Team Rusty Buckets

Cameron Moberg, Evan Gauer, Eli Charleville

Dr. Chetan Jaiswal

CS 470: Networking

7 March 2018

Project 1 Write-up

1 Project Report

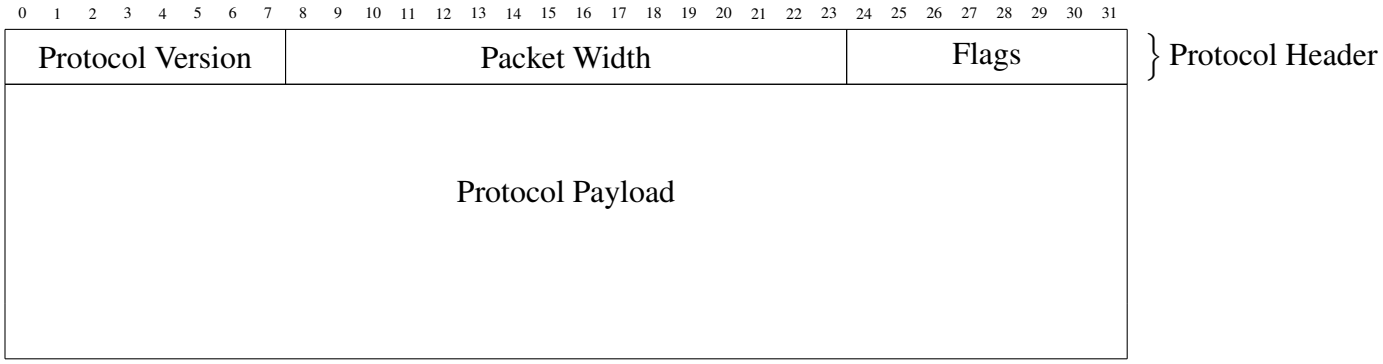
In Project 1 we designed a protocol that will allow an implementation to:

- To detect node failure periodically
- To inform the other nodes in the network about failure (peering option)
- To be able to detect when the failed node comes back to life
- To inform other nodes about the availability of a new node

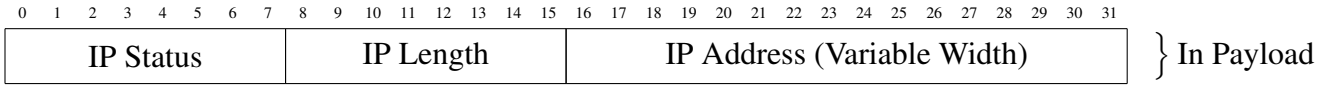
In the end we designed a protocol that does these things, and the implementation was designed for a client-server architecture alongside a peer-to-peer architecture.

2 Protocol Design

Upon designing the protocol we were faced with some constraints as to what it should be able to let us do. We used the UDP protocol as a top layer to transport our protocol, so we omitted some things such as timestamp and checksum since they were taken care of in UDP. In the end our protocol packet looked something like this:



Where the payload is split up with multiple of these:



The IP Address is going to be either 16 bytes (IPv6) or 4 bytes (IPv4).

The protocol version has 8 bits, packet width has 16 bits to represent the header and payload size, and 8 bits for flags. Currently only one is used: heartbeat. Inside of the payload there are multiple IP addresses formatted with 8 bits for IP status. In our implementation IP status had several numbers to represent different statuses such as IP revived, or new IP found. The next 8 bits tell how long the IP that follows is in bytes. Lastly we have the IP address in byte format of previously specified length. This allows support for multiple IP versions.

We believe this is a good HAC protocol design because it allows for an arbitrary amount of IPs to be sent ontop of the UDP protocol. We also have room for improvement in this protocol, with 7 unused flags in the header, and $2^8 - 1$ more version numbers we can use. We also believe this is a good design since it is very simple, and is left up to the implementations to convert the packet into usable data. With this simplicity we save bytes and therefore congestion in the network. We also have 8 bits for IP status, which means we can have 2^8 status codes for ultimate fine-grained control of IP address notifications.

As far as implementation goes, it takes full advantage of this protocol. It will notify the user when a node is new, revived, assumed failed, offline, and with the peering option will alert all other nodes.

