

---

## Protokoll: SNTP

**By rac**

Published: 15.01.2008 - 09:43

Java Implemntation eines SNTP clients

Package Generation in this example is partly taken from DemoNtpMessage.java and found on the Internet.

- [DemoNtpMessage.java](#)
- [SNTPclient.java](#)

[< echoserverThreaded.java](#) [up](#) [DemoNtpMessage.java](#) [>](#)

---

## DemoNtpMessage.java

**By rac**

Published: 15.01.2008 - 09:49

```
1.
package hsz.ITDP.rac.uebungen.SNTPclientServer;

2.

3.
import java.text.DecimalFormat;

4.
import java.text.SimpleDateFormat;

5.
import java.util.Date;

6.

7.

8.
/**
9.
 * This class represents a NTP message, as specified in RFC 2030. The message
```

- 
10.
    - \* format is compatible with all versions of NTP and SNTP.
  
  11.
    - \*
  
  12.
    - \* This class does not support the optional authentication protocol, and
  
  13.
    - \* ignores the key ID and message digest fields.
  
  14.
    - \*
  
  15.
    - \* For convenience, this class exposes message values as native Java types, not
  
  16.
    - \* the NTP-specified data formats. For example, timestamps are
  
  17.
    - \* stored as doubles (as opposed to the NTP unsigned 64-bit fixed point
  
  18.
    - \* format).
  
  19.
    - \*
  
  20.
    - \* However, the constructor `NtpMessage(byte[])` and the method `toByteArray()`
  
  21.
    - \* allow the import and export of the raw NTP message format.
  
  - 22.

---

\*

23.

\*

24.

\* Usage example

25.

\*

26.

\* // Send message

27.

\* DatagramSocket socket = new DatagramSocket();

28.

\* InetAddress address = InetAddress.getByName("ntp.cais.rnp.br");

29.

\* byte[] buf = new NtpMessage().toByteArray();

30.

\* DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 123);

31.

\* socket.send(packet);

32.

\*

33.

\* // Get response

34.

\* socket.receive(packet);

---

35.  
\* `System.out.println(msg.toString());`

36.  
\*

37.  
\*

38.  
\* This code is copyright (c) Adam Buckley 2004

39.  
\*

40.  
\* This program is free software; you can redistribute it and/or modify it

41.  
\* under the terms of the GNU General Public License as published by the Free

42.  
\* Software Foundation; either version 2 of the License, or (at your option)

43.  
\* any later version. A HTML version of the GNU General Public License can be

44.  
\* seen at <http://www.gnu.org/licenses/gpl.html>

45.  
\*

46.  
\* This program is distributed in the hope that it will be useful, but WITHOUT

47.

---

\* ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or

48.

\* FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for

49.

\* more details.

50.

\*

51.

\*

52.

\* Comments for member variables are taken from RFC2030 by David Mills,

53.

\* University of Delaware.

54.

\*

55.

\* Number format conversion code in NtpMessage(byte[] array) and toByteArray()

56.

\* inspired by <http://www.pps.jussieu.fr/~jch/enseignement/reseaux/>

57.

\* NTPMessage.java which is copyright (c) 2003 by Juliusz Chroboczek

58.

\*

59.

\* @author Adam Buckley

---

60.  
\*/

61.  
public class DemoNtpMessage

62.  
{

63.  
 /\*\*

64.  
 \* This is a two-bit code warning of an impending leap second to be

65.  
 \* inserted/deleted in the last minute of the current day. It's values

66.  
 \* may be as follows:

67.  
 \*

68.  
 \* Value Meaning

69.  
 \* -----

70.  
 \* 0 no warning

71.  
 \* 1 last minute has 61 seconds

72.

---

\* 2 last minute has 59 seconds)

73.

\* 3 alarm condition (clock not synchronized)

74.

\*/

75.

public byte leapIndicator = 0;

76.

77.

78.

/\*\*

79.

\* This value indicates the NTP/SNTP version number. The version number

80.

\* is 3 for Version 3 (IPv4 only) and 4 for Version 4 (IPv4, IPv6 and OSI).

