Notes on Configuring NTP and Setting up a NTP Subnet

 From NBS Special Publication 432 (out of print)

Introduction

This document is a collection of notes concerning the use of ntpd and relatedprograms, and on coping with the Network Time Protocol (NTP) in general. It is a major rewrite and update of an earlier document written by Dennis Ferguson of the University of Toronto and includes many changes and additions resulting from the NTP Version 3 specification and new Version 4 implementation features. It supersedes earlier documents, which should no longer be used for new configurations.

`ntpd` includes a complete implementation of the NTP Version 3 specification, as defined in:

- Mills, D.L. Network Time Protocol (Version 3) specification, implementation and analysis. Network Working Group Report RFC-1305, University of Delaware, March 1992, 113 pp. Abstract: PostScript | PDF, Body: PostScript | PDF, Appendices: PostScript | PDF

Additional features have are described for NTP Version 4. It also retains compatibility with both NTP Version 2, as defined in RFC-1119, and NTP Version 1, as defined in RFC-1059, although this compatibility is sometimes strained and only semiautomatic. In order to support in principle the ultimate precision of about 232 picoseconds in the NTP specification, `ntpd` uses NTP timestamp format for external communication and double precision floating point arithmetic internally. `ntpd` fully implements NTP Versions 2 and 3 authentication and in addition Version 4 autokey. It supports the NTP mode-6 control message facility along with a private mode-7 control- message facility used to remotely reconfigure the system and monitor a considerable amount of internal detail. As extensions to the specification, a flexible address-and-mask restriction facility has been included.

The code is biased towards the needs of a busy time server with numerous, often hundreds, of clients and other servers. Tables are hashed to allow efficient handling of many associations, though at the expense of additional overhead when the number of associations is small. Many fancy features have been included to permit efficient management and monitoring of a busy primary server, features which are probably excess baggage for a high stratum client. In such cases, a stripped-down version of the protocol, the Simple Network Time Protocol (SNTP) can be used. SNTP and NTP servers and clients can interwork in most situations, as described in: Mills, D.L. Simple Network Time Protocol (SNTP). Network Working Group Report RFC-2030, University of Delaware, October 1996, 14 pp. (ASCII).

The code was written with near demonic attention to details which can affect precision and as a consequence should be able to make good use of high performance, special purpose hardware such as precision oscillators and radio clocks. The present code supports a number of radio clocks, including those for the WWV, CHU, WWVB, MSF, DCF77, GOES and GPS radio and satellite time services and USNO, ACTS and PTB modem time services. It also supports the IRIG-B and IRIG-E signal format connected via an audio codec. The server methodically avoids the use of Unix-specific library routines where possible by implementing local versions, in order to aid in porting the code to perverse Unix and non-Unix platforms.

While this implementation conforms in most respects to the NTP Version 3 specification RFC-1305, a number of improvements have been made which are described in the conformance statement in the Further Information and Bibliography page. It has been specifically tuned to achieve the highest accuracy possible on whatever hardware and operating-system platform is available. In general, its precision and stability are limited only by the characteristics of the onboard clock source used by the hardware and operating system, usually an uncompensated crystal oscillator. On modern RISC-based processors connected directly to radio clocks via serial-asynchronous interfaces, the accuracy is usually limited by the radio clock and interface to the order of a millisecond or less. The code includes special features to support a pulse-per-second (PPS) signal and/or an IRIG-B signal generated by some radio clocks. When used in conjunction with a suitable hardware level converter, the accuracy can be improved to a few tens of microseconds. Further improvement is possible using an outboard, stabilized frequency source, in which the accuracy and stability are limited only by the characteristics of that source.

The NTP Version 4 distribution includes, in addition to the daemon itself (`ntpd`), several utility programs, including two remote-monitoring programs (`ntpq`, `ntpdc`), a remote clock-setting program similar to the Unix rdate program (`ntpdate`), a traceback utility u seful to discover suitable synchronization sources (`ntptrace`), and various programs used to configure the local platform and calibrate the intrinsic errors. NTP has been ported to a large number of platforms, including most RISC and CISC workstations and mainframes manufactured today. Example configuration files for many models of these machines are included in the distribution. While in most cases the standard version of the implementation runs with no hardware or operating system modifications, not all features of the distribution are available on all platforms. For instance, a special feature allowing Sun workstations to achieve accuracies in the order of 100 microseconds requires some minor changes and additions to the kernel and input/output support.

There are, however, several drawbacks to all of this. `ntpd` is quite fat. This is rotten if your intended platform for the daemon is memory limited. `ntpd` uses `SIGIO` for all input, a facility which appears to not enjoy universal support and whose use seems to exercise the parts of your vendors' kernels which are most likely to have been done poorly. The code is unforgiving in the face of kernel problems which affect performance, and generally requires that you repair the problems in order to achieve acceptable performance. The code has a distinctly experimental flavour and contains features which could charitably be termed failed experiments, but which have not been completely hacked out. Much was learned from the addition of support for a variety of radio clocks, with the result that some radio clock drivers could use some rewriting.

How NTP Works

The approach used by NTP to achieve reliable time synchronization from a set of possibly unreliable remote time servers is somewhat different than other protocols. In particular, NTP does not attempt to synchronize clocks to each other. Rather, each server attempts to synchronize to Universal Coordinated Time (UTC) using the best available source and available transmission paths to that source. This is a fine point which is worth understanding. A group of NTP-synchronized clocks may be close to each other in time, but this is not a consequence of the clocks in the group having synchronized to each other, but rather because each clock has synchronized closely to UTC via the best source it has access to. As such, trying to synchronize a set of clocks to a set of servers whose time is not in mutual agreement may not result in any sort of useful synchronization of the clocks, even if you don't care about

UTC. However, in networks isolated from UTC sources, provisions can made to nominate one of them as a phantom UTC source.