3 Team Member Contributions

- Cameron Moberg
  - Class Design
  - Writeup contributions
  - Testing
- Evan Gauer
  - Debugging
  - Protocol Design
  - Writeup contributions
- Eli Charleville
  - Class Design
  - Writeup contributions
  - Testing

```java
public enum PacketStatus
{
   OFFLINE(0), ONLINE(1), NEW(2), FAIL(3), REVIVE(4);

   private int statusCode;

   PacketStatus(int statusCode)
   {
      this.statusCode = statusCode;
   }

   public int getStatusCode()
   {
      return this.statusCode;
   }
}
```

```java
/**
 * This class contains the implementation at the lowest level of the
 * packet.
 */
public class AvailabilityPacket
{
   private static final int HEADER_SIZE = 4;
   private static final int VERSION = 1;

   private Map<InetAddress, PacketStatus> ips;
   private byte[] payload;
   private boolean heartbeat;
   private byte version;

   /**
    * Populates payload with a single InetAddress, useful for sending
    * updates in peering mode.
    *
    * @param addr      InetAddress to encode
    * @param status    PacketState of InetAddress
    * @param heartbeat Whether or not this also should heartbeat.
    */
   public AvailabilityPacket(InetAddress addr, PacketStatus status, boolean heartbeat)
   {
      this.heartbeat = heartbeat;
      this.version = VERSION;
      this.payload = encodeSingle(addr, status);
   }

   /**
    * Populates payload with InetAddress and the corresponding status of that
    * InetAddress.
    *
    * @param ips       Map of InetAddress and PacketStatuses to encode
    * @param heartbeat Whether or not this also should heartbeat.
    */
   public AvailabilityPacket(Map<InetAddress, PacketStatus> ips, boolean heartbeat)
   {
      this.heartbeat = heartbeat;
      this.version = VERSION;
      this.ips = ips;
      this.payload = encodeLists();
   }

   /**
    * Sets the payload, does not automatically decode anything; decode must be
    * called.
    *
    * @param payload Payload received using AvailabilityPacket protocol.
    */
   public AvailabilityPacket(byte[] payload)
   {
      this.payload = payload;
      this.ips = new HashMap<>();
   }

   /**
    * Given the current instance payload, decodes it into InetAddresses and
    * statuses.
    *
    * @return returns packet with decoded ips.
    */
   public AvailabilityPacket decode()
   {
```

```
65          this.version = payload[0];
66
67          int packetLength = (payload[1] << 8) + payload[2];
68          byte flags = payload[3];
69          this.heartbeat = ((flags >> 7) & 1) == 1;
70          int counter = HEADER_SIZE;
71
72          while (counter < packetLength)
73          {
74              int ipSize = payload[counter++];
75              int statusCode = payload[counter++];
76              byte[] ipAddr = Arrays.copyOfRange(payload, counter, counter + ipSize);
77
78              try
79              {
80                  InetAddress address = InetAddress.getByAddress(ipAddr);
81                  PacketStatus status = PacketStatus.values()[statusCode];
82                  ips.put(address, status);
83              } catch (UnknownHostException e)
84              {
85                  e.printStackTrace();
86              }
87              counter += ipSize;
88          }
89          return this;
90      }
91
92      /**
93       * Encodes a single InetAddress and status.
94       *
95       * @return InetAddress/Status encoded as our protocol.
96       */
97      public byte[] encodeSingle(InetAddress inetAddress, PacketStatus status)
98      {
99          byte[] bytes = new byte[1024];
100         int counter = HEADER_SIZE;
101
102         counter = copyInetAddrToPayload(bytes, counter, inetAddress, status);
103
104         bytes[0] = this.version;
105         //Give length field 2 bytes.
106         bytes[1] = (byte) ((counter >> 8) & 0xFF);
107         bytes[2] = (byte) (counter & 0xFF);
108         // If heartbeat set 7th bit to 1, else 0
109         bytes[3] = (byte) (heartbeat ? (bytes[2] | 1 << 7) : bytes[2] & ~(1 << 7));
110         return bytes;
111     }
112
113     /**
114      * Encodes current ips that packet was initialized with.
115      *
116      * @return returns payload encoded as byte[].
117      */
118     public byte[] encodeLists()
119     {
120         byte[] bytes = new byte[1024];
121         //leave 2 bits for width
122         int counter = HEADER_SIZE;
123
124         if (ips != null)
125         {
126             for (Map.Entry<InetAddress, PacketStatus> entry : ips.entrySet())
127             {
128                 InetAddress address = entry.getKey();
129                 PacketStatus status = entry.getValue();
```

```
130             counter = copyInetAddrToPayload(bytes, counter, address, status);
131          }
132       }
133
134       bytes[0] = this.version;