81.

\* If necessary to distinguish between IPv4, IPv6 and OSI, the

82.

\* encapsulating context must be inspected.

83.

\*/

84.

public byte version = 3;

---

85.

86.

87.

/\*\*

88.

\* This value indicates the mode, with values defined as follows:

89.

\*

90.

\* Mode      Meaning

91.

\* ----      -----

92.

\* 0          reserved

93.

\* 1          symmetric active

94.

\* 2          symmetric passive

95.

\* 3          client

96.

\* 4          server

97.

- 
- \* 5 broadcast
  - 98.
    - \* 6 reserved for NTP control message
  - 99.
    - \* 7 reserved for private use
  - 100.
    - \*
  - 101.
    - \* In unicast and anycast modes, the client sets this field to 3 (client)
  - 102.
    - \* in the request and the server sets it to 4 (server) in the reply. In
  - 103.
    - \* multicast mode, the server sets this field to 5 (broadcast).
  - 104.
    - \*/
  - 105.
    - public byte mode = 0;
  - 106.
  - 107.
  - 108.
    - /\*\*
  - 109.
    - \* This value indicates the stratum level of the local clock, with values

---

110.  
\* defined as follows:

111.  
\*

112.  
\* Stratum Meaning

113.  
\* -----

114.  
\* 0 unspecified or unavailable

115.  
\* 1 primary reference (e.g., radio clock)

116.  
\* 2-15 secondary reference (via NTP or SNTP)

117.  
\* 16-255 reserved

118.  
\*/

119.  
public short stratum = 0;

120.

121.

122.

---

/\*\*

123.

\* This value indicates the maximum interval between successive messages,

124.

\* in seconds to the nearest power of two. The values that can appear in

125.

\* this field presently range from 4 (16 s) to 14 (16284 s); however, most

126.

\* applications use only the sub-range 6 (64 s) to 10 (1024 s).

127.

\*/

128.

public byte pollInterval = 0;

129.

130.

131.

/\*\*

132.

\* This value indicates the precision of the local clock, in seconds to

133.

\* the nearest power of two. The values that normally appear in this field

134.

\* range from -6 for mains-frequency clocks to -20 for microsecond clocks

---

135.

\* found in some workstations.

136.

\*/

137.

public byte precision = 0;

138.

139.

140.

/\*\*

141.

\* This value indicates the total roundtrip delay to the primary reference

142.

\* source, in seconds. Note that this variable can take on both positive

143.

\* and negative values, depending on the relative time and frequency

144.

\* offsets. The values that normally appear in this field range from

145.

\* negative values of a few milliseconds to positive values of several

146.

\* hundred milliseconds.

147.

---

\*/

148.

public double rootDelay = 0;

149.

150.

151.

/\*\*

152.

\* This value indicates the nominal error relative to the primary reference

153.

\* source, in seconds. The values that normally appear in this field

154.

\* range from 0 to several hundred milliseconds.

155.

\*/

156.

public double rootDispersion = 0;

157.

158.

159.

/\*\*

- 
160.  
\* This is a 4-byte array identifying the particular reference source.
161.  
\* In the case of NTP Version 3 or Version 4 stratum-0 (unspecified) or
162.  
\* stratum-1 (primary) servers, this is a four-character ASCII string, left
163.  
\* justified and zero padded to 32 bits. In NTP Version 3 secondary
164.  
\* servers, this is the 32-bit IPv4 address of the reference source. In NTP
165.  
\* Version 4 secondary servers, this is the low order 32 bits of the latest
166.  
\* transmit timestamp of the reference source. NTP primary (stratum 1)
167.  
\* servers should set this field to a code identifying the external
168.  
\* reference source according to the following list. If the external
169.  
\* reference is one of those listed, the associated code should be used.
170.  
\* Codes for sources not listed can be contrived as appropriate.
171.  
\*
- 172.