NTP operates on the premise that there is one true standard time, and that if several servers which claim synchronization to standard time disagree about what that time is, then one or more of them must be broken. There is no attempt to resolve differences more gracefully since the premise is that substantial differences cannot exist. In essence, NTP expects that the time being distributed from the root of the synchronization subnet will be derived from some external source of UTC (e.g., a radio clock). This makes it somewhat inconvenient (though by no means impossible) to synchronize hosts together without a reliable source of UTC to synchronize them to. If your network is isolated and you cannot access other people's servers across the Internet, a radio clock may make a good investment.

Time is distributed through a hierarchy of NTP servers, with each server adopting a *stratum* which indicates how far away from an external source of UTC it is operating at. Stratum-1 servers, which are at the top of the pile (or bottom, depending on your point of view), have access to some external time source, usually a radio clock synchronized to time signal broadcasts from radio stations which explicitly provide a standard time service. A stratum-2 server is one which is currently obtaining time from a stratum-1 server, a stratum-3 server gets its time from a stratum-2 server, and so on. To avoid long lived synchronization loops the number of strata is limited to 15.

Each client in the synchronization subnet (which may also be a server for other, higher stratum clients) chooses exactly one of the available servers to synchronize to, usually from among the lowest stratum servers it has access to. This is, however, not always an optimal configuration, for indeed NTP operates under another premise as well, that each server's time should be viewed with a certain amount of distrust. NTP really prefers to have access to several sources of lower stratum time (at least three) since it can then apply an agreement algorithm to detect insanity on the part of any one of these. Normally, when all servers are in agreement, NTP will choose the best of these, where "best" is defined in terms of lowest stratum, closest (in terms of network delay) and claimed precision, along with several other considerations. The implication is that, while one should aim to provide each client with three or more sources of lower stratum time, several of these will only be providing backup service and may be of lesser quality in terms of network delay and stratum (i.e., a same-stratum peer which receives time from lower stratum sources the local server doesn't access directly can also provide good backup service).

Finally, there is the issue of association modes. There are a number of modes in which NTP servers can associate with each other, with the mode of each server in the pair indicating the behaviour the other server can expect from it. In particular, when configuring a server to obtain time from other servers, there is a choice of two modes which may be used. Configuring an association in symmetric-active mode (usually indicated by a `peer` declaration in the configuration file) indicates to the remote server that one wishes to obtain time from the remote server and that one is also willing to supply time to the remote server if need be. This mode is appropriate in configurations involving a number of redundant time servers interconnected via diverse network paths, which is presently the case for most stratum- 1 and stratum-2 servers on the Internet today. Configuring an association in client mode (usually indicated by a `server` declaration in the configuration file) indicates that one wishes to obtain time from the remote server, but that one is not willing to provide time to the remote server. This mode is appropriate for file-server and workstation clients that do not provide synchronization to other local clients.

Client mode is also useful for boot-date-setting programs and the like, which really have no time to provide and which don't retain state about associations over the longer term.

Where the requirements in accuracy and reliability are modest, clients can be configured to use broadcast and/or multicast modes. These modes are not normally utilized by servers with dependent clients. The advantage of these modes is that clients do not need to be configured for a specific server, so that all clients operating can use the same configuration file. Broadcast mode requires a broadcast server on the same subnet, while multicast mode requires support for IP multicast on the client machine, as well as connectivity via the MBONE to a multicast server. Since broadcast messages are not propagated by routers, only those broadcast servers on the same subnet will be used. There is at present no way to select which of possibly many multicast servers will be used, since all operate on the same group address.

Where the maximum accuracy and reliability provided by NTP are needed, clients and servers operate in either client/server or symmetric modes. Symmetric modes are most often used between two or more servers operating as a mutually redundant group. In these modes, the servers in the group members arrange the synchronization paths for maximum performance, depending on network jitter and propagation delay. If one or more of the group members fail, the remaining members automatically reconfigure as required. Dependent clients and servers normally operate in client/server mode, in which a client or dependent server can be synchronized to a group member, but no group member can synchronize to the client or dependent server. This provides protection against malfunctions or protocol attacks.

Servers that provide synchronization to a sizeable population of clients normally operate as a group of three or more mutually redundant servers, each operating with three or more stratum-one or stratum-two servers in client-server modes, as well as all other members of the group in symmetric modes. This provides protection against malfunctions in which one or more servers fail to operate or provide incorrect time. The NTP algorithms have been specifically engineered to resist attacks where some fraction of the configured synchronization sources accidently or purposely provide incorrect time. In these cases a special voting procedure is used to identify spurious sources and discard their data.

Configuring Your Subnet

At startup time the `ntpd` daemon running on a host reads the initial configuration information from a file, usually `/etc/ntp.conf`, unless a different name has been specified at compile time. Putting something in this file which will enable the host to obtain time from somewhere else is usually the first big hurdle after installation of the software itself, which is described in the Building and Installing the Distribution page. At its simplest, what you need to do in the configuration file is declare the servers that the daemon should poll for time synchronization. In principle, no such list is needed if some other time server operating in broadcast/multicast mode is available, which requires the client to operate in a broadcastclient mode.

In the case of a workstation operating in an enterprise network for a public or private organization, there is often an administrative department that coordinates network services, including NTP. Where available, the addresses of appropriate servers can be provided by that department. However, if this infrastructure is not available, it is necessary to explore some portion of the existing NTP subnet now running in the Internet. There are at present many thousands of time servers running NTP in the Internet, a significant number of which are willing to provide a public time- synchronization service. Some of these are listed in the list of public

time servers, which can be accessed via the NTP web page. These data are updated on a regular basis using information provided voluntarily by various site administrators. There are other ways to explore the nearby subnet using the `ntptrace` and `ntpdc` programs.

It is vital to carefully consider the issues of robustness and reliability when selecting the sources of synchronization. Normally, not less than three sources should be available, preferably selected to avoid common points of failure. It is usually better to choose sources which are likely to be "close" to you in terms of network topology, though you shouldn't worry overly about this if you are unable to determine who is close and who isn't. Normally, it is much more serious when a server becomes faulty and delivers incorrect time than when it simply stops operating, since an NTP-synchronized host normally can coast for hours or even days without its clock accumulating serious error approaching a second, for instance. Selecting at least three sources from different operating administrations, where possible, is the minimum recommended, although a lesser number could provide acceptable service with a degraded degree of robustness.

