be determined as rational multiples of 1 s. While NTP time has an inherent resolution of about 2.3×10^{-10} s, local clocks ordinarily have resolutions much worse than this, so the inherent error in resolving NTP time relative to the 1 s can be neglected.

Variable	Description
r	reading error
ρ	max reading error
f	frequency error
φ	max frequency error
θ, Θ	clock offset
δ, Δ	roundtrip delay
ε, E	error/dispersion
t	time
τ	time interval
T	NTP timestamp
s	clock divider increment

Table 12. Notation Used in Error Analysis

In this analysis the local clock is represented by a counter/divider which increments at intervals of *s* seconds and is driven by an oscillator which operates at frequency $f_c = \frac{n}{s}$ for some integer *n*. A timestamp $T(t)$ is determined by reading the clock at an arbitrary time *t* (the argument *t* will be usually omitted for conciseness). Strictly speaking, *s* is not known exactly, but can be assumed bounded from above by the maximum reading error ρ. The reading error itself is represented by the random variable *r* bounded by the interval $[-\rho, 0]$, where ρ depends on the particular clock implementation. Since the intervals between reading the same clock are almost always independent of and much larger than *s*, successive readings can be considered independent and identically distributed. The frequency error of the clock oscillator is represented by the random variable *f* bounded by the interval $[-\varphi, \varphi]$, where φ represents the maximum frequency tolerance of the oscillator throughout its service life. While *f* for a particular clock is a random variable with respect to the population of all clocks, for any one clock it ordinarily changes only slowly with time and can usually be assumed a constant for that clock. Thus, an NTP timestamp can be represented by the random variable *T*:

$$T = t + r + f\tau ,$$

where *t* represents a clock reading, τ represents the time interval since this reading and minor approximations inherent in the measurement of τ are neglected.

In order to assess the nature and expected magnitude of timestamp errors and the calculations based on them, it is useful to examine the characteristics of the probability density functions (pdf) $p_r(x)$ and $p_f(x)$ for *r* and *f* respectively. Assuming the clock reading and counting processes are independent, the pdf for *r* is uniform over the interval $[-\rho, 0]$. With conventional manufacturing processes and temperature variations the pdf for *f* can be approximated by a truncated, zero-mean Gaussian distribution with standard deviation σ. In conventional manufacturing processes σ is maneuvered so that the fraction of samples rejected outside the interval $[-\varphi, \varphi]$ is acceptable. The pdf for the total timestamp error $\varepsilon(x)$ is thus the sum of the *r* and *f* contributions, computed as

$$\varepsilon(x) = \int_{-\infty}^{\infty} p_r(t) p_f(x - t) dt ,$$

which appears as a bell-shaped curve, symmetric about $-\frac{\rho}{2}$ and bounded by the interval

$$[\min(r) + \min(f\tau), \max(r) + \max(f\tau)] = [-\rho - \varphi\tau, \varphi\tau] .$$

Since *f* changes only slowly over time for any single clock,

$$\varepsilon \equiv [\min(r) + f\tau, \max(r) + f\tau] = [-\rho, 0] + f\tau ,$$

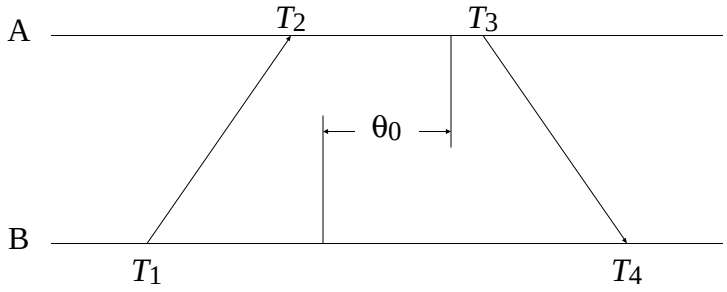

Figure 14. Measuring Delay and Offset

where ε without argument designates the interval and ε(*x*) designates the pdf. In the following development subscripts will be used on various quantities to indicate to which entity or timestamp the quantity applies. Occasionally, ε will be used to designate an absolute maximum error, rather than the interval, but the distinction will be clear from context.