135       //Give length field 2 bytes.
136       bytes[1] = (byte) ((counter >> 8) & 0xFF);
137       bytes[2] = (byte) (counter & 0xFF);
138       // If heartbeat set 7th bit to 1, else 0
139       bytes[3] = (byte) (heartbeat ? (bytes[2] | 1 << 7) : bytes[2] & ~(1 << 7));
140
141       return bytes;
142    }
143
144    /**
145     * Adds InetAddress and Status to payload.
146     *
147     * @param bytes   Array to be copied to.
148     * @param counter Starting byte index of payload byte[] array.
149     * @param address InetAddress to be copied.
150     * @param status  Status of address to be copied.
151     * @return New counter number in payload.
152     */
153    private int copyInetAddrToPayload(byte[] bytes, int counter,
154                                      InetAddress address, PacketStatus status)
155    {
156       // 4 if Ipv4 (4 bytes), 16 if ipv6 (16 bytes)
157       bytes[counter++] = (byte) ((address instanceof Inet6Address) ? 16 : 4);
158       // Set bit to the packet status code.
159       bytes[counter++] = (byte) status.getStatusCode();
160       // Copy IP Address over
161       byte[] ip = address.getAddress();
162       System.arraycopy(ip, 0, bytes, counter, ip.length);
163       counter += ip.length;
164
165       return counter;
166    }
167
168
169    public boolean isHeartbeat()
170    {
171       return this.heartbeat;
172    }
173
174    public Map<InetAddress, PacketStatus> getIps()
175    {
176       return ips;
177    }
178
179    public byte[] getPayload()
180    {
181       return this.payload;
182    }
183 }
```

```java
public class Client
{
   private static int PORT_NUM = 9998;

   private DatagramSocket socket;
   private String masterIp = readMasterIp();
   private Random random = new Random();

   private List<InetAddress> onlineIpList = new ArrayList<>();
   private List<InetAddress> offlineIpList = new ArrayList<>();
   private List<InetAddress> localIpList;

   /**
    * Receives packet, and hands off to another method
    */
   public void listenPacket()
   {
      byte[] buffer = new byte[1024];
      DatagramPacket dP = new DatagramPacket(buffer, buffer.length);
      try
      {
         socket.receive(dP);
      } catch (SocketTimeoutException ignored)
      {
         return;
      } catch (IOException e)
      {
         e.printStackTrace();
      }
      handlePayload(dP);
   }

   public void handleStatus(InetAddress address, PacketStatus status)
   {
      switch (status)
      {
         case NEW:
            if (!onlineIpList.contains(address))
            {
               System.out.println("New Node Available");
               onlineIpList.add(address);
            }
            break;
         case REVIVE:
            if (!onlineIpList.contains(address))
            {
               System.out.println("Node revived " + address.getHostAddress());
               onlineIpList.add(address);
               offlineIpList.remove(address);
            }
            break;
         case OFFLINE:
         case FAIL:
            if (!offlineIpList.contains(address))
            {
               System.out.println("Node Offline/Failed " + address.getHostAddress());
               offlineIpList.add(address);
               onlineIpList.remove(address);
            }
            break;
         case ONLINE:
            if (!onlineIpList.contains(address))
            {
               if (offlineIpList.contains(address))
```

```
65                  {
66                     System.out.println("New Node Available - Alerting (Revived)");
67                     onlineIpList.add(address);
68                     offlineIpList.remove(address);
69                  } else
70                  {
71                     System.out.println("New Node Available - Alerting (New)");
72                     onlineIpList.add(address);
73                  }
74               }
75               break;
76         }
77      }
78
79      /**
80       * Handles the Datagram packet and using our protocol determines status of all ips.
81       *
82       * @param packet DataGram packet received with data in Availability Packet form.
83       */
84      public void handlePayload(DatagramPacket packet)
85      {
86         AvailabilityPacket decoded = new AvailabilityPacket(packet.getData()).decode();
87
88         if (decoded.isHeartbeat())
89         {
90            //Handle sender of packet
91            handleStatus(packet.getAddress(), PacketStatus.ONLINE);
92         }
93
94         for (Map.Entry<InetAddress, PacketStatus> entry : decoded.getIps().entrySet())
95         {
96            InetAddress address = entry.getKey();
97            PacketStatus status = entry.getValue();
98
99            if (!localIpList.contains(address))
100           {
101              handleStatus(address, status);
