Network Time Protocol User's Guide

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No: 805-0078-10
Revision A, March 1997

THE NETWORK IS THE COMPUTER™

Table of Contents

Preface

This document describes the XNTP implementation of Network Time Protocol (NTP), which is supported on the Ultra Enterprise 10000 server and its SSP (System Service Processor) to maintain consistent time on both machines.

Intended Audience

This manual is intended for Ultra Enterprise 10000 users and system administrators. It is written for users who have a working knowledge of the Solaris operating system. If you do not have such knowledge, review the other Sun AnswerBooks provided with this system.

Related Documentation

NTP Reference, Part Number 805-0079 — See these man pages for more information about individual NTP commands.

Typographic Conventions

The following table describes the typefaces and symbols used in this manual.

Typeface or Symbol	**Meaning**	**Example**
`Courier` `(Constant` `-width)`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`

`Bold`	What you type. Often shown with on-screen computer output	`system%` **`su`**
		`Password:`

Italics	Command-line placeholder: replace with a real name or value	To delete a file, type **`rm`** *filename*.
	Document titles, new words or terms, or words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be root to do this.

Introduction 1

This document is a slightly edited version of the paper *Notes on Xntpd Configuration*, dated 14 January 1993, by David L. Mills at the University of Delaware. That document was a rewrite and update of one dated 5 November 1989 by Dennis Ferguson of the University of Toronto.

The *Network Time Protocol User's Guide* explains the use of `xntpd(1M)` and related programs, and of Network Time Protocol (NTP). It describes the NTP Version 3 specification, as defined in RFC 1305, and 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. To support the ultimate precision of about 232 picoseconds in the NTP specification, `xntpd(1M)` does no floating-point arithmetic, but instead, manipulates the 64-bit NTP timestamps as unsigned 64-bit integers. `xntpd(1M)` fully implements NTP versions 2 and 3 authentication and a mode-6 control-message facility. As extensions to the specification, a flexible address-and-mask restriction facility has been included, along with a private mode-7 control-message facility used to remotely reconfigure the system and monitor a considerable amount of internal detail.

NTP is biased toward the needs of a busy time server with numerous, possibly 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 features have been included to permit efficient management and monitoring of a busy primary server, features that are not useful to a server on a high-stratum client. NTP makes good use of high-performance, special-purpose hardware, such as precision oscillators and radio clocks. It supports a number of radio clocks, including those for the WWV, CHU, WWVB[1], DCF77, GOES and GPS radio and satellite services. The server avoids the use of UNIX-specific library routines where possible by implementing local versions, to aid in porting NTP to other UNIX and non-UNIX platforms.

While this implementation follows NTP specification RFC 1305, it has been specifically tuned to achieve the highest accuracy possible on any available hardware and operating system platform. In general, its precision is limited only by that of the on-board time-of-day clock maintained by the hardware and operating system, while its stability is limited only by that of the on-board frequency source, 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 that of the radio clock and interface to the order of a few milliseconds. The code includes special features to support a one-pulse-per-second (1pps) signal generated by some radio clocks. When used in conjunction with a suitable hardware level converter, the accuracy can be improved to the order of 100 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 `xntp3` distribution includes, in addition to the daemon itself (`xntpd(1M)`), several utility programs, including two remote-monitoring programs (`ntpq(1M)`, `xntpdc(1M)`), a remote clock-setting program (`ntpdate(1M)`) similar to the UNIX `rdate(1M)` program, a traceback utility (`ntptrace(1M)`) useful to discover suitable synchronization sources, and various programs for configuring the local platform and calibrating the intrinsic errors.

NTP has been ported to a large number of platforms, including most RISC and CISC workstations and mainframes manufactured today. 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 that enables Sun 4 systems to achieve accuracies in the order of 100 microseconds requires some minor changes and additions tot he kernel and input/output support.