8.3. Measurement Errors

In NTP the roundtrip delay and clock offset between two peers *A* and *B* are determined by a procedure in which timestamps are exchanged via the network paths between them. The procedure involves the four most recent timestamps numbered as shown in Figure 14, where the θ_0 represents the true clock offset of peer *B* relative to peer *A*. The T_1 and T_4 timestamps are determined relative to the *A* clock, while the T_2 and T_3 timestamps are determined relative to the *B* clock. The measured roundtrip delay δ and clock offset θ of *B* relative to *A* are given by

$$\delta = (T_4 - T_1) - (T_3 - T_2) \quad \text{and} \quad \theta = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} .$$

The errors inherent in determining the timestamps T_1, T_2, T_3 and T_4 are, respectively,

$$\varepsilon_1 = [-\rho_A, 0], \ \varepsilon_2 = [-\rho_B, 0], \ \varepsilon_3 = [-\rho_B, 0] + f_B(T_3 - T_2), \ \varepsilon_4 = [-\rho_A, 0] + f_A(T_4 - T_1) .$$

For specific peers *A* and *B*, where f_A and f_B can be considered constants, the interval containing the maximum error inherent in determining δ is given by

$$\begin{aligned}[\min(\varepsilon_4) - \max(\varepsilon_1) - \max(\varepsilon_3) + \min(\varepsilon_2), \max(\varepsilon_4) - \min(\varepsilon_1) - \min(\varepsilon_3) + \max(\varepsilon_2)] \\ = [-\rho_A - \rho_B, \rho_A + \rho_B] + f_A(T_4 - T_1) - f_B(T_3 - T_2) .\end{aligned}$$

In the NTP local clock model the residual frequency errors f_A and f_B are minimized through the use of a type-II phase-lock loop (PLL). Under most conditions these errors will be small and can be ignored. The pdf for the remaining errors is symmetric, so that $\hat{\delta} = \langle\delta\rangle$ is an unbiased maximum-likelihood estimator for the true roundtrip delay, independent of the particular values of ρ_A and ρ_B.

However, in order to reliably bound the errors under all conditions of component variation and operational regimes, the design of the PLL and the tolerance of its intrinsic oscillator must be

controlled so that it is not possible under any circumstances for f_A or f_B to exceed the bounds $[-\varphi_A, \varphi_A]$ or $[-\varphi_B, \varphi_B]$, respectively. Setting $\rho = \max(\rho_A, \rho_B)$ for convenience, the absolute maximum error ε_δ inherent in determining roundtrip delay δ is given by

$$\varepsilon_\delta \equiv \rho + \varphi_A(T_4 - T_1) + \varphi_B(T_3 - T_2) ,$$

neglecting residuals.

As in the case for δ, where f_A and f_B can be considered constants, the interval containing the maximum error inherent in determining θ is given by

$$\frac{[\min(\varepsilon_2) - \max(\varepsilon_1) + \min(\varepsilon_3) - \max(\varepsilon_4), \max(\varepsilon_2) - \min(\varepsilon_1) + \max(\varepsilon_3) - \min(\varepsilon_4)]}{2}$$
$$= [-\rho_B, \rho_A] + \frac{f_B(T_3 - T_2) - f_A(T_4 - T_1)}{2} .$$

Under most conditions the errors due to f_A and f_B will be small and can be ignored. If $\rho_A = \rho_B = \rho$; that is, if both the *A* and *B* clocks have the same resolution, the pdf for the remaining errors is symmetric, so that $\hat{\theta} = \langle\theta\rangle$ is an unbiased maximum-likelihood estimator for the true clock offset θ_0, independent of the particular value of ρ. If $\rho_A \neq \rho_B$, $\langle\theta\rangle$ is not an unbiased estimator; however, the bias error is in the order of

$$\frac{\rho_A - \rho_B}{2} .$$

and can usually be neglected.