- 
- | * Code | External Reference Source                                       |
|--------|---|
| 173.   | * ---- -----  |
| 174.   | * LOCL uncalibrated local clock used as a primary reference for |
| 175.   | * a subnet without external means of synchronization            |
| 176.   | * PPS atomic clock or other pulse-per-second source             |
| 177.   | * individually calibrated to national standards                 |
| 178.   | * ACTS NIST dialup modem service                                |
| 179.   | * USNO USNO modem service                                       |
| 180.   | * PTB PTB (Germany) modem service                               |
| 181.   | * TDF Allouis (France) Radio 164 kHz                            |
| 182.   | * DCF Mainflingen (Germany) Radio 77.5 kHz                      |
| 183.   | * MSF Rugby (UK) Radio 60 kHz                                   |
| 184.   | * WWV Ft. Collins (US) Radio 2.5, 5, 10, 15, 20 MHz             |

- 
185.  
\* WWVB Boulder (US) Radio 60 kHz
186.  
\* WWVH Kauai Hawaii (US) Radio 2.5, 5, 10, 15 MHz
187.  
\* CHU Ottawa (Canada) Radio 3330, 7335, 14670 kHz
188.  
\* LORC LORAN-C radionavigation system
189.  
\* OMEG OMEGA radionavigation system
190.  
\* GPS Global Positioning Service
191.  
\* GOES Geostationary Orbit Environment Satellite
192.  
\*/
193.  
public byte[] referenceIdentifier = {0, 0, 0, 0};
- 194.
- 195.
196.  
/\*\*
- 197.

---

\* This is the time at which the local clock was last set or corrected, in

198.

\* seconds since 00:00 1-Jan-1900.

199.

\*/

200.

public double referenceTimestamp = 0;

201.

202.

203.

/\*\*

204.

\* This is the time at which the request departed the client for the

205.

\* server, in seconds since 00:00 1-Jan-1900.

206.

\*/

207.

public double originateTimestamp = 0;

208.

209.

---

210.

/\*\*

211.

\* This is the time at which the request arrived at the server, in seconds

212.

\* since 00:00 1-Jan-1900.

213.

\*/

214.

public double receiveTimestamp = 0;

215.

216.

217.

/\*\*

218.

\* This is the time at which the reply departed the server for the client,

219.

\* in seconds since 00:00 1-Jan-1900.

220.

\*/

221.

public double transmitTimestamp = 0;

222.

---

223.

224.

225.  
/\*\*

226.  
\* Constructs a new NtpMessage from an array of bytes.

227.  
\*/

228.  
public DemoNtpMessage(byte[] array)

229.  
{

230.  
    // See the packet format diagram in RFC 2030 for details

231.  
    leapIndicator = (byte) ((array[0] >> 6) & 0x3);

232.  
    version = (byte) ((array[0] >> 3) & 0x7);

233.  
    mode = (byte) (array[0] & 0x7);

234.  
    stratum = unsignedByteToShort(array[1]);

---

235.           pollInterval = array[2];

236.           precision = array[3];

237.

238.           rootDelay = (array[4] \* 256.0) +

239.                           unsignedByteToShort(array[5]) +

240.                           (unsignedByteToShort(array[6]) / 256.0) +

241.                           (unsignedByteToShort(array[7]) / 65536.0);

242.

243.           rootDispersion = (unsignedByteToShort(array[8]) \* 256.0) +

244.                           unsignedByteToShort(array[9]) +

245.                           (unsignedByteToShort(array[10]) / 256.0) +

246.                           (unsignedByteToShort(array[11]) / 65536.0);

247.

---

```
248.     referencIdentifier[0] = array[12];

249.     referencIdentifier[1] = array[13];

250.     referencIdentifier[2] = array[14];

251.     referencIdentifier[3] = array[15];

252.

253.     referenceTimestamp = decodeTimestamp(array, 16);

254.     originateTimestamp = decodeTimestamp(array, 24);

255.     receiveTimestamp = decodeTimestamp(array, 32);

256.     transmitTimestamp = decodeTimestamp(array, 40);

257.     }

258.

259.
```

---

260.

261.

/\*\*

262.

\* Constructs a new NtpMessage in client -> server mode, and sets the

263.