Normally, it is not considered good practice for a single workstation to request synchronization from a primary (stratum-1) time server. At present, these servers provide synchronization for hundreds of clients in many cases and could, along with the network access paths, become seriously overloaded if large numbers of workstation clients requested synchronization directly. Therefore, workstations located in sparsely populated administrative domains with no local synchronization infrastructure should request synchronization from nearby stratum-2 servers instead. In most cases the keepers of those servers in the lists of public servers provide unrestricted access without prior permission; however, in all cases it is considered polite to notify the administrator listed in the file upon commencement of regular service. In all cases the access mode and notification requirements listed in the file must be respected. Under no conditions should servers not in these lists be used without prior permission, as to do so can create severe problems in the local infrastructure, especially in cases of dial-up access to the Internet.

In the case of a gateway or file server providing service to a significant number of workstations or file servers in an enterprise network it is even more important to provide multiple, redundant sources of synchronization and multiple, diversity-routed, network access paths. The preferred configuration is at least three administratively coordinated time servers providing service throughout the administrative domain including campus networks and subnetworks. Each of these should obtain service from at least two different outside sources of synchronization, preferably via different gateways and access paths. These sources should all operate at the same stratum level, which is one less than the stratum level to be used by the local time servers themselves. In addition, each of these time servers should peer with all of the other time servers in the local administrative domain at the stratum level used by the local time servers, as well as at least one (different) outside source at this level. This configuration results in the use of six outside sources at a lower stratum level (toward the primary source of synchronization, usually a radio clock), plus three outside sources at the same stratum level, for a total of nine outside sources of synchronization. While this may seem excessive, the actual load on network resources is minimal, since the interval between polling messages exchanged between peers usually ratchets back to no more than one message every 17 minutes.

The stratum level to be used by the local time servers is an engineering choice. As a matter of policy, and in order to reduce the load on the primary servers, it is desirable to use the highest

stratum consistent with reliable, accurate time synchronization throughout the administrative domain. In the case of enterprise networks serving hundreds or thousands of client file servers and workstations, conventional practice is to obtain service from stratum-1 primary servers listed for public access. When choosing sources away from the primary sources, the particular synchronization path in use at any time can be verified using the `ntptrace` program included in this distribution. It is important to avoid loops and possible common points of failure when selecting these sources. Note that, while NTP detects and rejects loops involving neighboring servers, it does not detect loops involving intervening servers. In the unlikely case that all primary sources of synchronization are lost throughout the subnet, the remaining servers on that subnet can form temporary loops and, if the loss continues for an interval of many hours, the servers will drop off the subnet and free-run with respect to their internal (disciplined) timing sources. After some period with no outside timing source (currently one day), a host will declare itself unsynchronized and provide this information to local application programs.

In many cases the purchase of one or more radio clocks is justified, in which cases good engineering practice is to use the configurations described above anyway and connect the radio clock to one of the local servers. This server is then encouraged to participate in a special primary-server subnetwork in which each radio-equipped server peers with several other similarly equipped servers. In this way the radio-equipped server may provide synchronization, as well as receive synchronization, should the local or remote radio clock(s) fail or become faulty. `ntpd` treats attached radio clock(s) in the same way as other servers and applies the same criteria and algorithms to the time indications, so can detect when the radio fails or becomes faulty and switch to alternate sources of synchronization. It is strongly advised, and in practice for most primary servers today, to employ the authentication or access-control features of the NTP specification in order to protect against hostile intruders and possible destabilization of the time service. Using this or similar strategies, the remaining hosts in the same administrative domain can be synchronized to the three (or more) selected time servers. Assuming these servers are synchronized directly to stratum-1 sources and operate normally as stratum-2, the next level away from the primary source of synchronization, for instance various campus file servers, will operate at stratum 3 and dependent workstations at stratum 4. Engineered correctly, such a subnet will survive all but the most exotic failures or even hostile penetrations of the various, distributed timekeeping resources.

The above arrangement should provide very good, robust time service with a minimum of traffic to distant servers and with manageable loads on the local servers. While it is theoretically possible to extend the synchronization subnet to even higher strata, this is seldom justified and can make the maintenance of configuration files unmanageable. Serving time to a higher stratum peer is very inexpensive in terms of the load on the lower stratum server if the latter is located on the same concatenated LAN. When justified by the accuracy expectations, NTP can be operated in broadcast and multicast modes, so that clients need only listen for periodic broadcasts and do not need to send anything.

When planning your network you might, beyond this, keep in mind a few generic don'ts, in particular:

- Don't synchronize a local time server to another peer at the same stratum, unless the latter is receiving time from lower stratum sources the former doesn't talk to directly. This minimizes the occurrence of common points of failure, but does not eliminate them in cases where the usual chain of associations to the primary sources of synchronization are disrupted due to failures.

- Don't configure peer associations with higher stratum servers. Let the higher strata configure lower stratum servers, but not the reverse. This greatly simplifies configuration file maintenance, since there is usually much greater configuration churn in the high stratum clients such as personal workstations.

- Don't synchronize more than one time server in a particular administrative domain to the same time server outside that domain. Such a practice invites common points of failure, as well as raises the possibility of massive abuse, should the configuration file be automatically distributed do a large number of clients.

There are many useful exceptions to these rules. When in doubt, however, follow them.

Configuring Your Server or Client

As mentioned previously, the configuration file is usually called /etc/ntp.conf. This is an ASCII file conforming to the usual comment and whitespace conventions. A working configuration file might look like (in this and other examples, do not copy this directly):

```
# peer configuration for host whimsy
# (expected to operate at stratum 2)

server rackety.udel.edu
server umd1.umd.edu
server lilben.tn.cornell.edu

driftfile /etc/ntp.drift
```

(Note the use of host names, although host addresses in dotted-quad notation can also be used. It is always preferable to use names rather than addresses, since over time the addresses can change, while the names seldom change.)