Again setting $\rho = \max(\rho_A, \rho_B)$ for convenience, the absolute maximum error ε_θ inherent in determining clock offset θ is given by

$$\varepsilon_\theta \equiv \frac{\rho + \varphi_A(T_4 - T_1) + \varphi_B(T_3 - T_2)}{2} .$$

8.4. Network Errors

In practice, errors due to stochastic network delays usually dominate. In general, it is not possible to characterize network delays as a stationary random process, since network queues can grow and shrink in chaotic fashion and arriving customer traffic is frequently bursty. However, it is a simple exercise to calculate bounds on clock offset errors as a function of measured delay. Let $T_2 - T_1 = a$ and $T_3 - T_4 = b$. Then,

$$\delta = a - b \quad \text{and} \quad \theta = \frac{a + b}{2} .$$

The true offset of *B* relative to *A* is called θ_0 in Figure 14. Let *x* denote the actual delay between the departure of a message from *A* and its arrival at *B*. Therefore, $x + \theta_0 = T_2 - T_1 \equiv a$. Since *x* must be

positive in our universe, $x = a - \theta_0 \geq 0$, which requires $\theta_0 \leq a$. A similar argument requires that $b \leq \theta_0$, so surely $b \leq \theta_0 \leq a$. This inequality can also be expressed

$$b = \frac{a+b}{2} - \frac{a-b}{2} \leq \theta_0 \leq \frac{a+b}{2} + \frac{a-b}{2} = a ,$$

which is equivalent to

$$\theta - \frac{\delta}{2} \leq \theta_0 \leq \theta + \frac{\delta}{2} .$$

In the previous section bounds on delay and offset errors were determined. Thus, the inequality can be written

$$\theta - \varepsilon_\theta - \frac{\delta + \varepsilon_\delta}{2} \leq \theta_0 \leq \theta + \varepsilon_\theta + \frac{\delta + \varepsilon_\delta}{2} ,$$

where ε_θ is the maximum offset error and ε_δ is the maximum delay error derived previously. The quantity

$$\varepsilon = \varepsilon_\theta + \frac{\varepsilon_\delta}{2} = \rho + \varphi_A(T_4 - T_1) + \varphi_B(T_3 - T_2) ,$$

called the peer dispersion, defines the maximum error in the inequality. Thus, the correctness interval I can be defined as the interval

$$I = [\theta - \frac{\delta}{2} - \varepsilon, \theta + \frac{\delta}{2} + \varepsilon] ,$$

in which the clock offset $C = \theta$ is the midpoint. By construction, the true offset θ_0 must lie somewhere in this interval.

8.5. Inherited Errors

As described in the NTP specification, the NTP time server maintains the local clock Θ, together with the root roundtrip delay Δ and root dispersion E relative to the primary reference source at the root of the synchronization subnet. The values of these variables are either included in each update message or can be derived as described in the NTP specification. In addition, the protocol exchange and clock-filter algorithm provide the clock offset θ and roundtrip delay δ of the local clock relative to the peer clock, as well as various error accumulations as described below. The following discussion establishes how errors inherent in the time-transfer process accumulate within the subnet and contribute to the overall error budget at each server.

An NTP measurement update includes three parts: clock offset θ, roundtrip delay δ and maximum error or dispersion ε of the local clock relative to a peer clock. In case of a primary clock update, these values are usually all zero, although ε can be tailored to reflect the specified maximum error