\* transmit timestamp to the current time.

264.

\*/

265.

public DemoNtpMessage()

266.

{

267.

// Note that all the other member variables are already set with

268.

// appropriate default values.

269.

this.mode = 3;

270.

this.transmitTimestamp = ([System](#).currentTimeMillis()/1000.0) + 2208988800.0;

271.

}

272.

---

273.

274.

275.  
/\*\*

276.  
\* This method constructs the data bytes of a raw NTP packet.

277.  
\*/

278.  
public byte[] toByteArray()

279.  
{

280.  
    // All bytes are automatically set to 0

281.  
    byte[] p = new byte[48];

282.

283.  
    p[0] = (byte) (leapIndicator << 6 | version << 3 | mode);

284.  
    p[1] = (byte) stratum;

---

```
285.     p[2] = (byte) pollInterval;

286.     p[3] = (byte) precision;

287.

288.     // root delay is a signed 16.16-bit FP, in Java an int is 32-bits

289.     int l = (int) (rootDelay * 65536.0);

290.     p[4] = (byte) ((l >> 24) & 0xFF);

291.     p[5] = (byte) ((l >> 16) & 0xFF);

292.     p[6] = (byte) ((l >> 8) & 0xFF);

293.     p[7] = (byte) (l & 0xFF);

294.

295.     // root dispersion is an unsigned 16.16-bit FP, in Java there are no

296.     // unsigned primitive types, so we use a long which is 64-bits

297.
```

---

```
    long ul = (long) (rootDispersion * 65536.0);

298.    p[8] = (byte) ((ul >> 24) & 0xFF);

299.    p[9] = (byte) ((ul >> 16) & 0xFF);

300.    p[10] = (byte) ((ul >> 8) & 0xFF);

301.    p[11] = (byte) (ul & 0xFF);

302.

303.    p[12] = referencIdentifier[0];

304.    p[13] = referencIdentifier[1];

305.    p[14] = referencIdentifier[2];

306.    p[15] = referencIdentifier[3];

307.

308.    encodeTimestamp(p, 16, referenceTimestamp);

309.    encodeTimestamp(p, 24, originateTimestamp);
```

---

```
310.         encodeTimestamp(p, 32, receiveTimestamp);

311.         encodeTimestamp(p, 40, transmitTimestamp);

312.

313.         return p;

314.     }

315.

316.

317.

318.     /**

319.      * Returns a string representation of a NtpMessage

320.      */

321.     public String toString()

322.
```

---

```
{  
323.     String precisionStr =  
324.         new DecimalFormat ("0.##E0").format( Math .pow(2, precision));  
325.  
326.     return "Leap indicator: " + leapIndicator + "n" +  
327.         "Version: " + version + "n" +  
328.         "Mode: " + mode + "n" +  
329.         "Stratum: " + stratum + "n" +  
330.         "Poll: " + pollInterval + "n" +  
331.         "Precision: " + precision + " (" + precisionStr + " seconds)n" +  
332.         "Root delay: " + new DecimalFormat ("0.00").format(rootDelay*1000) + " msn  
" +  
333.         "Root dispersion: " + new DecimalFormat ("0.00").format(rootDispersion*1000  
) + " msn" +
```

---

334.                   "Reference identifier: " + referencIdentifierToString(referencIdentifier,  
stratum, version) + "n" +

335.                   "Reference timestamp: " + timestampToString(referenceTimestamp) + "n" +

336.                   "Originate timestamp: " + timestampToString(originateTimestamp) + "n" +

337.                   "Receive timestamp:  " + timestampToString(receiveTimestamp) + "n" +

338.                   "Transmit timestamp:  " + timestampToString(transmitTimestamp);

339.                   }

340.

341.

342.

343.                   /\*\*

344.                   \* Converts an unsigned byte to a short.  By default, Java assumes that

345.                   \* a byte is signed.

---

346.

\*/

347.

public static short unsignedByteToShort(byte b)

348.

{

349.

if((b & 0x80)==0x80) return (short) (128 + (b & 0x7f));

350.

else return (short) b;

351.