1. The Sun implementation of this code supports WWVB and PST radio clocks.

Network Time Protocol 2

The approach used by NTP to achieve reliable time synchronization from a set of remote time servers is somewhat different than other such protocols. In particular, NTP does not attempt to synchronize clocks to each other. Rather, each server attempts to synchronize to UTC (Universal Coordinated Time) using the best available source and transmission paths to that source. For example, 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 to which it has access. 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 UTC is not an issue. NTP assumes there is only one true standard time, and that if several servers that claim synchronization to standard time disagree about what that time is, one or more of them must be broken. It does not attempt to resolve differences more gracefully, since its premise is that substantial differences cannot exist. In essence, NTP expects that the time being distributed from the root of the synchronization subnet is derived from some external source of UTC (e.g., a radio clock). This makes it somewhat inconvenient (though not impossible) to synchronize hosts without a reliable source of UTC. If your network is isolated and you cannot access other servers across the Internet, a radio clock may be a better solution.

Time is distributed through a hierarchy of NTP servers, with each server adopting a “stratum” that indicates how far away from an external source of UTC it is operating at. Stratum-1 servers have access to an external time source, usually a radio clock synchronized to time signal broadcasts from radio stations that explicitly provide a standard time service. A stratum-2 server is one that 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 to which it has access. It is thus possible to construct a synchronization subnet where each server has exactly one source of lower stratum time to which it can synchronize. This is, however, not an optimal configuration, for NTP operates under another premise as well, that each server's time should be viewed with a certain amount of distrust. NTP prefers to have access to several sources of lower stratum time (at least three) since it can then apply an agreement algorithm to detect errors by any one of these. Normally, when all servers are in agreement, NTP chooses the best, 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 you 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; that is, a same-stratum peer that receives time from lower-stratum sources not accessed directly by the local server can 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 behavior the other server can expect from it. In particular, when configuring a server to obtain time from other servers, you have two choices:

- Configuring an association in symmetric-active mode (usually indicated by a “peer” declaration in configuration files) indicates that one wants to obtain time from the remote server, and is willing to supply time to it, as well. 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 configuration files) indicates that one wants to obtain time from the remote server, but is not willing to provide time to it. 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 neither provide nor retain state about associations over a long period of time.

Configuring Your Subnet

At start-up time the `xntpd(1M)` daemon running on a host reads the initial configuration information from a file, usually `/etc/opt/SUNWxntp/ntp.conf.` Putting something in this file that 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 other documents included in the xntp3 distribution. 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 explicitly mentions the host and is willing to provide synchronization; however, this is considered dangerous, unless the access control or authentication features (described later) are in use.

Synchronization

When a workstation is used in an enterprise network for a public or private organization, and the addresses of appropriate servers are not available, you can explore some portion of the existing NTP subnet running in the Internet. Many thousands of time servers are doing so, a significant number of which are willing to provide a public time-synchronization service. Some of these are listed in a file maintained on the Internet host `louie.udel.edu` (128.175.1.3) on the path `pub/ntp/doc/clock.txt.` This file is updated on a regular basis using information provided voluntarily by various site administrators. Other ways to explore the nearby subnet include use of the `ntptrace(1M)` and `ntpq(1M)` programs. See the associated man pages for further information on these 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 that 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 over one 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 listed in the `clock.txt` file 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.

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 such as listed in the `clock.txt` file. When choosing sources away from the primary sources, the particular synchronization path in use at any time can be verified using the `ntptrace(1M)` program included in the xntp3 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.

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 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. `xntpd(1M)` 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 xntp3 distribution in order to protect against hostile penetration 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 mode, 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:

1. 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.

2. 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.

3. 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 to a large number of clients.