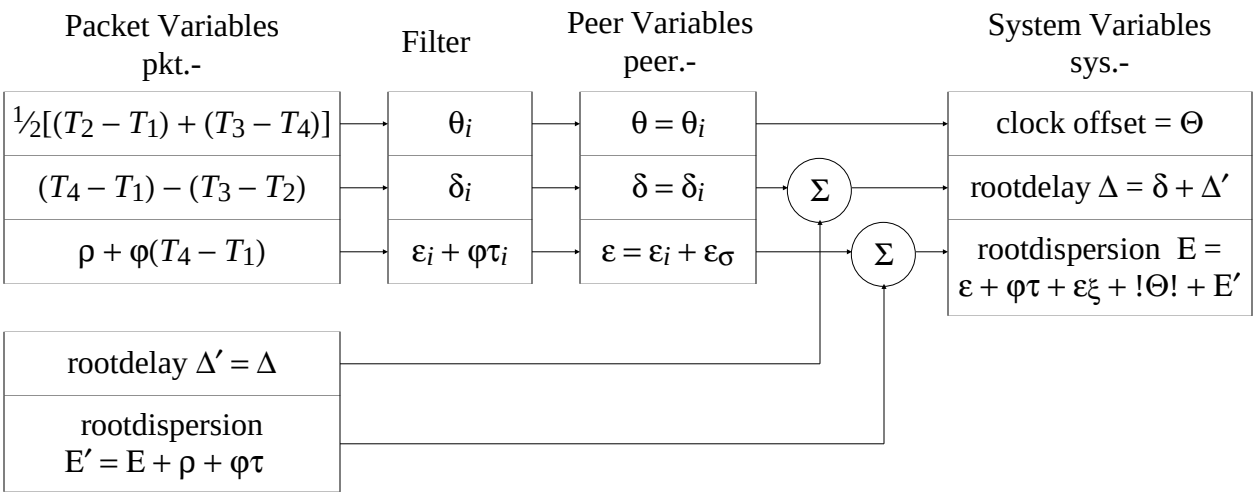

Figure 15. Error Accumulations

of the primary reference source itself. In other cases θ and δ are calculated directly from the four most recent timestamps, as described in the NTP specification. The dispersion ε includes the following contributions:

1. Each time the local clock is read a reading error is incurred due to the finite granularity or precision of the implementation. This is called the measurement dispersion ρ.

2. Once an offset is determined, an error due to frequency offset or skew accumulates with time. This is called the skew dispersion $\varphi\tau$, where φ represents the skew-rate constant ($\frac{\text{NTP.MAXSKEW}}{\text{NTP.MAXAGE}}$ in the NTP specification) and τ is the interval since the dispersion was last updated.

3 When a series of offsets are determined at regular intervals and accumulated in a window of samples, as in the NTP clock-filter algorithm, the (estimated) additional error due to offset sample variance is called the filter dispersion ε_σ.

4. When a number of peers are considered for synchronization and two or more are determined to be correctly synchronized to a primary reference source, as in the NTP clock-selection algorithm, the (estimated) additional error due to offset sample variance is called the selection dispersion $\varepsilon\xi$.

Figure 15 shows how these errors accumulate in the ordinary course of NTP processing. Received messages from a single peer are represented by the packet variables. From the four most recent timestamps T_1, T_2, T_3 and T_4 the clock offset and roundtrip delay sample for the local clock relative to the peer clock are calculated directly. Included in the message are the root roundtrip delay Δ' and root dispersion E' of the peer itself; however, before sending, the peer adds the measurement

dispersion ρ and skew dispersion $\varphi\tau$, where these quantities are determined by the peer and τ is the interval according to the peer clock since its clock was last updated.

The NTP clock-filter procedure saves the most recent samples θ_i and δ_i in the clock filter as described in the NTP specification. The quantities ρ and φ characterize the local clock maximum reading error and frequency error, respectively. Each sample includes the dispersion $\varepsilon_i = \rho + \varphi(T_4 - T_1)$, which is set upon arrival. Each time a new sample arrives all samples in the filter are updated with the skew dispersion $\varphi\tau_i$, where τ_i is the interval since the last sample arrived, as recorded in the variable peer.update. The clock-filter algorithm determines the selected clock offset θ (peer.offset), together with the associated roundtrip delay δ (peer.delay) and filter dispersion ε_σ, which is added to the associated sample dispersion ε_i to form the peer dispersion ε (peer.dispersion).

The NTP clock-selection procedure selects a single peer to become the synchronization source as described in the NTP specification. The operation of the algorithm determines the final clock offset Θ (local clock), roundtrip delay Δ (sys.rootdelay) and dispersion E (sys.rootdispersion) relative to the root of the synchronization subnet, as shown in Figure 15. Note the inclusion of the selected peer dispersion and skew accumulation since the dispersion was last updated, as well as the select dispersion $\varepsilon\xi$ computed by the clock-select algorithm itself. Also, note that, in order to preserve overall synchronization subnet stability, the final clock offset Θ is in fact determined from the offset of the local clock relative to the peer clock, rather than the root of the subnet. Finally, note that the packet variables Δ' and E′ are in fact determined from the latest message received, not at the precise time the offset selected by the clock-filter algorithm was determined. Minor errors arising due to these simplifications will be ignored. Thus, the total dispersion accumulation relative to the root of the synchronization subnet is

$$E = \varepsilon + \varphi\tau + \varepsilon\xi + |\Theta| + E' ,$$

where τ is the time since the peer variables were last updated and $|\Theta|$ is the initial absolute error in setting the local clock.

