Java Nio Heartbeat Example

examples.javacodegeeks.com/core-java/nio/java-nio-heartbeat-example/

This article is a tutorial on implementing a simple Java NIO Heartbeat. This example will take the form of “n” number of “Broadcast” mode processes which will multicast data via UDP to “n” number of “Subscribe” processes that have expressed interest in receiving said traffic.

1. Introduction

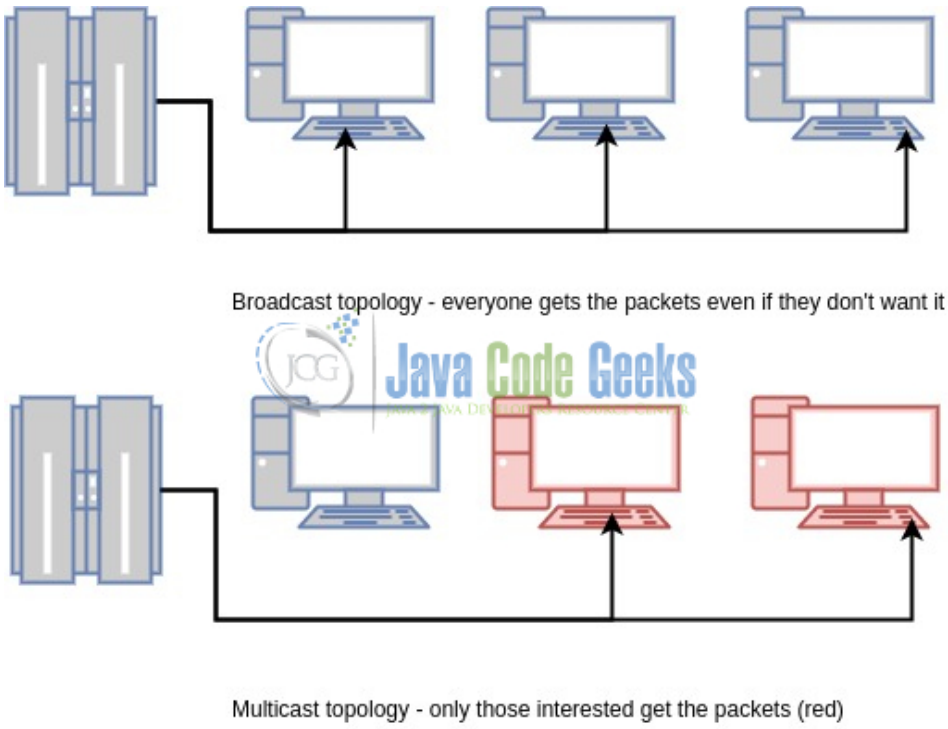
This article builds on three earlier articles on the subject of Java NIO, namely “Java Nio Tutorial for Beginners”, “Java Nio Asynchronous Channels Tutorial” and “Java Nio EchoServer“. Before getting stuck into the “meat” of our example, it is best to get some background into the topic itself. According to Wikipedia a “heartbeat” in computer systems is a periodic signal generated by hardware or software to indicate normal operation or to synchronize parts of a system. So true to the name it is indeed a measure of life of individual components in a distributed computer system and one can deduce then by it’s absence, presence and frequency, the state of the system in which it occurs.

In the same breath, when one talks about “heartbeats” in computer systems the term “UDP” often comes up and with good reason. It is the protocol of choice when implementing “hearbeat” type solutions, weather it be cluster membership negotiations or life signing (heartbeats). The low latency of this “connection-less” protocol also plays to the nature of “heartbeating” in distributed systems.

Important to note that unlike TCP, UDP makes no guarantee on delivery of packets, the low latency of UDP stems from this not having to guarantee delivery via the typical SYN ACK (3 way handshake etc).

We go one step further in this example and we multicast the traffic out to interested parties. Now why would we do this and what other choices are there? Typically the following choices would present themselves:

- Unicast: From one machine to another. One-to-One
- Broadcast: From one machine to all possible machines. One-to-All (within the broadcast domain – ie: behind a router or in a private network)
- Multicast: From one machine to multiple machines that have stated interest in receiving said traffic. This can traverse the broadcast domain and extend past a router.