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

Dennis Ferguson writes: Note that mention was made of machines with “good” clocks versus machines with “bad” ones. There are two things that make a clock good, the precision of the clock (e.g. how many low order bits in a time value are actually significant) and the frequency of occurrence (or lack thereof) of such things as lost clock interrupts. Among the most common computers I have observed there to be a fairly simple algorithm for determining the goodness of its clock. If the machine is a Vax, it probably has a good clock (the low order bit in the time is in the microseconds and most of these seem to manage to get along without losing clock interrupts). If the machine is a Sun 3 it probably doesn't (the low order clock bit is at the 10 or 20 millisecond mark and Sun 3s like to lose clock interrupts, particularly if they have a screen and particularly if they run SunOS 4.0.x). If you have IBM RTts running AOS 4.3, they have fair clocks (low order clock bit at about a millisecond and they don't lose clock interrupts, though they do have trouble with clock rollovers while reading the low order clock bits) but I recommend them as low stratum NTP servers anyway since they aren't much use as anything else. Sun 4s running SunOS 4.1.1 make very good time servers, once some native foolishness mentioned below is surmounted. However, it is very important to avoid using the keyboard firmware, which can cause severe interrupt latencies, in favor of the software drivers ordinarily used in conjunction with a windowing system. If at all possible you should try to use machines with good clocks for the lower strata.

Configuring Your Server or Client

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

```
peer config for 128.100.100.7 (expected to operate at stratum 2)
server 128.4.1.1       # rackety.udel.edu
server 128.8.10.1      # umd1.umd.edu
server 192.35.82.50    # lilben.tn.cornell.edu
driftfile /etc/opt/SUNWxntp/ntp.drift
```

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, actually) 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(1M)` program included in the xntp3 distribution. 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(1M)` program included in the xntp3 distribution; however, `xntpd(1M)` includes a monitoring feature (described later) that 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. 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 that 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 `xntpd(1M)` 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.

Time-Of-Day (TOD)

The Ultra Enterprise 10000 server and the SSP keep time independently, but are kept in sync by the Network Time Protocol dameon, `xntpd(1M)`. During host boot, the host’s kernel asks the SSP for the time, then sets its time to match. If the date on the SSP is changed, `xntpd(1M)` changes the time on the host. The host’s clock device has no battery backup.

If you use the `date(1)` command to change the time on the host, and it differs from that of the SSP, `xntpd(1M)` immediately begins to gradually adjust the host’s time toward that of the SSP. This can prove confusing to users and programs.

The only time you should use the `date(1)` command to set the time on the host is if a problem prevents it from getting its time from the SSP. If this problem occurs and is detected, the following message appears during the boot flow:

```
"WARNING: TOD clock not initialized -- CHECK AND RESET THE
DATE!"
```

If you see this message, execute, as super user, the `date` command from the host. Set the time as close as possible to that shown on the SSP, and the host time should quickly sync up with it.

Xntpd3 Versus Previous Versions

There are several items of note when dealing with a mixture of xntp3 and previous distributions of xntp (NTP Version 2 `xntpd(1M)`) and ntp3.4 (NTP Version 1 ntpd). The xntp3 implementation of `xntpd(1M)` is an NTP Version 3 implementation. As such, by default when no additional information is available concerning the preferences of the peer, `xntpd(1M)` claims to be version 3 in the packets that it sends.

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 previous version is compatible with the version-3 algorithms and protocol. Ntpd, while implementing most of the version-2 algorithms, still believes itself to be a version-1 implementation. The sticky part here is that, when either `xntpd(1M)` version 2 or `ntpd` version 1 receives a packet claiming to be from a version-3 server, it discards it without further processing. Hence there is a danger that in some situations synchronization with previous versions will fail.

`xntpd(1M)` is aware of this problem. In particular, when `xntpd(1M)` is polled first by a host claiming to be a previous version 1 or version 2 implementation, `xntpd(1M)` claims to be a version 1 or 2 implementation, respectively, in packets returned to the poller. This allows `xntpd(1M)` to serve previous version clients transparently. The trouble occurs when an previous version is to be included in an `xntpd(1M)` configuration file. With no further indication, `xntpd(1M)` will send packets claiming to be version 3 when it polls. To get around this, xntpd(1M) allows a qualifier to be added to configuration entries to indicate which version to use when polling. Hence, the entry will cause version 1 packets to be sent to the host address 130.43.2.2:

```
# specify NTP version 1
peer 130.43.2.2 version 1 # apple.com (running ntpd version 1)
peer 130.43.2.2 version 2 # apple.com (running xntpd version 2)
```

If you are testing xntpd(1M) 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, popularly called NTP Version 0.