The three values of clock offset, roundtrip delay and dispersion are all additive; that is, if Θ_i, Δ_i and E_i represent the values at peer *i* relative to the root of the synchronization subnet, the values

$$\Theta_j(t) \equiv \Theta_i + \theta_j(t) , \quad \Delta_j(t) \equiv \Delta_i + \delta_j , \quad E_j(t) \equiv E_i + \varepsilon_i + \varepsilon_j(t) ,$$

represent the clock offset, roundtrip delay and dispersion of peer *j* at time *t*. The time dependence of $\theta_j(t)$ and $\varepsilon_j(t)$ represents the local-clock correction and dispersion accumulated since the last update was received from peer *i*, while the term ε_i represents the dispersion accumulated by peer *i* from the time its clock was last set until the latest update was sent to peer *j*. Note that, while the offset of the local clock relative to the peer clock can be determined directly, the offset relative to the root of the synchronization subnet is not directly determinable, except on a probabilistic basis and within the bounds established in this and the previous section.

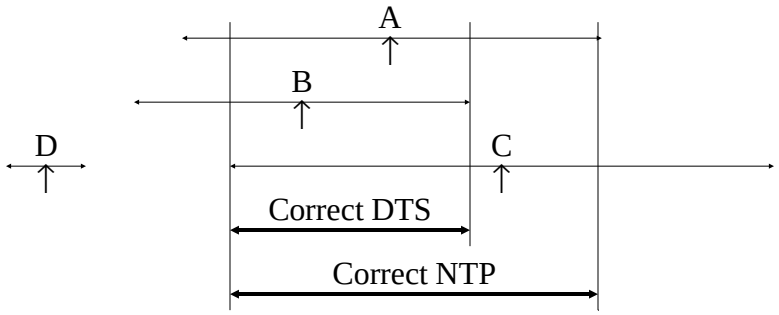

Figure 16. Confidence Intervals and Intersections

The NTP synchronization subnet topology is that of a tree rooted at the primary server(s). Thus, there is an unbroken path from every time server to the primary reference source. Accuracy and stability are proportional to synchronization distance Λ, defined as

$$\Lambda \equiv E + \frac{\Delta}{2} .$$

The selection algorithm favors the minimum-distance paths and thus maximizes accuracy and stability. Since Θ_0, Δ_0 and E_0 are all zero, the sum of the clock offsets, roundtrip delays and dispersions of each server along the minimum-distance path from the root of the synchronization subnet to a given server *i* are the clock offset Θ_i, roundtrip delay Δ_i and dispersion E_i inherited by and characteristic of that server.

8.6. Correctness Principles

In order to minimize the occurrence of errors due to incorrect clocks and maximize the reliability of the service, NTP relies on multiple peers and disjoint peer paths whenever possible. In the previous development it was shown that, if the primary reference source at the root of the synchronization subnet is in fact a correct clock, then the true offset θ_0 relative to that clock must be contained in the interval

$$[\Theta - \Lambda, \Theta + \Lambda] \equiv [\Theta - E - \frac{\Delta}{2}, \Theta + E + \frac{\Delta}{2}] .$$

When a number of clocks are involved, it is not clear beforehand which are correct and which are not; however, as cited previously, there are a number of techniques based on clustering and filtering principles which yield a high probability of detecting and discarding incorrect clocks. Marzullo and Owicki [MAR85] devised an algorithm designed to find an appropriate interval containing the correct time given the confidence intervals of *m* clocks, of which no more than *f* are considered incorrect. The algorithm finds the smallest single intersection containing all points in at least $m - f$ of the given confidence intervals.

Figure 16 illustrates the operation of this algorithm with a scenario involving four clocks *A*, *B*, *C* and *D*, with the calculated time (shown by the ↑ symbol) and confidence interval shown for each.

