[Team LiB] PREVIOUS NEXT

21.11 Simple Network Time Protocol (SNTP)

NTP is a sophisticated protocol for synchronizing clocks across a WAN or a LAN, and can often achieve millisecond accuracy. RFC 1305 [Mills 1992] describes the protocol in detail and RFC 2030 [Mills 1996] describes SNTP, a simplified but protocol-compatible version intended for hosts that do not need the complexity of a complete NTP implementation. It is common for a few hosts on a LAN to synchronize their clocks across the Internet to other NTP hosts and then redistribute this time on the LAN using either broadcasting or multicasting.

In this section, we will develop an SNTP client that listens for NTP broadcasts or multicasts on all attached networks and then prints the time difference between the NTP packet and the host's current time-of-day. We do not try to adjust the time-of-day, as that takes superuser privileges.

The file `ntp.h`, shown in Figure 21.20, contains some basic definitions of the NTP packet format.

Figure 21.20 `ntp.h` header: NTP packet format and definitions.

ssntp/ntp.h

```
 1 #define JAN_1970    2208988800UL     /* 1970 - 1900 in seconds */

 2 struct l_fixedpt {               /* 64-bit fixed-point */
 3     uint32_t int_part;
 4     uint32_t fraction;
 5 };

 6 struct s_fixedpt {               /* 32-bit fixed-point */
 7     uint16_t int_part;
 8     uint16_t fraction;
 9 };

10 struct ntpdata {                 /* NTP header */
11     u_char  status;
12     u_char  stratum;
13     u_char  ppoll;
14     int     precision:8;
15     struct s_fixedpt distance;
16     struct s_fixedpt dispersion;
17     uint32_t refid;
18     struct l_fixedpt reftime;
19     struct l_fixedpt org;
20     struct l_fixedpt rec;
21     struct l_fixedpt xmt;
22 };

23 #define VERSION_MASK    0x38
24 #define MODE_MASK       0x07

25 #define MODE_CLIENT     3
26 #define MODE_SERVER     4
27 #define MODE_BROADCAST  5
```

2–22 `l_fixedpt` defines the 64-bit fixed-point values used by NTP for timestamps and `s_fixedpt` defines the 32-bit fixed-point values that are also used by NTP. The `ntpdata` structure is the 48-byte NTP packet format.

Figure 21.21 shows the `main` function.

Get multicast IP address

12–14 When the program is executed, the user must specify the multicast address to join as the command-line argument. With IPv4, this would be 224.0.1.1 or the name `ntp.mcast.net`. With IPv6, this would be `ff05::101` for the site-local scope NTP. Our `udp_client` function allocates space for a socket address structure of the correct type (either IPv4 or IPv6) and stores the multicast address and port in that structure. If this program is run on a host that does not support multicasting, any IP address can be specified, as only the address family and port are used from this structure. Note that our `udp_client` function does not `bind` the address to the socket; it just creates the socket and fills in the socket address structure.

Figure 21.21 `main` function.

ssntp/main.c

```
 1 #include    "sntp.h"
 2 int
 3 main(int argc, char **argv)
 4 {
 5     int     sockfd;
 6     char    buf[MAXLINE];
 7     ssize_t n;
 8     socklen_t salen, len;
 9     struct ifi_info *ifi;
10     struct sockaddr *mcastsa, *wild, *from;
11     struct timeval now;

12     if (argc != 2)
13         err_quit("usage: ssntp <IPaddress>");

14     sockfd = Udp_client(argv[1], "ntp", (void **) &mcastsa, &salen);

15     wild = Malloc(salen);
16     memcpy(wild, mcastsa, salen);   /* copy family and port */
17     sock_set_wild(wild, salen);
18     Bind(sockfd, wild, salen);  /* bind wildcard */

19 #ifdef  MCAST
20         /* obtain interface list and process each one */
21     for (ifi = Get_ifi_info(mcastsa->sa_family, 1); ifi != NULL;
22          ifi = ifi->ifi_next) {
23         if (ifi->ifi_flags & IFF_MULTICAST) {
24             Mcast_join(sockfd, mcastsa, salen, ifi->ifi_name, 0);
25             printf("joined %s on %s\n",
26                    Sock_ntop(mcastsa, salen), ifi->ifi_name);
27         }
28     }
29 #endif

30     from = Malloc(salen);
31     for ( ; ; ) {
32         len = salen;
33         n = Recvfrom(sockfd, buf, sizeof(buf), 0, from, &len);
34         Gettimeofday(&now, NULL);
35         sntp_proc(buf, n, &now);
36     }
37 }
```

Bind wildcard address to socket

15–18 We allocate space for another socket address structure and fill it in by copying the structure that was filled in by `udp_client`. This sets the address family and port. We call our `sock_set_wild` function to set the IP address to the wildcard and then call `bind`.

Get interface list

20–22 Our `get_ifi_info` function returns information on all the interfaces and addresses. The address family that we ask for is taken from the socket address structure that was filled in by `udp_client` based on the command-line argument.