102           }
103        }
104     }
105
106     /**
107      * Sends datagram packet with a byte[0] as payload.
108      */
109     private void sendHeartbeat()
110     {
111        System.out.println("Sending hb");
112
113        //In file the ip is formatted like xxx.xxx.xxx.xxx:8888
114        String ip = masterIp.split("\\:")[0];
115        int port = Integer.valueOf(masterIp.split("\\:")[1]);
116
117        try
118        {
119           InetAddress inetAddress = InetAddress.getByName(ip);
120           // Set flags for heartbeat.
121           byte heartbeat = (byte) (1 << 7);
122           DatagramPacket packet = new DatagramPacket(new byte[]{0, 3, heartbeat},
123                 2, inetAddress, port);
124           socket.send(packet);
125        } catch (IOException e)
126        {
127           e.printStackTrace();
128        }
129     }
```

```
130
131     /**
132      * outputs all ips
133      */
134     private void outputIps()
135     {
136        System.out.println("----- Online -----");
137        for (InetAddress ip : onlineIpList)
138        {
139           System.out.println(ip.getHostAddress());
140        }
141        System.out.println("----- Offline -----");
142        for (InetAddress ip : offlineIpList)
143        {
144           System.out.println(ip.getHostAddress());
145        }
146        System.out.println("-----------------");
147     }
148
149     /**
150      * Reads in IPs from file.
151      *
152      * @return Array of ips in string format xxx.xxx.xxx.xxx:xxxx
153      */
154     private String readMasterIp()
155     {
156        File file = new File("masterIp");
157        String masterIp = null;
158        try
159        {
160           BufferedReader br = new BufferedReader(new FileReader(file));
161           masterIp = br.readLine();
162
163        } catch (IOException e)
164        {
165           e.printStackTrace();
166        }
167        return masterIp;
168     }
169
170     /**
171      * Gets all local interface InetAddresses, to be used to
172      * determine not to report local machine to ourselves.
173      */
174     private List<InetAddress> populateLocalAddresses() throws SocketException
175     {
176        List<InetAddress> listOfAddr = new ArrayList<>();
177        Enumeration e = NetworkInterface.getNetworkInterfaces();
178
179        while (e.hasMoreElements())
180        {
181           NetworkInterface netInterface = (NetworkInterface) e.nextElement();
182           Enumeration ee = netInterface.getInetAddresses();
183           while (ee.hasMoreElements())
184           {
185              listOfAddr.add((InetAddress) ee.nextElement());
186           }
187        }
188        return listOfAddr;
189     }
190
191     /**
192      * Begins both a listener and sender thread that
193      * listens and sends on the DatagramSocket.
194      */
```

```
195     public void begin()
196     {
197        try
198        {
199           this.socket = new DatagramSocket(PORT_NUM);
200           this.localIpList = populateLocalAddresses();
201        } catch (SocketException e)
202        {
203           e.printStackTrace();
204        }
205
206        // Start listener thread
207        new Thread(() ->
208        {
209           while (true)
210           {
211              listenPacket();
212              outputIps();
213           }
214        }).start();
215
216        // Start sender thread
217        new Thread(() ->
218        {
219           Instant nextBeat = Instant.now();
220
221           while (true)
222           {
223              if (Duration.between(Instant.now(), nextBeat).isNegative())
224              {
225                 int randSec = random.nextInt(30) + 1;
226                 nextBeat = Instant.now().plusSeconds(randSec);
227                 sendHeartbeat();
228              }
229           }
230        }).start();
231     }
232  }
```

```
1  public class Server
2  {
3     private static int PORT_NUM = 9999;
4     private static int NODE_TIMEOUT = 31;
5
6     private DatagramSocket socket = null;
7     private Random random = new Random();
8
9     private Map<InetAddress, Instant> onlineIpMap = new ConcurrentHashMap<>();
10    private List<InetAddress> offlineIpList = new ArrayList<>();
11
12    private String[] ips = readIps();
13
14    /**
15     * Receives packet, and hands off to another method
16     */
17    public void listenPacket()
18    {
19       byte[] buffer = new byte[1024];
20       DatagramPacket dP = new DatagramPacket(buffer, buffer.length);
21       try
22       {
23          socket.receive(dP);
24       } catch (SocketTimeoutException ignored)
25       {
26          return;
27       } catch (IOException e)
28       {
29          e.printStackTrace();
30       }
31
32       onlineIpMap.put(dP.getAddress(), Instant.now());