These intervals are computed as described in previous sections of this appendix. For instance, any point in the *A* interval may possibly represent the actual time associated with that clock. If all clocks are correct, there must exist a nonempty intersection including all four intervals; but, clearly this is not the case in this scenario. However, if it is assumed that one of the clocks is incorrect (e.g., *D*), it might be possible to find a nonempty intersection including all but one of the intervals. If not, it might be possible to find a nonempty intersection including all but two of the intervals and so on.

The algorithm proposed by DEC for use in the Digital Time Service [DEC89] is based on these principles. For the scenario illustrated in Figure 16, it computes the interval for $m = 4$ clocks, three of which turn out to be correct and one not. The low endpoint of the intersection is found as follows. A variable f is initialized with the number of presumed incorrect clocks, in this case zero, and a counter i is initialized at zero. Starting from the lowest endpoint, the algorithm increments i at each low endpoint, decrements i at each high endpoint, and stops when $i \geq m - f$. The counter records the number of intersections and thus the number of presumed correct clocks. In the example the counter never reaches four, so f is increased by one and the procedure is repeated. This time the counter reaches three and stops at the low endpoint of the intersection marked DTS. The upper endpoint of this intersection is found using a similar procedure.

This algorithm will always find the smallest single intersection containing points in at least one of the original $m - f$ confidence intervals as long as the number of incorrect clocks is less than half the total $f < \frac{m}{2}$. However, some points in the intersection may not be contained in all $m - f$ of the original intervals; moreover, some or all of the calculated times (such as for *C* in Figure 16) may lie outside the intersection. In the NTP clock-selection procedure the above algorithm is modified so as to include at least $m - f$ of the calculated times. In the modified algorithm a counter c is initialized at zero. When starting from either endpoint, c is incremented at each calculated time; however, neither f nor c are reset between finding the low and high endpoints of the intersection. If after both endpoints have been found $c > f$, f is increased by one and the entire procedure is repeated. The revised algorithm finds the smallest intersection of $m - f$ intervals containing at least $m - f$ calculated times. As shown in Figure 16, the modified algorithm produces the intersection marked NTP and including the calculated time for *C*.

In the NTP clock-selection procedure the peers represented by the clocks in the final intersection, called the survivors, are placed on a candidate list. In the remaining steps of the procedure one or more survivors may be discarded from the list as outlyers. Finally, the clock-combining algorithm described in Appendix F provides a weighted average of the remaining survivors based on synchronization distance. The resulting estimates represent a synthetic peer with offset between the maximum and minimum offsets of the remaining survivors. This defines the clock offset Θ, total roundtrip total delay Δ and total dispersion E which the local clock inherits. In principle, these values could be included in the time interface provided by the operating system to the user, so that the user could evaluate the quality of indications directly.

9. Appendix I. Selected C-Language Program Listings

Following are C-language program listings of selected algorithms described in the NTP specification. While these have been tested as part of a software simulator using data collected in regular operation, they do not necessarily represent a standard implementation, since many other implementations could in principle conform to the NTP specification.

9.1. Common Definitions and Variables

The following definitions are common to all procedures and peers.

```
#define NMAX 40                 /* max clocks */
#define FMAX 8                  /* max filter size */
#define HZ 1000                 /* clock rate */
#define MAXSTRAT 15             /* max stratum */
#define MAXSKEW 1               /* max skew error per MAXAGE */
#define MAXAGE 86400            /* max clock age */
#define MAXDISP 16              /* max dispersion */
#define MINCLOCK 3              /* min survivor clocks */
#define MAXCLOCK 10             /* min candidate clocks */
#define FILTER .5               /* filter weight */
#define SELECT .75              /* select weight */
```

The folowing are peer state variables (one set for each peer).

```
double filtp[NMAX][FMAX];       /* offset samples */
double fildp[NMAX][FMAX];       /* delay samples */
double filep[NMAX][FMAX];       /* dispersion samples */
double tp[NMAX];                /* offset */
double dp[NMAX];                /* delay */
double ep[NMAX];                /* dispersion */
double rp[NMAX];                /* last offset */
double utc[NMAX];               /* update tstamp */
int st[NMAX];                   /* stratum */
```