Join multicast group

23–27 We call our `mcast_join` function to join the multicast group specified by the command-line argument for each multicast-capable interface. All these joins are done on the one socket that this program uses. As we said earlier, there is normally a limit of `IP_MAX_MEMBERSHIPS` (often 20) joins per socket, but few multihomed hosts have that many interfaces.

Read and process all NTP packets

30–36 Another socket address structure is allocated to hold the address returned by `recvfrom` and the program enters an infinite loop, reading all the NTP packets that the host receives and calling our `sntp_proc` function (described next) to process the packet. Since the socket was bound to the wildcard address, and since the multicast group was joined on all multicast-capable interfaces, the socket should receive any unicast, broadcast, or multicast NTP packet that the host receives. Before calling `sntp_proc`, we call `gettimeofday` to fetch the current time, because `sntp_proc` calculates the difference between the time in the packet and the current time.

Our `sntp_proc` function, shown in Figure 21.22, processes the actual NTP packet.

Validate packet

10-21 We first check the size of the packet and then print the version, mode, and server stratum. If the mode is `MODE_CLIENT`, the packet is a client request, not a server reply, and we ignore it.

Obtain transmit time from NTP packet

22-33 The field in the NTP packet that we are interested in is `xmt`, the transmit timestamp, which is the 64-bit fixed-point time at which the packet was sent by the server. Since NTP timestamps count seconds beginning in 1900 and Unix timestamps count seconds beginning in 1970, we first subtract `JAN_1970` (the number of seconds in these 70 years) from the integer part.

The fractional part is a 32-bit unsigned integer between 0 and 4,294,967,295, inclusive. This is copied from a 32-bit integer (`useci`) to a double-precision floating-point variable (`usecf`) and then divided by 4,294,967,296 (2^{32}). The result is greater than or equal to 0.0 and less than 1.0. We multiply this by 1,000,000, the number of microseconds in a second, storing the result as a 32-bit unsigned integer in the variable `useci`. This is the number of microseconds and will be between 0 and 999,999 (see Exercise 21.5). We convert to microseconds because the Unix timestamp returned by `gettimeofday` is returned as two integers: the number of seconds since January 1, 1970, UTC, along with the number of microseconds. We then calculate and print the difference between the host's time-of-day and the NTP server's time-of-day, in microseconds.

Figure 21.22 `sntp_proc` function: processes the NTP packet.

ssntp/sntp_proc.c

```
 1 #include    "sntp.h"

 2 void
 3 sntp_proc(char *buf, ssize_t n, struct timeval *nowptr)
 4 {
 5     int     version, mode;
 6     uint32_t nsec, useci;
 7     double  usecf;
 8     struct timeval diff;
 9     struct ntpdata *ntp;

10     if (n < (ssize_t) sizeof(struct ntpdata)) {
11         printf("\npacket too small: %d bytes\n", n);
12         return;
13     }

14     ntp = (struct ntpdata *) buf;
15     version = (ntp->status & VERSION_MASK) >> 3;
16     mode = ntp->status & MODE_MASK;
17     printf("\nv%d, mode %d, strat %d, ", version, mode, ntp->stratum);
18     if (mode == MODE_CLIENT) {
19         printf("client\n");
20         return;
21     }

22     nsec = ntohl(ntp->xmt.int_part) - JAN_1970;
23     useci = ntohl(ntp->xmt.fraction);   /* 32-bit integer fraction */
24     usecf = useci;                /* integer fraction -> double */
25     usecf /= 4294967296.0;       /* divide by 2**32 -> [0, 1.0) */
26     useci = usecf * 1000000.0;   /* fraction -> parts per million */

27     diff.tv_sec = nowptr->tv_sec - nsec;
28     if ( (diff.tv_usec = nowptr->tv_usec - useci) < 0) {
29         diff.tv_usec += 1000000;
30         diff.tv_sec--;
31     }
32     useci = (diff.tv_sec * 1000000) + diff.tv_usec; /* diff in microsec */
33     printf("clock difference = %d usec\n", useci);
34 }
```

One thing that our program does not take into account is the network delay between the server and the client. But we assume that the NTP packets are normally received as a broadcast or multicast on a LAN, in which case, the network delay should be only a few milliseconds.

If we run this program on our host `macosx` with an NTP server on our host `freebsd4`, which is multicasting NTP packets to the Ethernet every 64 seconds, we have the following output:

```
macosx # ssntp 224.0.1.1
joined 224.0.1.1.123 on lo0
joined 224.0.1.1.123 on en1

v4, mode 5, strat 3, clock difference = 661 usec

v4, mode 5, strat 3, clock difference = -1789 usec
```

```
v4, mode 5, strat 3, clock difference = -2945 usec

v4, mode 5, strat 3, clock difference = -3689 usec

v4, mode 5, strat 3, clock difference = -5425 usec

v4, mode 5, strat 3, clock difference = -6700 usec

v4, mode 5, strat 3, clock difference = -8520 usec
```

To run our program, we first terminated the normal NTP server running on this host, so when our program starts, the time is very close to the server's time. We see this host lost 9181 microseconds in the 384 seconds we ran the program, or about 2 seconds in 24 hours.

[Team LiB] PREVIOUS NEXT