33       offlineIpList.remove(dP.getAddress());
34    }
35
36    /**
37     * Sends datagram packet with AvailabilityPacket as payload.
38     *
39     * @param proto Packet wanting to send.
40     */
41    public void sendPacket(AvailabilityPacket proto)
42    {
43       byte[] encodedPacket = proto.getPayload();
44
45       if (encodedPacket == null)
46       {
47          return;
48       }
49
50       for (String socketAddr : ips)
51       {
52          //In file the ips are formatted like xxx.xxx.xxx.xxx:8888
53          String ip = socketAddr.split("\\:")[0];
54          int port = Integer.valueOf(socketAddr.split("\\:")[1]);
55
56          try
57          {
58             InetAddress inetAddress = InetAddress.getByName(ip);
59             DatagramPacket packet = new DatagramPacket(encodedPacket, encodedPacket.length,
60                   inetAddress, port);
61             socket.send(packet);
62          } catch (IOException e)
63          {
64             e.printStackTrace();
```

```
65          }
66       }
67    }
68
69    /**
70     * Combines onlineIp and offlineIp into a map that Availability packet can handle.
71     * Could be implemented in the Packet class.
72     *
73     * @return Map containing online/offline ip addresses.
74     */
75    private Map<InetAddress, PacketStatus> combineIpsIntoMap()
76    {
77       HashMap<InetAddress, PacketStatus> map = new HashMap<>();
78       for (InetAddress address : onlineIpMap.keySet())
79       {
80          map.put(address, PacketStatus.ONLINE);
81       }
82       for (InetAddress address : offlineIpList)
83       {
84          map.put(address, PacketStatus.OFFLINE);
85       }
86       return map;
87    }
88
89    /**
90     * Removes nodes that've been offline for more than NODE_OFFLINE time
91     * from onlineIp list. Also sends packet to all in ip file that a node has gone offline.
92     */
93    private void pruneNodes()
94    {
95       for (Map.Entry<InetAddress, Instant> ip : onlineIpMap.entrySet())
96       {
97          Instant ipLastKnown = ip.getValue();
98          if (Instant.now().isAfter(ipLastKnown.plusSeconds(NODE_TIMEOUT)))
99          {
100            System.out.println("Node Assumed Offline - Alerting (Failure): "
101                                  + ip.getKey().getHostAddress());
102            sendPacket(new AvailabilityPacket(ip.getKey(), PacketStatus.FAIL, false));
103            offlineIpList.add(ip.getKey());
104            onlineIpMap.remove(ip.getKey());
105         }
106      }
107   }
108
109   /**
110    * Reads in IPs from file.
111    *
112    * @return Array of ips in string format xxx.xxx.xxx.xxx:xxxx
113    */
114   private String[] readIps()
115   {
116      File file = new File("ips");
117      ArrayList<String> ipList = new ArrayList<>();
118
119      try
120      {
121         BufferedReader br = new BufferedReader(new FileReader(file));
122         String str;
123         while ((str = br.readLine()) != null)
124         {
125            ipList.add(str);
126         }
127
128      } catch (IOException e)
129      {
```

```
130            e.printStackTrace();
131         }
132
133         return ipList.toArray((new String[0]));
134      }
135
136      /**
137       * outputs all ips
138       */
139      private void outputIps()
140      {
141         System.out.println("----- Online -----");
142         for (InetAddress ip : onlineIpMap.keySet())
143         {
144            System.out.println(ip.getHostAddress());
145         }
146         System.out.println("----- Offline -----");
147         for (InetAddress ip : offlineIpList)
148         {
149            System.out.println(ip.getHostAddress());
150         }
151         System.out.println("-----------------");
152      }
153
154      /**
155       * Start up the 2 threads, one that receives and outputs, and one that
156       * sends heartbeats out.
157       */
158      public void begin()
159      {
160         try
161         {
162            this.socket = new DatagramSocket(PORT_NUM);
163         } catch (SocketException e)
164         {
165            e.printStackTrace();
166         }
167
168         new Thread(() ->
169         {
170            while (true)
171            {
172               listenPacket();
173               outputIps();
174            }
175         }).start();
176
177         new Thread(() ->
178         {
179            Instant nextBeat = Instant.now();
180
181            while (true)
182            {
183               if (Duration.between(Instant.now(), nextBeat).isNegative())
184               {
185                  int randSec = random.nextInt(30) + 1;
186                  nextBeat = Instant.now().plusSeconds(randSec);
187                  sendPacket(new AvailabilityPacket(combineIpsIntoMap(), true));
188               } else