The following are system state variables and constants.

```
double rho = 1./HZ;             /* max reading error */
double phi = MAXSKEW/MAXAGE;    /* max skew rate */
double bot, top;                /* confidence interval limits */
double theta;                   /* clock offset */
double delta;                   /* roundtrip delay */
double epsil;                   /* dispersion */
double tstamp;                  /* current time */
int source;                     /* clock source */
int n1, n2;                     /* min/max clock ids */
```

The folowing are temporary lists shared by all peers and procedures.

```
double list[3*NMAX];                    /* temporary list*/
int index[3*NMAX];                      /* index list */
```

9.2. Clock–Filter Algorithm

```
/*
   clock filter algorithm

   n = peer id, offset = sample offset, delay = sample delay, disp = sample dispersion;
   computes tp[n] = peer offset, dp[n] = peer delay, ep[n] = peer dispersion
 */

void filter(int n, double offset, double delay, double disp) {

        int i, j, k, m;                 /* int temps */
        double x;                       /* double temps */

        for (i = FMAX–1; i > 0; i– –) {         /* update/shift filter */
                filtp[n][i] = filtp[n][i–1]; fildp[n][i] = fildp[n][i–1];
                filep[n][i] = filep[n][i–1]+phi*(tstamp–utc[n]);
                }
        utc[n] = tstamp; filtp[n][0] = offset–tp[0]; fildp[n][0] = delay; filep[n][0] = disp;
        m = 0;                          /* construct/sort temp list */
        for (i = 0; i < FMAX; i++) {
                if (filep[n][i] >= MAXDISP) continue;
                list[m] = filep[n][i]+fildp[n][i]/2.; index[m] = i;
                for (j = 0; j < m; j++) {
                        if (list[j] > list[m]) {
                                x = list[j]; k = index[j]; list[j] = list[m]; index[j] = index[m];
                                list[m] = x; index[m] = k;
                                }
                        }
                m = m+1;
                }

        if (m <= 0) ep[n] = MAXDISP;            /* compute filter dispersion */
        else {
                ep[n] = 0;
                for (i = FMAX–1; i >= 0; i– –) {
                        if (i < m) x = fabs(filtp[n][index[0]]–filtp[n][index[i]]);
                        else x = MAXDISP;
                        ep[n] = FILTER*(ep[n]+x);
                        }
                i = index[0]; ep[n] = ep[n]+filep[n][i]; tp[n] = filtp[n][i]; dp[n] = fildp[n][i];
```

```
            }
        return;
        }
```

9.3. Interval Intersection Algorithm

```
/*
   compute interval intersection

   computes bot = lowpoint, top = highpoint (bot > top if no intersection)
*/

void dts() {

        int f;                                  /* intersection ceiling */
        int end;                                /* endpoint counter */
        int clk;                                /* falseticker counter */
        int i, j, k, m, n;                      /* int temps */
        double x, y;                            /* double temps */

        m = 0; i = 0;
        for (n = n1; n <= n2; n++) { /* construct endpoint list */
                if (ep[n] >= MAXDISP) continue;
                m = m+1;
                list[i] = tp[n]–dist(n); index[i] = –1; /* lowpoint */
                for (j = 0; j < i; j++) {
                        if ((list[j] > list[i]) || ((list[j] == list[i]) && (index[j] > index[i]))) {
                                x = list[j]; k = index[j]; list[j] = list[i]; index[j] = index[i];
                                list[i] = x; index[i] = k;
                                }
                        }
                i = i+1;

                list[i] = tp[n]; index[i] = 0;          /* midpoint */
                for (j = 0; j < i; j++) {
                        if ((list[j] > list[i]) || ((list[j] == list[i]) && (index[j] > index[i]))) {
                                x = list[j]; k = index[j]; list[j] = list[i]; index[j] = index[i];
                                list[i] = x; index[i] = k;
                                }
                        }
                i = i+1;

                list[i] = tp[n]+dist(n); index[i] = 1;  /* highpoint */
                for (j = 0; j < i; j++) {
                        if ((list[j] > list[i]) || ((list[j] == list[i]) && (index[j] > index[i]))) {
                                x = list[j]; k = index[j]; list[j] = list[i]; index[j] = index[i];
```

```
                        list[i] = x; index[i] = k;
                        }
                }
        i = i+1;
        }

    if (m <= 0) return;
    for (f = 0; f < m/2; f++) {                 /* find intersection */
        clk = 0; end = 0;                       /* lowpoint */
        for (j = 0; j < i; j++) {
            end = end–index[j]; bot = list[j];
            if (end >= (m–f)) break;
            if (index[j] == 0) clk = clk+1;
            }
        end = 0;                                /* highpoint */
        for (j = i–1; j >= 0; j– –) {
            end = end+index[j]; top = list[j];
            if (end >= (m–f)) break;
            if (index[j] == 0) clk = clk+1;
            }
        if (clk <= f) break;
        }
    return;
    }
```