There are a few other items to watch when converting an ntpd configuration file for use with xntpd(1M). The first is to reconsider the precision entry from the configuration file, if there is one. There was a time when the precision claimed by a server was mostly commentary, with no particularly useful purpose. This is no longer the case, however, and so changing the precision a server claims should only be done with some consideration as to how this alters the performance of the server. The default precision claimed by xntpd(1M) will be right for most situations. A section later on will deal with when and how it is appropriate to change a server's precision without doing things you don't intend.

Second, note that in the example configuration file above numeric addresses are used in the peer and server declarations. It is also possible to use names requiring resolution instead, but only if some additional configuration is done (xntpd(1M) doesn't include the resolver routines itself, and requires that a second program be used to do name resolution). If you find numeric addresses offensive, see below.

Finally, “passive” and “client” entries in an ntpd configuration file have no useful equivalent semantics for xntpd(1M) and should be deleted. xntpd(1M) won't reset the kernel variable tickadj(1M) when it starts, so you can remove anything dealing with this in the configuration file. The configuration of radio clock peers is done using different language in xntpd(1M) configuration files, so you will need to delete these entries from your ntpd configuration file and see below for the equivalent language.

Traffic Monitoring

xntpd(1M) handles peers whose stratum is higher than the stratum of the local server and pollers using client mode by a fast path that 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, xntpd(1M) implements a traffic monitoring facility that records the source address and a minimal amount of other information from each packet received by the server. This can be enabled by adding the following line to the server's configuration file:

```
# enable monitoring feature
monitor yes
```

The recorded information can be displayed using the xntpdc(1M) query program, described briefly below.

Address-and-Mask Restrictions

The address-and-mask configuration facility supported by xntpd(1M) 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 well designed, 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:

```
(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 runtime configuration. There are many more flags, see the `xntpd(1M)` man 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 runtime 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 that 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

`xntpd(1M)` 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-bit crypto checksum of the contents of the packet computed using either the Data Encryption Standard (DES) or Message Digest (MD5) algorithms. Note that this implementation provides only the MD5 algorithm. Also 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. 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 “authenticate” configuration declaration. In particular, in authenticated mode, peers that send unauthenticated packets, peers that send authenticated packets that the local server is unable to decrypt and peers that 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 that requires authenticated time and trusts the server, but 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
authenticate yes# enable authentication
keys /etc/opt/SUNWxntp/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 0.000047# authentication delay (Sun4c/50 IPX DES)
authdelay 0.000094# authentication delay (Sun4c/50 IPX MD5)
```

There are a couple of previously unmentioned things in here. The `authenticate yes` line enables authentication processing, while the `keys /etc/opt/SUNWxntp/ntp.keys` specifies the path to the keys file (see below and the `xntpd(1M)` man 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 minimrequestkey 15izing the frequency of delicate key-distribution procedures. The “requestkey 15" line establishes the key to be used for mode-6 control messages as specified in RFC 1305 and used by the `ntpq(1M)` utility program, while the “controlkey 15" establishes the key to be used for mode-7 private control messages used by the `xntpdc(1M)` utility program these keys are used to prevent unauthorized modification of daemon variables.

The “authdelay” declaration is an estimate of the amount of 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), and is used as a correction for the transmit timestamp. This can be computed for your CPU by the authspeed program included in the `opt/SUNWxntp/bin` in the xntp3 distribution. The usage is illustrated to the following:

```
# for DES keys
authspeed -n 30000 auth.samplekeys
# for MD5 keys
authspeed -nd 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 the server). The contents of this file might look like the following. Note, however, that this implementation of NTP supports only the MD5 algorithm.