This particular host is expected to operate as a client at stratum 2 by virtue of the `server` keyword and the fact that two of the three servers declared (the first two) have radio clocks and usually run at stratum 1. The third server in the list has no radio clock, but is known to maintain associations with a number of stratum 1 peers and usually operates at stratum 2. Of particular importance with the last host is that it maintains associations with peers besides the two stratum 1 peers mentioned. This can be verified using the `ntpq` program mentioned above. When configured using the `server` keyword, this host can receive synchronization from any of the listed servers, but can never provide synchronization to them.

Unless restricted using facilities described later, this host can provide synchronization to dependent clients, which do not have to be listed in the configuration file. Associations maintained for these clients are transitory and result in no persistent state in the host. These clients are normally not visible using the `ntpq` program included in the distribution; however, `ntpd` includes a monitoring feature (described later) which caches a minimal amount of client information useful for debugging administrative purposes.

A time server expected to both receive synchronization from another server, as well as to provide synchronization to it, is declared using the `peer` keyword instead of the `server` keyword. In all other aspects the server operates the same in either mode and can provide

synchronization to dependent clients or other peers. If a local source of UTC time is available, it is considered good engineering practice to declare time servers outside the administrative domain as `peer` and those inside as `server` in order to provide redundancy in the global Internet, while minimizing the possibility of instability within the domain itself. A time server in one domain can in principle heal another domain temporarily isolated from all other sources of synchronization. However, it is probably unwise for a casual workstation to bridge fragments of the local domain which have become temporarily isolated.

Note the inclusion of a `driftfile` declaration. One of the things the NTP daemon does when it is first started is to compute the error in the intrinsic frequency of the clock on the computer it is running on. It usually takes about a day or so after the daemon is started to compute a good estimate of this (and it needs a good estimate to synchronize closely to its server). Once the initial value is computed, it will change only by relatively small amounts during the course of continued operation. The `driftfile` declaration indicates to the daemon the name of a file where it may store the current value of the frequency error so that, if the daemon is stopped and restarted, it can reinitialize itself to the previous estimate and avoid the day's worth of time it will take to recompute the frequency estimate. Since this is a desirable feature, a `driftfile` declaration should always be included in the configuration file.

An implication in the above is that, should `ntpd` be stopped for some reason, the local platform time will diverge from UTC by an amount that depends on the intrinsic error of the clock oscillator and the time since last synchronized. In view of the length of time necessary to refine the frequency estimate, every effort should be made to operate the daemon on a continuous basis and minimize the intervals when for some reason it is not running.

Ntp4 Versus Previous Versions

There are several items of note when dealing with a mixture of `ntp4` and previous distributions of NTP Version 2 (`ntpd`) and NTP Version 1 (`ntp3.4`). The `ntp4` implementation conforms to the NTP Version 3 specification RFC-1305 and, in addition, contains additional feaures documented in the Release Notes page. As such, by default when no additional information is available concerning the preferences of the peer, `ntpd` claims to be version 4 in the packets that it sends from configured associations. The `version` subcommand of the `server`, `peer`, `broadcast` and `manycastclient` command can be used to change the default. In unconfigured (ephemeral) associaitons, the daemon always replies in the same version as the request.

An NTP implementation conforming to a previous version specification ordinarily discards packets from a later version. However, in most respects documented in RFC-1305, The version 2 implementation is compatible with the version 3 algorithms and protocol. The version 1 implementation contains most of the version 2 algorithms, but without important features for clock selection and robustness. Nevertheless, in most respects the NTP versions are backwards compatible. The sticky part here is that, when a previous version implementation receives a packet claiming to be from a version 4 server, it discards it without further processing. Hence there is a danger that in some situations synchronization with previous versions will fail.

The trouble occurs when an previous version is to be included in an `ntpd` configuration file. With no further indication, `ntpd` will send packets claiming to be version 4 when it polls. To get around this, `ntpd` allows a qualifier to be added to configuration entries to indicate which version to use when polling. Hence the entries

```
     # specify NTP version 1

     server mimsy.mil version
1     # server running ntpd version 1
     server apple.com version
2     # server running ntpd version 2
```

will cause version 1 packets to be sent to the host mimsy.mil and version 2 packets to be sent to apple.com. If you are testing `ntpd` against previous version servers you will need to be careful about this. Note that, as indicated in the RFC-1305 specification, there is no longer support for the original NTP specification, once called NTP Version 0.

Traffic Monitoring

`ntpd` handles peers whose stratum is higher than the stratum of the local server and pollers using client mode by a fast path which minimizes the work done in responding to their polls, and normally retains no memory of these pollers. Sometimes, however, it is interesting to be able to determine who is polling the server, and how often, as well as who has been sending other types of queries to the server.

To allow this, `ntpd` implements a traffic monitoring facility which records the source address and a minimal amount of other information from each packet which is received by the server. This feature is normally enabled, but can be disabled if desired using the configuration file entry:

```
     # disable monitoring feature
     disable monitor
```

The recorded information can be displayed using the `ntpdc` query program, described briefly below.

Address-and-Mask Restrictions

The address-and-mask configuration facility supported by `ntpd` is quite flexible and general, but is not an integral part of the NTP Version 3 specification. The major drawback is that, while the internal implementation is very nice, the user interface is not. For this reason it is probably worth doing an example here. Briefly, the facility works as follows. There is an internal list, each entry of which holds an address, a mask and a set of flags. On receipt of a packet, the source address of the packet is compared to each entry in the list, with a match being posted when the following is true:

```
     (source_addr & mask) == (address &
mask)
```

A particular source address may match several list entries. In this case the entry with the most one bits in the mask is chosen. The flags associated with this entry are used to control the access.

In the current implementation the flags always add restrictions. In effect, an entry with no flags set leaves matching hosts unrestricted. An entry can be added to the internal list using a `restrict` declaration. The flags associated with the entry are specified textually. For example,

the `notrust` flag indicates that hosts matching this entry, while treated normally in other respects, shouldn't be trusted to provide synchronization even if otherwise so enabled. The `nomodify` flag indicates that hosts matching this entry should not be allowed to do run-time configuration. There are many more flags, see the `ntpd` page.