}

352.

353.

354.

355.

/\*\*

356.

\* Will read 8 bytes of a message beginning at <code>pointer

357.

---

```

* and return it as a double, according to the NTP 64-bit timestamp
* format.
*/
public static double decodeTimestamp(byte[] array, int pointer)
{
    double r = 0.0;

    for(int i=0; i
    {
        r += unsignedByteToShort(array[pointer+i]) * Math.pow(2, (3-i)*8);
    }

    return r;
}

/**
* Encodes a timestamp in the specified position in the message
*/
public static void encodeTimestamp(byte[] array, int pointer, double timestamp)
{
    // Converts a double into a 64-bit fixed point
for(int i=0; i
    {
        // 2^24, 2^16, 2^8, .. 2^-32

```

---

```
double base = Math.pow(2, (3-i)*8);

        // Capture byte value

array[pointer+i] = (byte) (timestamp / base);

        // Subtract captured value from remaining total

timestamp = timestamp - (double) (unsignedByteToShort(array[pointer+i]) * base);

    }

        // From RFC 2030: It is advisable to fill the non-significant

// low order bits of the timestamp with a random, unbiased

// bitstring, both to avoid systematic roundoff errors and as

// a means of loop detection and replay detection.

array[7] = (byte) (Math.random()*255.0);

    }

/**

 * Returns a timestamp (number of seconds since 00:00 1-Jan-1900) as a

 * formatted date/time string.

 */

public static String timestampToString(double timestamp)

{

    if(timestamp==0) return "0";
```

---

```
        // timestamp is relative to 1900, utc is used by Java and is relative
// to 1970
double utc = timestamp - (2208988800.0);

        // milliseconds

long ms = (long) (utc * 1000.0);

        // date/time

String date = new SimpleDateFormat("dd-MMM-yyyy HH:mm:ss").format(new Date(ms));

        // fraction

double fraction = timestamp - ((long) timestamp);

        String fractionSting = new DecimalFormat(".000000").format(fraction);

        return date + fractionSting;
}

/**
 * Returns a string representation of a reference identifier according
 * to the rules set out in RFC 2030.
 */

public static String referenceIdentifierToString(byte[] ref, short stratum, byte version)
```

---

```
{  
    // From the RFC 2030:  
  
    // In the case of NTP Version 3 or Version 4 stratum-0 (unspecified)  
    // or stratum-1 (primary) servers, this is a four-character ASCII  
    // string, left justified and zero padded to 32 bits.  
    if(stratum==0 || stratum==1)  
        {  
            return new String(ref);  
        }  
  
    // In NTP Version 3 secondary servers, this is the 32-bit IPv4  
    // address of the reference source.  
    else if(version==3)  
        {  
            return unsignedByteToShort(ref[0]) + "." +  
                unsignedByteToShort(ref[1]) + "." +  
                unsignedByteToShort(ref[2]) + "." +  
                unsignedByteToShort(ref[3]);  
        }  
  
    // In NTP Version 4 secondary servers, this is the low order 32 bits  
    // of the latest transmit timestamp of the reference source.  
    else if(version==4)  
        {
```

---

```
        return "" + ((unsignedByteToShort(ref[0]) / 256.0) +
                    (unsignedByteToShort(ref[1]) / 65536.0) +
                    (unsignedByteToShort(ref[2]) / 16777216.0) +
                    (unsignedByteToShort(ref[3]) / 4294967296.0));
    }

    return "";
}
}
```

[< Protokoll: SNTP](#) [up](#) [SNTPclient.java >](#)

---

## SNTPclient.java

**By rac**

Published: 15.01.2008 - 09:46

1.  
package hsz.ITDP.rac.uebungen.SNTPclientServer;

2.

3.  
import java.io.IOException;

4.  
import java.net.DatagramPacket;

5.  
import java.net.DatagramSocket;

6.  
import java.net.InetAddress;

7.  
import java.text.DecimalFormat;

8.  
import java.util.\*;

9.