```
# 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. 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.

Query Programs

Three utility query programs are included with the xntp3 distribution, `ntpq(1M)`, `ntptrace(1M)` and `xntpdc(1M)`. `ntpq(1M)` is a rather handy program that 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(1M)` 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.

Xnptdc is a horrid program that uses NTP private mode-7 control messages to query local or remote servers. The format and contents of these messages are specific to `xntpd(1M)`. 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 `xntpdc(1M)` is that it provides a user interface to the runtime reconfiguration facility.

See the respective man pages for details on the use of these programs. The primary reason for mentioning them here is to point out an inconsistency that can be awfully annoying if it catches you, and that is worth keeping firmly in mind. Both `xntpdc(1M)` and `xntpd(1M)` demand that anything that has dimensions of time be specified in units of seconds, both in the configuration file and when doing runtime reconfiguration. Both programs also print the values in seconds. `ntpq(1M)` on the other hand, obeys the standard by printing all time values in milliseconds. This makes the process of looking at values with ntpq(1M) and then changing them in the configuration file or with xntpdc(1M) very prone to errors (by three orders of magnitude). I wish this problem didn't exist, but xntpd(1M) and its love of seconds predate the mode-6 protocol and the latter's (Fuzzball-inspired) millisecond orientation, making the inconsistency unresolvable without considerable work.

Runtime Reconfiguration

xntpd(1M) was written specifically to allow its configuration to be fully modifiable at runtime. Indeed, the only way to configure the server is at runtime. The configuration file is read only after the rest of the server has been initialized into a running, but default unconfigured, 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, runtime 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 xntpdc(1M) 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(1M) program provides generic support for the latter.

Mode-6 and mode-7 messages that 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(1M)) and mode-7 (xntpdc(1M)) 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 runtime 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

`xntpd(1M)` includes the capability to specify host names requiring resolution in “peer” and “server” declarations in the configuration file. There are several reasons why this was not permitted in the past. Chief among these is the fact that name service is unreliable and the interface to the UNIX resolver routines is synchronous. The hang-ups 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.

Instead of running the resolver itself the daemon can defer this task to a separate program, `xntpres`. 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 an instance of `xntpres` with the temporary file as an argument. The server then continues on normally but with the offending peers/servers omitted from its configuration.

When `xntpres` successfully resolves a name from this file, it configures the associated entry into the server using the same mode-7 runtime reconfiguration facility that `xntpdc(1M)` uses. If temporary resolver failures occur, `xntpres` will periodically retry the offending requests until a definite response is received. The program will continue to run until all entries have been resolved.

There are several configuration requirements if `xntpres` is to be used. The path to the `xntpres` program must be made known to the daemon via a "resolver" configuration entry, and mode-7 runtime reconfiguration must be enabled. The following fragment might be used to accomplish this:

```
# specify host name resolver data
resolver /opt/SUNWxntp/bin
keys /etc/opt/SUNWxntp
requestkey 65535
```

Note that `xntpres` sends packets to the server with a source address of 127.0.0.1. You should obviously avoid "restrict" modification requests from this address or `xntpres` will fail.

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.

However, in the case of certain notorious platforms, in particular Sun 4s, the 100-ppm tolerance is routinely violated. In such cases it may be necessary to adjust the values of certain kernel variables; in particular, *tick* and `tickadj(1M)`. The variable *tick* is the increment in microseconds added to the system time on each interval-timer interrupt, while the variable `tickadj(1M)` is used by the time adjustment code as a slew rate. When the time is being adjusted via a call to the system routine `adjtime(2)`, the kernel increases or reduces *tick* by `tickadj(1M)` microseconds until the specified adjustment has been completed. Unfortunately, in most UNIX implementations the *tick* increment must be either zero or plus/minus exactly `tickadj(1M)` microseconds, meaning that adjustments are truncated to be an integral multiple of `tickadj(1M)` (this latter behavior is a misfeature, and is the only reason the `xntpd(1M)` code needs to concern itself with the internal implementation of `adjtime(2)` 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 `xntpd(1M)` daemon reads the values of *tick* and `tickadj(1M)` from `/dev/kmem` when it starts. It then ensures that all adjustments given to `adjtime(2)` are an even multiple of `tickadj(1M)` microseconds and computes the largest adjustment that can be completed in the adjustment interval (using both the value of `tickadj(1M)` and the value of tick) so it can avoid exceeding this limit.