Now the example. Suppose you are running the server on a host whose address is 128.100.100.7. You would like to ensure that run time reconfiguration requests can only be made from the local host and that the server only ever synchronizes to one of a pair of off-campus servers or, failing that, a time source on net 128.100. The following entries in the configuration file would implement this policy:

```
     # by default, don't trust and don't allow
modifications

     restrict default notrust nomodify

     # these guys are trusted for time, but no
modifications allowed

     restrict 128.100.0.0 mask 255.255.0.0 nomodify
     restrict 128.8.10.1 nomodify
     restrict 192.35.82.50 nomodify

     # the local addresses are unrestricted

     restrict 128.100.100.7
     restrict 127.0.0.1
```

The first entry is the default entry, which all hosts match and hence which provides the default set of flags. The next three entries indicate that matching hosts will only have the `nomodify` flag set and hence will be trusted for time. If the mask isn't specified in the `restrict` keyword, it defaults to 255.255.255.255. Note that the address 128.100.100.7 matches three entries in the table, the default entry (mask 0.0.0.0), the entry for net 128.100 (mask 255.255.0.0) and the entry for the host itself (mask 255.255.255.255). As expected, the flags for the host are derived from the last entry since the mask has the most bits set.

The only other thing worth mentioning is that the `restrict` declarations apply to packets from all hosts, including those that are configured elsewhere in the configuration file and even including your clock pseudopeer(s), if any. Hence, if you specify a default set of restrictions which you don't wish to be applied to your configured peers, you must remove those restrictions for the configured peers with additional `restrict` declarations mentioning each peer separately.

Authentication

`ntpd` supports the optional authentication procedure specified in the NTP Version 2 and 3 specifications. Briefly, when an association runs in authenticated mode, each packet transmitted has appended to it a 32-bit key ID and a 64/128-bit cryptographic checksum of the packet contents computed using either the Data Encryption Standard (DES) or Message Digest (MD5) algorithms. Note that, while either of these algorithms provide sufficient protection from message- modification attacks, distribution of the former algorithm implementation is restricted to the U.S. and Canada, while the latter presently is free from such

restrictions. For this reason, the DES algorithm is not included in the current distribution. Directions for obtaining it in other countries is in the Building and Installing the Distribution page. With either algorithm the receiving peer recomputes the checksum and compares it with the one included in the packet. For this to work, the peers must share at least one encryption key and, furthermore, must associate the shared key with the same key ID.

This facility requires some minor modifications to the basic packet processing procedures, as required by the specification. These modifications are enabled by the `enable auth` configuration declaration, which is currently the default. In authenticated mode, peers which send unauthenticated packets, peers which send authenticated packets which the local server is unable to decrypt and peers which send authenticated packets encrypted using a key we don't trust are all marked untrustworthy and unsuitable for synchronization. Note that, while the server may know many keys (identified by many key IDs), it is possible to declare only a subset of these as trusted. This allows the server to share keys with a client which requires authenticated time and which trusts the server, but which is not trusted by the server. Also, some additional configuration language is required to specify the key ID to be used to authenticate each configured peer association. Hence, for a server running in authenticated mode, the configuration file might look similar to the following:

```
     # peer configuration for 128.100.100.7
     # (expected to operate at stratum 2)
     # fully authenticated this time

     peer 128.100.49.105 key 22 #
suzuki.ccie.utoronto.ca
     peer 128.8.10.1 key 4    #
umd1.umd.edu
     peer 192.35.82.50 key 6  #
lilben.tn.cornell.edu

     keys /usr/local/etc/ntp.keys  # path for
key file
     trustedkey 1 2 14 15     #
define trusted keys
     requestkey
15           #
key (7) for accessing server variables
     controlkey
15           #
key (6) for accessing server variables

     authdelay
0.000094       # authentication delay
(Sun4c/50 IPX)
```

There are a couple of previously unmentioned things in here. The `keys` line specifies the path to the keys file (see below and the `ntpd` document page for details of the file format). The `trustedkey` declaration identifies those keys that are known to be uncompromised; the remainder presumably represent the expired or possibly compromised keys. Both sets of keys must be declared by key identifier in the `ntp.keys` file described below. This provides a way to retire old keys while minimizing the frequency of delicate key-distribution procedures. The `requestkey` line establishes the key to be used for mode-6 control messages as specified in RFC-1305 and used by the `ntpq` utility program, while the `controlkey` line establishes the key to be used for mode-7 private control messages used by the `ntpdc` utility program. These keys

are used to prevent unauthorized modification of daemon variables.

Ordinarily, the authentication delay; that is, the processing time taken between the freezing of a transmit timestamp and the actual transmission of the packet when authentication is enabled (i.e. more or less the time it takes for the DES or MD5 routine to encrypt a single block) is computed automatically by the daemon. If necessary, the delay can be overriden by the `authdelay` line, which is used as a correction for the transmit timestamp. This can be computed for your CPU by the `authspeed` program included in the distribution. The usage is illustrated by the following:

```
     # for DES keys

     authspeed -n 30000 auth.samplekeys
     # for MD5 keys

     authspeed -mn 30000 auth.samplekeys
```

Additional utility programs included in the `./authstuff` directory can be used to generate random keys, certify implementation correctness and display sample keys. As a general rule, keys should be chosen randomly, except possibly the request and control keys, which must be entered by the user as a password.