---

```
10. public class SntpClient {
11.
12.     /**
13.      * @param args
14.      */
15.     public static void main( String [] args) {
16.
17.         String sntpHost = "lisa.hsz-t.ch";
18.
19.         int sntpPort = 123;
20.
21.         try {
22.
23.             byte[] data = generatePacket();
24.
25.             InetAddress addr = InetAddress .getByName( sntpHost );
```

---

```
sntpPort );  
  
23.     DatagramPacket send = new DatagramPacket ( data, data.length, addr,  
  
24.     DatagramSocket ds = new DatagramSocket ();  
  
25.     System .out.println("NTP request sent, waiting for response...n");  
  
26.     DatagramPacket recv = new DatagramPacket (data, data.length);  
  
27.     ds.receive(recv);  
  
28.     // Immediately record the incoming timestamp  
  
29.     //double destinationTimestamp = (System.currentTimeMillis()/1000.0) +  
2208988800.0;  
  
30.     byte[] recvdata = recv.getData();  
  
31.  
  
32.     long TransmitTimestampInt = 0;  
  
33.     long TransmitTimestampFrac = 0;
```

---

34.

35.

```
for(int i=0; i<4; i++)
```

36.

```
{
```

37.

```
short b = recvdata[40+i];
```

38.

```
if((b & 0x80)==0x80) b = (short) (128 + (b & 0x7f));
```

39.

```
else b = (short) b;
```

40.

```
TransmitTimestampInt += b * Math .pow(2, (3-i)*8);
```

41.

```
}
```

42.

```
for(int i=4; i<8; i++)
```

43.

```
{
```

44.

```
short b = recvdata[40+i];
```

45.

```
if((b & 0x80)==0x80) b = (short) (128 + (b & 0x7f));
```

46.

---

```
        else b = (short) b;

47.        TransmitTimestampFrac += b * Math .pow(2, (3-i)*8);

48.    }

49.    Date TS = getTimeDate(TransmitTimestampInt,
TransmitTimestampFrac);

50.    System .out.println("TransmitTimestamp: "+TS);

51.    ds.close();

52. }

53. catch ( IOException ioe) {

54.    System .out.println( ioe );

55. }

56. }

57.

58.
```

---

```
static private byte[] generatePacket()
```

```
59.
```

```
{
```

```
60.
```

```
//Define packet fields
```

```
61.
```

```
byte LI = 0;
```

```
62.
```

```
byte VN = 4;
```

```
63.
```

```
byte Mode = 3;
```

```
64.
```

```
short Stratum = 0;
```

```
65.
```

```
byte Poll = 0;
```

```
66.
```

```
byte Precision = 0;
```

```
67.
```

```
double RootDelay = 0;
```

```
68.
```

```
double RootDispersion = 0;
```

```
69.
```

```
byte[] ReferenceIdentifier = {0, 0, 0, 0};
```

```
70.
```

```
double ReferenceTimestamp = 0;
```

---

71.           double OriginateTimestamp = 0;

72.           double ReceiveTimestamp = 0;

73.           double TransmitTimestamp = ( [System](#) .currentTimeMillis()/1000.0) + 2208988800.0;

74.           byte[] p = new byte[48];

75.           p[0] = (byte) (LI << 6 | VN << 3 | Mode);

76.           p[1] = (byte) Stratum;

77.           p[2] = (byte) Poll;

78.           p[3] = (byte) Precision;

79.           int l = (int) (RootDelay \* 65536.0);

80.           p[4] = (byte) ((l >> 24) & 0xFF);

81.           p[5] = (byte) ((l >> 16) & 0xFF);

82.           p[6] = (byte) ((l >> 8) & 0xFF);

83.