Unfortunately, the value of `tickadj(1M)` set by default is almost always too large for `xntpd(1M)`. NTP operates by continuously making small adjustments to the clock, usually at one-second intervals. If `tickadj(1M)` is set too large, the adjustments will disappear in the roundoff; while, if `tickadj(1M)` 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 *tick* and `tickadj(1M)`, it will not change the values, even if they are unsuitable.

NOTE: If you need to reinitialize the `xntp` daemon, use the following start-up script (you need to be root in order to execute it):

```
tickadj -s -a 1000
ntpdate -v server1 server2
sleep 20
ntpdate -v server1 server2
sleep 20
tickadj -a 200
xntpd
```

where *server1* and *server2* are the Internet addresses of the time servers.

The `tickadj(1M)` program can reset several other kernel variables if asked. It can also change the value of *tick* if asked, this being necessary on a few machines with very broken clocks, like Sun 4s. With these machines it should also 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 `xntpd(1M)` is trying to keep it under control.

In order to maintain reasonable correctness bounds, as well as reasonably good accuracy with acceptable polling intervals, `xntpd(1M)` will complain if the frequency error is greater than 100 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(1M)` useful on Sun 4s is:

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

which sets *tick* 100 ppm fast, `tickadj(1M)` 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.

Tuning Your Subnet

There are several parameters available for tuning the NTP subnet for maximum accuracy and minimum jitter. Two important parameters are the "precision" and "prefer" configuration declarations. The precision declaration specifies the number of significant bits of the system clock representation relative to one second. For instance, the default value of -6 corresponds to 1/64 second or about 16 milliseconds.

The NTP protocol makes use of the precision parameter in several places. It is included in packets sent to peers and is used by them to calculate the maximum absolute error and maximum statistical error. When faced with selecting one of several servers of the same stratum and about the same network path delay for synchronization purposes, clients will usually prefer to synchronize to those servers claiming the smallest (most negative) precision, since this maximizes the accuracy and minimizes the jitter apparent to application programs running on the client platform. Therefore, when the maximum attainable accuracy is required, it is important that every platform configure an accurate value for the precision variable. This can be done using the optional "precision" declaration in the configuration file:

```
# precision declaration
precision -18 # for microsecond clocks (Sun 4s, DEC 5000/240)
```

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 128.4.1.1 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

`xntpd(1M)` understands leap seconds and will attempt to take appropriate action when one occurs. In principle, every host running `xntpd(1M)` will insert a leap second in the local timescale in precise synchronization with UTC. This requires that the leap-warning bits be manually 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 `xntpd(1M)` and the time-conversion routines of each host. The implementation is independent of the idiosyncracies 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 `xntpd(1M)` 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

xntpd(1M) 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 instances 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 runtime 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 xntpdc(1M) 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 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 that 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 xntpd(1M) man page.

Towards the Ultimate Tick

This section considers issues in providing precision time synchronization in NTP subnets that 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 1-pps signal produced by a calibrated cesium clock, it is possible to remove the jitter and reset 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 xntp3 distribution are used together with the 1-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 1-pps signal that 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 100 microseconds with a fast RISC- based platform.

There are three ways to implement 1-pps support, depending on the radio clock model, platform model and serial line interface. Each of these requires circuitry to convert the TTL signal produced by most clocks to the EIA levels used by most serial interfaces. Besides being useful for this purpose, this device includes an inexpensive modem designed for use with the Canadian CHU time/frequency radio station.