The `ntp.keys` file contains the list of keys and associated key IDs the server knows about (for obvious reasons this file is better left unreadable by anyone except root). The contents of this file might look like:

```
     # ntp keys file (ntp.keys)
     1    N
29233E0461ECD6AE    # des key in NTP format
     2    M
RIrop8KPPvQvYotM    # md5 key as an ASCII random string
     14   M
sundial
;  # md5 key as an ASCII string
     15   A
sundial
;  # des key as an ASCII string

     # the following 3 keys are identical

     10   A    SeCReT
     10   N
d3e54352e5548080
     10   S
a7cb86a4cba80101
```

In the keys file the first token on each line indicates the key ID, the second token the format of the key and the third the key itself. There are four key formats. An `A` indicates a DES key written as a 1- to-8 character string in 7-bit ASCII representation, with each character standing for a key octet (like a Unix password). An `S` indicates a DES key written as a hex number in the DES standard format, with the low order bit (LSB) of each octet being the (odd) parity bit. An `N` indicates a DES key again written as a hex number, but in NTP standard format with the high order bit of each octet being the (odd) parity bit (confusing enough?). An `M` indicates an MD5 key written as a 1-to-31 character ASCII string in the `A` format. Note that, because of the simple

tokenizing routine, the characters `' '`, `'#'`, `'\t'`, `'\n'` and `'\0'` can't be used in either a DES or MD5 ASCII key. Everything else is fair game, though. Key 0 (zero) is used for special purposes and should not appear in this file.

The big trouble with the authentication facility is the keys file. It is a maintenance headache and a security problem. This should be fixed some day. Presumably, this whole bag of worms goes away if/when a generic security regime for the Internet is established. An alternative with NTP Version 4 is the autokey feature, which uses random session keys and public-key cruptography and avoids the key file entirely. While this feature is not completely finished yet, details can be found in the Release Notes page.

Query Programs

Three utility query programs are included with the distribution, `ntpq`, `ntptrace` and `ntpdc`. `ntpq` is a handy program which sends queries and receives responses using NTP standard mode-6 control messages. Since it uses the standard control protocol specified in RFC- 1305, it may be used with NTP Version 2 and Version 3 implementations for both Unix and Fuzzball, but not Version 1 implementations. It is most useful to query remote NTP implementations to assess timekeeping accuracy and expose bugs in configuration or operation.

`ntptrace` can be used to display the current synchronization path from a selected host through possibly intervening servers to the primary source of synchronization, usually a radio clock. It works with both version 2 and version 3 servers, but not version 1.

`ntpdc` is a horrid program which uses NTP private mode-7 control messages to query local or remote servers. The format and contents of these messages are specific to this version of `ntpd` and some older versions. The program does allow inspection of a wide variety of internal counters and other state data, and hence does make a pretty good debugging tool, even if it is frustrating to use. The other thing of note about `ntpdc` is that it provides a user interface to the run time reconfiguration facility. See the respective document pages for details on the use of these programs.

Run-Time Reconfiguration

`ntpd` was written specifically to allow its configuration to be fully modifiable at run time. Indeed, the only way to configure the server is at run time. The configuration file is read only after the rest of the server has been initialized into a running default-configured state. This facility was included not so much for the benefit of Unix, where it is handy but not strictly essential, but rather for dedicated platforms where the feature is more important for maintenance. Nevertheless, run time configuration works very nicely for Unix servers as well.

Nearly all of the things it is possible to configure in the configuration file may be altered via NTP mode-7 messages using the `ntpdc` program. Mode-6 messages may also provide some limited configuration functionality (though the only thing you can currently do with mode-6 messages is set the leap-second warning bits) and the `ntpq` program provides generic support for the latter. The leap bits that can be set in the `leap_warning` variable (up to one month ahead) and in the `leap_indication` variable have a slightly different encoding than the usual interpretation:

```
Value           Action


00           &nbs
p; The daemon passes the leap bits of its


synchronisation source (usual mode of operation)

        01/10   A leap
second is added/deleted


11           &nbs
p; Leap information from the synchronization source

            is
ignored (thus LEAP_NOWARNING is passed on)
```

Mode-6 and mode-7 messages which would modify the configuration of the server are required to be authenticated using standard NTP authentication. To enable the facilities one must, in addition to specifying the location of a keys file, indicate in the configuration file the key IDs to be used for authenticating reconfiguration commands. Hence the following fragment might be added to a configuration file to enable the mode-6 (ntpq) and mode-7 (ntpdc) facilities in the daemon:

```
     # specify mode-6 and mode-7 trusted keys

     requestkey 65535    # for mode-7
requests
     controlkey 65534    # for mode-6
requests
```

If the `requestkey` and/or the `controlkey` configuration declarations are omitted from the configuration file, the corresponding run-time reconfiguration facility is disabled.

The query programs require the user to specify a key ID and a key to use for authenticating requests to be sent. The key ID provided should be the same as the one mentioned in the configuration file, while the key should match that corresponding to the key ID in the keys file. As the query programs prompt for the key as a password, it is useful to make the request and control authentication keys typeable (in ASCII format) from the keyboard.

Name Resolution

`ntpd` includes the capability to specify host names requiring resolution in `peer` and `server` declarations in the configuration file. However, in some outposts of the Internet, name resolution is unreliable and the interface to the Unix resolver routines is synchronous. The hangups and delays resulting from name-resolver clanking can be unacceptable once the NTP server is running (and remember it is up and running before the configuration file is read). However, it is advantageous to resolve time server names, since their addresses are occasionally changed.

In order to prevent configuration delays due to the name resolver, the daemon runs the name resolution process in parallel with the main daemon code. When the daemon comes across a

`peer` or `server` entry with a non-numeric host address, it records the relevant information in a temporary file and continues on. When the end of the configuration file has been reached and one or more entries requiring name resolution have been found, the server runs the name resolver from the temporary file. The server then continues on normally but with the offending peers/servers omitted from its configuration.

As each name is resolved, it configures the associated entry into the server using the same mode-7 run time reconfiguration facility that `ntpdc` uses. If temporary resolver failures occur, the resolver will periodically retry the requests until a definite response is received. The program will continue to run until all entries have been resolved.

Dealing with Frequency Tolerance Violations (`tickadj` and Friends)

The NTP Version 3 specification RFC-1305 calls for a maximum oscillator frequency tolerance of +-100 parts-per-million (PPM), which is representative of those components suitable for use in relatively inexpensive workstation platforms. For those platforms meeting this tolerance, NTP will automatically compensate for the frequency errors of the individual oscillator and no further adjustments are required, either to the configuration file or to various kernel variables. For the NTP Version 4 release, this tolerance has been increased to +-500 PPM.