9.4. Clock–Selection Algorithm

```
/*
  select best subset of clocks in candidate list

  bot = lowpoint, top = highpoint; constructs index = candidate index list,
  m = number of candidates, source = clock source,
  theta = clock offset, delta = roundtrip delay, epsil = dispersion
*/

void select() {

    double xi;                                  /* max select dispersion */
    double eps;                                 /* min peer dispersion */
    int i, j, k, n;                             /* int temps */
    double x, y, z;                             /* double temps */

    m = 0;
    for (n = n1; n <= n2; n++) { /* make/sort candidate list */
        if ((st[n] > 0) && (st[n] < MAXSTRAT) && (tp[n] >= bot) && (tp[n] <= top)) {
            list[m] = MAXDISP*st[n]+dist(n); index[m] = n;
```

```
                for (j = 0; j < m; j++) {
                        if (list[j] > list[m]) {
                                x = list[j]; k = index[j]; list[j] = list[m]; index[j] = index[m];
                                list[m] = x; index[m] = k;
                                }
                        }
                m = m+1;
                }
        }
if (m <= 0) {
        source = 0; return;
        }
if (m > MAXCLOCK) m = MAXCLOCK;

while (1) {                                     /* cast out falsetickers */
        xi = 0.; eps = MAXDISP;
        for (j = 0; j < m; j++) {
                x = 0.;
                for (k = m–1; k >= 0; k– –)
                        x = SELECT*(x+fabs(tp[index[j]]–tp[index[k]]));
                if (x > xi) {
                        xi = x; i = j;          /* max(xi) */
                        }
                x = ep[index[j]]+phi*(tstamp–utc[index[j]]);
                if (x < eps) eps = x;           /* min(eps) */
                }
        if ((xi <= eps) || (m <= MINCLOCK)) break;
        if (index[i] == source) source = 0;
        for (j = i; j < m–1; j++) index[j] = index[j+1];
        m = m–1;
        }

i = index[0];                                   /* declare winner */
if (source != i)
        if (source == 0) source = i;
        else if (st[i] < st[source]) source = i;
theta = combine(); delta = dp[i]; epsil = ep[i]+phi*(tstamp–utc[i])+xi;
return;
}
```

9.5. Clock–Combining Procedure

```
/*
  compute weighted ensemble average
```

```
   index = candidate index list, m = number of candidates; returns combined clock offset
*/

double combine() {

        int i;                                  /* int temps */
        double x, y, z;                         /* double temps */

        z = 0. ; y = 0.;
        for (i = 0; i < m; i++) {               /* compute weighted offset */
                j = index[i]; x = dist(j)); z = z+tp[j]/x; y = y+1./x;
                }
        return z/y;                             /* normalize */
        }
```

9.6. Subroutine to Compute Synchronization Distance

```
/*
   compute synchronization distance

   n = peer id; returns synchronization distance
 */

double dist(int n) {

        return ep[n]+phi*(tstamp–utc[n])+fabs(dp[n])/2.;
        }
```

Security considerations
see Section 3.6 and Appendix C

Author’s address
David L. Mills
Electrical Engineering Department
University of Delaware
Newark, DE 19716
Phone (302) 451–8247
EMail mills@udel.edu