Note – The Sun-supplied binary does not support the implementations that follow. Proceed to the next section, “Swatting Bugs”.

In order to select the appropriate implementation, it is important to understand the underlying 1-pps mechanism used by `xntpd(1M)`. The 1-pps support depends on a continuous source of 1-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 1-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 1-pps support is disabled and the timecode used directly to control the local clock.

The first method of implementation uses a dedicated serial port and either the BSD line discipline or System V streams module, which can be found in the kernel directory of the xntp3 distribution. This method can be used with any radio clock or in the absence of any clock. The line discipline and streams modules take receive timestamps in the kernel, specifically the interrupt routine of the serial port hardware driver. Using this method the port is dedicated to serve the 1-pps signal and cannot be used for other purposes. Instructions for implementing the feature, which requires rebuilding the kernel, are included in the modules themselves. Note that xndpd must be compiled with the -DPPSDEV compiler switch in this case. There is an inherent error in this method due to the latency of the interrupt system and remaining serial-line protocol modules in the order of a millisecond with Sun 4s. While the jitter in this latency is unavoidable, the systematic component can be calibrated out using a special configuration declaration:

```
# pps delay and baud rate
pps delay.0017 baud 19200 # pps delay (ms) and baud rate
```

Note that the delay defaults to zero and the baud to 38400.

The second method uses mechanisms embedded in the radio clock driver, which call the 1-pps support directly and do not require a dedicated serial port. Currently, only the DCF77 (German radio time service) driver uses this method.

The third method and the most accurate and intrusive of all uses the carrier-detect modem-control lead monitored by the serial port driver. This method can be used with any radio clock and 1-pps interface mentioned above. It requires in addition to a special streams module, replacement of the kernel high resolution time-of-day clock routine. This method is applicable only to Sun 4 platforms running SunOS 4.1.1 and then only with either of the two onboard serial ports. It does not work with other platforms, operating systems or external (SBus) serial multiplexors.

Swatting Bugs

In the Ultra Enterprise 10000 system the `xntpd(1M)` daemon and utility programs (`ntpq(1M)`, `ntptrace(1M)` and `xntpdc(1M)`) are installed in the `/opt/SUNWxntp/bin` directory along with the key file (`ntp.keys`), while the configuration file (`ntp.conf`) and drift file (`ntp.drift`) are installed in the `/etc/opt/SUNWxntp` directory. The daemon is usually started from the rc.local shell script at system boot time, but could be started (and stopped) at other times for debugging, etc. How do you verify that the daemon can form associations with remote peers and verify correct synchronization? For this you need the `ntpq(1M)` utility described in the `ntpq(1M)` man page.

After starting the daemon, run the `ntpq(1M)` program using the -n switch, which will avoid possible distractions due to name resolutions. Use the peer command to display a billboard showing the status of configured peers and possibly other clients poking the daemon. After operating for a few minutes, the display should be something like:

remote	**refid**	**st**	**when**	**poll**	**reach**	**delay**	**offset**	**disp**
+128.4.2.6	132.249.16.1	2	131	256	373	9.89	16.28	23.25
*128.4.1.20	.WWVB.	1	137	256	377	280.62	21.74	20.23
-128.8.2.88	128.8.10.1	2	49	128	376	294.14	5.94	17.47
+128.4.2.17	.WWVB.	1	173	256	377	279.95	20.56	16.40

The hosts shown in the “remote” column should agree with the entries in the configuration file, plus any peers not mentioned in the file at the same or lower than your stratum that happen to be configured to peer with you. The “refid” entry shows the current source of synchronization for that peer, while the “st” reveals its stratum and the “poll” entry the polling interval, in seconds. The “when” entry shows the time since the peer was last heard, in seconds, while the “reach” entry shows the status of the reachability register (see specification), which is in octal format. The remaining entries show the latest delay, offset and dispersion computed for the peer, in milliseconds.

Index