However, in the case of certain notorious platforms, in particular Sun 4.1.1 systems, the performance can be improved by adjusting the values of certain kernel variables; in particular, `tick` and `tickadj`. The variable `tick` is the increment in microseconds added to the system time on each interval- timer interrupt, while the variable `tickadj` is used by the time adjustment code as a slew rate, in microseconds per tick. When the time is being adjusted via a call to the system routine `adjtime()`, the kernel increases or reduces tick by `tickadj` microseconds per tick until the specified adjustment has been completed. Unfortunately, in most Unix implementations the tick increment must be either zero or plus/minus exactly `tickadj` microseconds, meaning that adjustments are truncated to be an integral multiple of `tickadj` (this latter behaviour is a misfeature, and is the only reason the `tickadj` code needs to concern itself with the internal implementation of `tickadj` at all). In addition, the stock Unix implementation considers it an error to request another adjustment before a prior one has completed.

Thus, to make very sure it avoids problems related to the roundoff, the `tickadj` program can be used to adjust the values of `tick` and `tickadj`. This ensures that all adjustments given to `adjtime()` are an even multiple of `tickadj` microseconds and computes the largest adjustment that can be completed in the adjustment interval (using both the value of `tick` and the value of `tickadj`) so it can avoid exceeding this limit. It is important to note that not all systems will allow inspection or modification of kernel variables other than at system build time. It is also important to know that, with the current NTP tolerances, it is rarely necessary to make these changes, but in many cases they will substantially improve the general accurace of the time service.

Unfortunately, the value of `tickadj` set by default is almost always too large for `ntpd`. NTP operates by continuously making small adjustments to the clock, usually at one-second intervals. If `tickadj` is set too large, the adjustments will disappear in the roundoff; while, if `tickadj` is too small, NTP will have difficulty if it needs to make an occasional large adjustment. While the daemon itself will read the kernel's values of these variables, it will not change the values, even if they are unsuitable. You must do this yourself before the daemon is

started using the `tickadj` program included in the `./util` directory of the distribution. Note that the latter program will also compute an optimal value of `tickadj` for NTP use based on the kernel's value of `tick`.

The `tickadj` program can reset several other kernel variables if asked. It can change the value of `tick` if asked. This is handy to compensate for kernel bugs which cause the clock to run with a very large frequency error, as with SunOS 4.1.1 systems. It can also be used to set the value of the kernel `dosynctodr` variable to zero. This variable controls whether to synchronize the system clock to the time- of-day clock, something you really don't want to be happen when `ntpd` is trying to keep it under control. In some systems, such as recent Sun Solaris kernels, the `dosynctodr` variable is the only one that can be changed by the `tickadj` program. In this and other modern kernels, it is not necessary to change the other variables in any case.

In order to maintain reasonable correctness bounds, as well as reasonably good accuracy with acceptable polling intervals, `ntpd` will complain if the frequency error is greater than 500 PPM. For machines with a value of `tick` in the 10-ms range, a change of one in the value of `tick` will change the frequency by about 100 PPM. In order to determine the value of `tick` for a particular CPU, disconnect the machine from all sources of time (`dosynctodr` = 0) and record its actual time compared to an outside source (eyeball-and-wristwatch will do) over a day or more. Multiply the time change over the day by 0.116 and add or subtract the result to tick, depending on whether the CPU is fast or slow. An example call to `tickadj` useful on SunOS 4.1.1 is:

```
tickadj -t 9999 -a 5 -s
```

which sets tick 100 PPM fast, `tickadj` to 5 microseconds and turns off the clock/calendar chip fiddle. This line can be added to the `rc.local` configuration file to automatically set the kernel variables at boot time.

All this stuff about diddling kernel variables so the NTP daemon will work is really silly. If vendors would ship machines with clocks that kept reasonable time and would make their `adjtime()` system call apply the slew it is given exactly, independent of the value of `tickadj`, all this could go away. This is in fact the case on many current Unix systems.

Tuning Your Subnet

There are several parameters available for tuning the NTP subnet for maximum accuracy and minimum jitter. One of these is the `prefer` configuration declaration described in Mitigation Rules and the `prefer` Keyword documentation page. When more than one eligible server exists, the NTP clock-selection and combining algorithms act to winnow out all except the "best" set of servers using several criteria based on differences between the readings of different servers and between successive readings of the same server. The result is usually a set of surviving servers that are apparently statistically equivalent in accuracy, jitter and stability. The population of survivors remaining in this set depends on the individual server characteristics measured during the selection process and may vary from time to time as the result of normal statistical variations. In LANs with high speed RISC-based time servers, the population can become somewhat unstable, with individual servers popping in and out of the surviving population, generally resulting in a regime called *clockhopping*.

When only the smallest residual jitter can be tolerated, it may be convenient to elect one of the

servers at each stratum level as the preferred one using the keyword `prefer` on the configuration declaration for the selected server:

```
     # preferred server declaration

     peer rackety.udel.edu prefer
# preferred server
```

The preferred server will always be included in the surviving population, regardless of its characteristics and as long as it survives preliminary sanity checks and validation procedures.

The most useful application of the `prefer` keyword is in high speed LANs equipped with precision radio clocks, such as a GPS receiver. In order to insure robustness, the hosts need to include outside peers as well as the GPS-equipped server; however, as long as that server is running, the synchronization preference should be that server. The keyword should normally be used in all cases in order to prefer an attached radio clock. It is probably inadvisable to use this keyword for peers outside the LAN, since it interferes with the carefully crafted judgement of the selection and combining algorithms.