189               {
190                  pruneNodes();
191               }
192            }
193
194         }).start();
```

```
195       }
196   }
```

```java
public class P2PNode
{
   private static int NODE_OFFLINE = 31;
   private static int PORT_NUM = 9998;

   private Map<InetAddress, Instant> onlineIpMap = new ConcurrentHashMap<>();
   private List<InetAddress> offlineIpList = new ArrayList<>();
   private List<InetAddress> localIpList;

   private String[] ips = readIps();
   private DatagramSocket socket;
   private Random random = new Random();

   /**
    * Sends datagram packet with AvailabilityPacket as payload.
    *
    * @param proto
    */
   public void sendPacket(AvailabilityPacket proto)
   {
      byte[] encodedPacket = proto.getPayload();

      if (encodedPacket == null)
      {
         return;
      }

      for (String socketAddr : ips)
      {
         //In file the ips are formatted like xxx.xxx.xxx.xxx:8888
         String ip = socketAddr.split("\\:")[0];
         int port = Integer.valueOf(socketAddr.split("\\:")[1]);

         try
         {
            InetAddress inetAddress = InetAddress.getByName(ip);
            DatagramPacket packet = new DatagramPacket(encodedPacket, encodedPacket.length,
                  inetAddress, port);
            socket.send(packet);
         } catch (IOException e)
         {
            e.printStackTrace();
         }
      }
   }

   /**
    * Receives packet, and hands off to another method
    */
   public void listenPacket()
   {
      byte[] buffer = new byte[1024];
      DatagramPacket dP = new DatagramPacket(buffer, buffer.length);
      try
      {
         socket.receive(dP);
      } catch (SocketTimeoutException ignored)
      {
         return;
      } catch (IOException e)
      {
         e.printStackTrace();
      }
      handlePayload(dP);
```

```
65      }
66
67      public void handleStatus(InetAddress address, PacketStatus status)
68      {
69          switch (status)
70          {
71              case NEW:
72                  if (!onlineIpMap.containsKey(address))
73                  {
74                      System.out.println("New Node Available");
75                      onlineIpMap.put(address, Instant.now());
76                  }
77                  break;
78              case REVIVE:
79                  if (!onlineIpMap.containsKey(address))
80                  {
81                      System.out.println("Node revived " + address.getHostAddress());
82                      onlineIpMap.put(address, Instant.now());
83                      offlineIpList.remove(address);
84                  }
85                  break;
86              case OFFLINE:
87              case FAIL:
88                  if (!offlineIpList.contains(address))
89                  {
90                      System.out.println("Node Offline/Failed " + address.getHostAddress());
91                      offlineIpList.add(address);
92                      onlineIpMap.remove(address);
93                  }
94                  break;
95              case ONLINE:
96                  if (!onlineIpMap.containsKey(address))
97                  {
98                      if (offlineIpList.contains(address))
99                      {
100                         System.out.println("New Node Available - Alerting (Revived)");
101                         onlineIpMap.put(address, Instant.now());
102                         offlineIpList.remove(address);
103                         sendPacket(new AvailabilityPacket(address, PacketStatus.REVIVE, false));
104                     } else
105                     {
106                         System.out.println("New Node Available - Alerting (New)");
107                         onlineIpMap.put(address, Instant.now());
108                         sendPacket(new AvailabilityPacket(address, PacketStatus.NEW, false));
109                     }
110                 }
111                 break;
112         }
113     }
114
115     /**
116      * Handles the Datagram packet and using our protocol determines status of all ips.
117      *
118      * @param packet DataGram packet received with data in Availability Packet form.
119      */
120     public void handlePayload(DatagramPacket packet)
121     {
122         //Handle sender of packet
123         handleStatus(packet.getAddress(), PacketStatus.ONLINE);
124
125         AvailabilityPacket decoded = new AvailabilityPacket(packet.getData()).decode();
126
127         for (Map.Entry<InetAddress, PacketStatus> entry : decoded.getIps().entrySet())
128         {
129             InetAddress address = entry.getKey();
```

```
130          PacketStatus status = entry.getValue();
131
132          if (!localIpList.contains(address))
133          {
134             handleStatus(address, status);