---

```
p[7] = (byte) (l & 0xFF);

84.    long ul = (long) (RootDispersion * 65536.0);

85.    p[8] = (byte) ((ul >> 24) & 0xFF);

86.    p[9] = (byte) ((ul >> 16) & 0xFF);

87.    p[10] = (byte) ((ul >> 8) & 0xFF);

88.    p[11] = (byte) (ul & 0xFF);

89.    p[12] = ReferencelIdentifier[0];

90.    p[13] = ReferencelIdentifier[1];

91.    p[14] = ReferencelIdentifier[2];

92.    p[15] = ReferencelIdentifier[3];

93.    for(int i=0; i<8; i++)

94.    {

95.        double base = Math.pow(2, (3-i)*8);
```

---

```
96.         p[16+i] = (byte) (ReferenceTimestamp / base);

97.         short b = p[16+i];

98.         if((b & 0x80)==0x80) b = (short) (128 + (b & 0x7f));

99.         else b = (short) b;

100.        ReferenceTimestamp = ReferenceTimestamp - (double) (b * base);

101.

102.    }

103.    p[7] = (byte) ( Math .random()*255.0);

104.    for(int i=0; i<8; i++)

105.    {

106.        double base = Math .pow(2, (3-i)*8);

107.        p[24+i] = (byte) (OriginTimestamp / base);

108.
```

---

```
        short b = p[24+i];

109.        if((b & 0x80)==0x80) b = (short) (128 + (b & 0x7f));

110.        else b = (short) b;

111.        OriginateTimestamp = OriginateTimestamp - (double) (b * base);

112.

113.    }

114.    p[7] = (byte) ( Math .random()*255.0);

115.    for(int i=0; i<8; i++)

116.    {

117.        double base = Math .pow(2, (3-i)*8);

118.        p[32+i] = (byte) (ReceiveTimestamp / base);

119.        short b = p[32+i];

120.        if((b & 0x80)==0x80) b = (short) (128 + (b & 0x7f));
```

---

```
121.         else b = (short) b;

122.         ReceiveTimestamp = ReceiveTimestamp - (double) (b * base);

123.

124.     }

125.     p[7] = (byte) ( Math .random()*255.0);

126.     for(int i=0; i<8; i++)

127.     {

128.         double base = Math .pow(2, (3-i)*8);

129.         p[40+i] = (byte) (TransmitTimestamp / base);

130.         short b = p[40+i];

131.         if((b & 0x80)==0x80) b = (short) (128 + (b & 0x7f));

132.         else b = (short) b;

133.
```

---

```
TransmitTimestamp = TransmitTimestamp - (double) (b * base);
```

```
134.
```

```
135.
```

```
    }
```

```
136.
```

```
    p[7] = (byte) ( Math .random()*255.0);
```

```
137.
```

```
    return p;
```

```
138.
```

```
    }
```

```
139.
```

```
    static private Date getTimeDate(long integerPart, long fractionalPart)
```

```
140.
```

```
    {
```

```
141.
```

```
        TimeZone UTC = new SimpleTimeZone (0,"UTC");
```

```
142.
```

```
        Calendar c = new GregorianCalendar (1900, Calendar .JANUARY,1,0,0,0);
```

```
143.
```

```
        c.setTimeZone(UTC); //set time zone to 0-longitudinal (London Greenwich) because time from  
        SNTP server is GMT (=UTC when no summertime)
```

```
144.
```

```
        Date startOfCentury = c.getTime();
```

```
145.
```

---

```
integerPart = integerPart * 1000; //get milliseconds

146. System.out.println("integerPart = " + Long.toHexString(integerPart));

147. System.out.println("fractionalPart = " + Long.toHexString(fractionalPart));

148. float fPart = fractionalPart; //first assign long value to float

149. fPart = fPart / 4294967296L; //shift value left so only fractional part remains; division by
2^32

150. //now fPart is a number < 1

151. fPart = fPart * 1000; //get milliseconds (drop higher precision)

152. System.out.println("fPart = " + Float.toString(fPart) + "[ms]");

153. //now 0 <= fPart <= 999 (number of milliseconds)

154. long msSinceStartOfCentury = integerPart + (long)fPart; //add integer and fractional part

155. Date now = new Date (msSinceStartOfCentury+startOfCentury.getTime());

156. return now;

157.
```

---

```
}
```

```
158.
```

```
159.
```

```
}
```

[< DemoNtpMessage.java up](#)