Provisions for Leap Seconds and Accuracy Metrics

`ntpd` understands leap seconds and will attempt to take appropriate action when one occurs. In principle, every host running ntpd will insert a leap second in the local timescale in precise synchronization with UTC. This requires that the leap-warning bits be activated some time prior to the occurrence of a leap second at the primary (stratum 1) servers. Subsequently, these bits are propagated throughout the subnet depending on these servers by the NTP protocol itself and automatically implemented by `ntpd` and the time- conversion routines of each host. The implementation is independent of the idiosyncrasies of the particular radio clock, which vary widely among the various devices, as long as the idiosyncratic behavior does not last for more than about 20 minutes following the leap. Provisions are included to modify the behavior in cases where this cannot be guaranteed. While provisions for leap seconds have been carefully crafted so that correct timekeeping immediately before, during and after the occurrence of a leap second is scrupulously correct, stock Unix systems are mostly inept in responding to the available information. This caveat goes also for the maximum-error and statistical-error bounds carefully calculated for all clients and servers, which could be very useful for application programs needing to calibrate the delays and offsets to achieve a near- simultaneous commit procedure, for example. While this information is maintained in the `ntpd` data structures, there is at present no way for application programs to access it. This may be a topic for further development.

Clock Support Overview

`ntpd` was designed to support radio (and other external) clocks and does some parts of this function with utmost care. Clocks are treated by the protocol as ordinary NTP peers, even to the point of referring to them with an (invalid) IP host address. Clock addresses are of the form 127.127.*t.u*, where *t* specifies the particular type of clock (i.e., refers to a particular clock driver) and *u* is a unit number whose interpretation is clock-driver dependent. This is analogous to the use of major and minor device numbers by Unix and permits multiple instantiations of clocks of the same type on the same server, should such magnificent redundancy be required.

Because clocks look much like peers, both configuration file syntax and run time reconfiguration commands can be used to control clocks in the same way as ordinary peers. Clocks are configured via `server` declarations in the configuration file, can be started and stopped using ntpdc and are subject to address-and-mask restrictions much like a normal peer, should this stretch of imagination ever be useful. As a concession to the need to sometimes transmit additional information to clock drivers, an additional configuration file is available: the `fudge` statement. This enables one to specify the values of two time quantities, two integral values and two flags, the use of which is dependent on the particular clock driver. For example, to configure a PST radio clock which can be accessed through the serial device `/dev/pst1`, with propagation delays to WWV and WWVH of 7.5 and 26.5 milliseconds, respectively, on a machine with an imprecise system clock and with the driver set to disbelieve the radio clock once it has gone 30 minutes without an update, one might use the following configuration file entries:

```
# radio clock fudge fiddles
server 127.127.3.1
fudge 127.127.3.1 time1 0.0075 time2 0.0265
fudge 127.127.3.1 value2 30 flag1 1
```

Additional information on the interpretation of these data with respect to various radio clock drivers is given in the Reference Clock Drivers document page and in the individual driver documents accessible via that page.

Towards the Ultimate Tick

This section considers issues in providing precision time synchronization in NTP subnets which need the highest quality time available in the present technology. These issues are important in subnets supporting real-time services such as distributed multimedia conferencing and wide-area experiment control and monitoring.

In the Internet of today synchronization paths often span continents and oceans with moderate to high variations in delay due to traffic spasms. NTP is specifically designed to minimize timekeeping jitter due to delay variations using intricately crafted filtering and selection algorithms; however, in cases where these variations are as much as a second or more, the residual jitter following these algorithms may still be excessive. Sometimes, as in the case of some isolated NTP subnets where a local source of precision time is available, such as a PPS signal produced by a calibrated cesium clock, it is possible to remove the jitter and retime the local clock oscillator of the NTP server. This has turned out to be a useful feature to improve the synchronization quality of time distributed in remote places where radio clocks are not available. In these cases special features of the distribution are used together with the PPS signal to provide a jitter-free timing signal, while NTP itself is used to provide the coarse timing and resolve the seconds numbering.

Most available radio clocks can provide time to an accuracy in the order of milliseconds, depending on propagation conditions, local noise levels and so forth. However, as a practical matter, all clocks can occasionally display errors significantly exceeding nominal specifications. Usually, the algorithms used by NTP for ordinary network peers, as well as radio clock peers will detect and discard these errors as discrepancies between the disciplined local clock oscillator and the decoded time message produced by the radio clock. Some radio clocks can produce a special PPS signal which can be interfaced to the server platform in a number of ways and used to substantially improve the (disciplined) clock oscillator jitter and wander

characteristics by at least an order of magnitude. Using these features it is possible to achieve accuracies in the order of a few tens of microseconds with a fast RISC- based platform.

There are three ways to implement PPS support, depending on the radio clock model, platform model and serial line interface. These are described in detail in the application notes mentioned in the The Network Time Protocol (NTP) Distribution document page. Each of these requires circuitry to convert the TTL signal produced by most clocks to the EIA levels used by most serial interfaces. The Gadget Box PPS Level Converter and CHU Modem document page describes a device designed to do this. Besides being useful for this purpose, this device includes an inexpensive modem designed for use with the Canadian CHU time/frequency radio station.

In order to select the appropriate implementation, it is important to understand the underlying PPS mechanism used by ntpd. The PPS support depends on a continuous source of PPS pulses used to calculate an offset within +-500 milliseconds relative to the local clock. The serial timecode produced by the radio or the time determined by NTP in absence of the radio is used to adjust the local clock within +-128 milliseconds of the actual time. As long as the local clock is within this interval the PPS support is used to discipline the local clock and the timecode used only to verify that the local clock is in fact within the interval. Outside this interval the PPS support is disabled and the timecode used directly to control the local clock.

Parting Shots

There are several undocumented programs which can be useful in unusual cases. They can be found in the `./clockstuff` and `./authstuff` directories of the distribution. One of these is the `propdelay` program, which can compute high frequency radio propagation delays between any two points whose latitude and longitude are known. The program understands something about the phenomena which allow high frequency radio propagation to occur, and will generally provide a better estimate than a calculation based on the great circle distance. Other programs of interest include `clktest`, which allows one to exercise the generic clock line discipline, and `chutest`, which runs the basic reduction algorithms used by the daemon on data received from a serial port.

David L. Mills <mills@udel.edu>