135          }
136       }
137    }
138
139    /**
140     * Removes nodes that've been offline for more than NODE_OFFLINE time
141     * from onlineIp list. Also sends packet to all in ip file that a node
142     * has gone offline.
143     */
144    private void pruneNodes()
145    {
146       for (Map.Entry<InetAddress, Instant> ip : onlineIpMap.entrySet())
147       {
148          Instant ipLastKnown = ip.getValue();
149          if (Instant.now().isAfter(ipLastKnown.plusSeconds(NODE_OFFLINE)))
150          {
151             System.out.println("Node Assumed Offline - Alerting (Failure): " + ip.getKey().getHos
152             sendPacket(new AvailabilityPacket(ip.getKey(), PacketStatus.FAIL, false));
153             offlineIpList.add(ip.getKey());
154             onlineIpMap.remove(ip.getKey());
155          }
156       }
157    }
158
159    /**
160     * outputs all ips
161     */
162    private void outputIps()
163    {
164       System.out.println("----- Online -----");
165       for (InetAddress ip : onlineIpMap.keySet())
166       {
167          System.out.println(ip.getHostAddress());
168       }
169       System.out.println("----- Offline -----");
170       for (InetAddress ip : offlineIpList)
171       {
172          System.out.println(ip.getHostAddress());
173       }
174       System.out.println("-----------------");
175    }
176
177    /**
178     * Reads in IPs from file.
179     *
180     * @return Array of ips in string format xxx.xxx.xxx.xxx:xxxx
181     */
182    private String[] readIps()
183    {
184       File file = new File("ips");
185       ArrayList<String> ipList = new ArrayList<>();
186
187       try
188       {
189          BufferedReader br = new BufferedReader(new FileReader(file));
190          String str;
191          while ((str = br.readLine()) != null)
192          {
193             ipList.add(str);
194          }
```

```
195
196         } catch (IOException e)
197         {
198             e.printStackTrace();
199         }
200
201         return ipList.toArray((new String[0]));
202     }
203
204     /**
205      * Combines onlineIp and offlineIp into a map that Availability packet can
206      * handle. Could be implemented in the Packet class.
207      *
208      * @return Map containing online/offline ip addresses.
209      */
210     private Map<InetAddress, PacketStatus> getAllPackets()
211     {
212         HashMap<InetAddress, PacketStatus> map = new HashMap<>();
213         for (InetAddress address : onlineIpMap.keySet())
214         {
215             map.put(address, PacketStatus.ONLINE);
216         }
217         for (InetAddress address : offlineIpList)
218         {
219             map.put(address, PacketStatus.OFFLINE);
220         }
221         return map;
222     }
223
224     /**
225      * Gets all local interface InetAddresses to not report local machine status.
226      */
227     private List<InetAddress> populateLocalAddresses() throws SocketException
228     {
229         List<InetAddress> listOfAddr = new ArrayList<>();
230         Enumeration e = NetworkInterface.getNetworkInterfaces();
231
232         while (e.hasMoreElements())
233         {
234             NetworkInterface netInterface = (NetworkInterface) e.nextElement();
235             Enumeration ee = netInterface.getInetAddresses();
236             while (ee.hasMoreElements())
237             {
238                 listOfAddr.add((InetAddress) ee.nextElement());
239             }
240         }
241         return listOfAddr;
242     }
243
244     /**
245      * Start up the 2 threads, one that receives and outputs, and one that
246      * sends heartbeats out.
247      */
248     public void begin()
249     {
250         try
251         {
252             this.socket = new DatagramSocket(PORT_NUM);
253             this.localIpList = populateLocalAddresses();
254         } catch (SocketException e)
255         {
256             e.printStackTrace();
257         }
258
259         new Thread(() ->
```

```
260         {
261             while (true)
262             {
263                 listenPacket();
264                 outputIps();
265             }
266         }).start();
267 
268         new Thread(() ->
269         {
270             Instant nextBeat = Instant.now();
271 
272             while (true)
273             {
274                 if (Duration.between(Instant.now(), nextBeat).isNegative())
275                 {
276                     int randSec = random.nextInt(30) + 1;
277                     nextBeat = Instant.now().plusSeconds(randSec);
278                     sendPacket(new AvailabilityPacket(getAllPackets(), true));
279                 } else
280                 {
281                     pruneNodes();
282                 }
283             }
284 
285         }).start();
286     }
287 }
```