Topology of Broadcast vs Multicast

2. Technologies used

The example code in this article was built and run using:

- Java 1.8.101 (1.8.x will do fine)
- Maven 3.3.9 (3.3.x will do fine)
- Spring source tool suite 4.6.3 (Any Java IDE would work)
- Ubuntu 16.04 (Windows, Mac or Linux will do fine)

3. Overview

The abstractions of use to us when wanting to effect UDP in Java Nio would be the DatagramChannel, which also happens to be a SelectableChannel priming it for use by a Selector in a very Thread efficient manner. It also happens to implement MulticastChannel which supports Internet Protocol (IP) multicasting.

3.1 DatagramChannel

A DatagramChannel is opened by one of the static `open(...)` methods of the class itself. One of the `open(...)` methods are of particular interest to us and that is:

DatagramChannel open for multicast

```
1    public static DatagramChannel open(ProtocolFamily family) throws IOException
```

The ProtocolFamily is required when attempting to multicast with this Channel and should correspond to the IP type of the multicast group that this Channel will join. eg: IPV4 `StandardProtocolFamily.INET` A DatagramChannel need not be connected to use the `send(...)` and `receive(...)` methods of this class, conversely so the `read(...)` and `write(...)` methods do.

3.2 MulticastChannel

This Channel supports (IP) multicasting. Of particular interest to us is this part of it's API:

DatagramChannel configuration

```
1  ...
2  channel.setOption(StandardSocketOptions.IP_MULTICAST_IF, NetworkInterface);
3  channel.join(InetAddress,this.multicastNetworkInterface);
4  ...
```

line 2: the NetworkInterface is the interface through which we will send / receive UDP multicast traffic
line 3: we ensure we join the multicast group (express interest in receiving traffic to this group) by way of passing a InetAddress (the multicast IP) and a NetworkInterface (the interface through which we will receive said multicast traffic). Multicast IP ranges range from 224.0.0.0 to 239.255.255.255 typically.

A MulticastChannel can join "n" number of multicast groups and can join a group on different network interfaces.

4. Multicaster

Multicaster

```
01                                  ScheduledChannelOperation
    final class Multicasterimplements {
02
03                  String
        private final id;
04                  ScheduledExecutorService
        private final scheduler;
05                  NetworkInterface
        private final networkInterface;
06      private final InetSocketAddress multicastGroup;
07
08                      String          String          String
        Multicaster(final id,       final ip,       final interfaceName,       final int port,
             poolSize)
    final int {
09          if
    (StringUtils.isEmpty(id) || StringUtils.isEmpty(ip) ||
    StringUtils.isEmpty(interfaceName)) {
```

```
10                                          "required id, ip and
            throw new IllegalArgumentException(interfaceName"                    );
11          }
12
13              .id =
            thisid;
14              .scheduler =
            thisExecutors.newScheduledThreadPool(poolSize);
15              .multicastGroup     InetSocketAddress(ip,
            this=                 new port);
16
17          try {
18                  .networkInterface =
                thisNetworkInterface.getByName(interfaceName);
19                (SocketException e)
            }catch {
20                                      "unable to start           ,
                throw new RuntimeException(broadcaster"               e);
21          }
22      }
23
24      @Override
25            ScheduledExecutorService getService()
        public {
26          return this.scheduler;
27      }
28
29                  CountDownLatch endLatch)
        void run(final {
```

```
30          assert !Objects.isNull(endLatch);
31
32              (DatagramChannel channel = DatagramChannel.open())
            try {
33
34              initChannel(channel);
35              doSchedule(channel);
36
37              endLatch.await();
38                (IOException | InterruptedException e)
            }catch {
39                                          "unable to run            ,
                throw new RuntimeException(broadcaster"               e);
40          }finally {
41              this.scheduler.shutdownNow();
42          }
43      }
44
45                                      DatagramChannel channel)
        private void doSchedule(final {
46          assert !Objects.isNull(channel);
47
48                                  Runnable()
            doSchedule(channel,new {
```

```
49                          run()
              public void {
50                                                  "Multicasting for    ,
                System.out.println(String.format(%s"                 Multicaster.  this
    .id));
51
52                  try {
53                      Multicaster.this.doBroadcast(channel);
54                         (IOException e)
                    }catch {
55                      e.printStackTrace();
56                  }
57              }
58          }, 0L, Constants.Schedule.PULSE_DELAY_IN_MILLISECONDS,
            TimeUnit.MILLISECONDS);
59      }
60
61                                  DatagramChannel               IOException
        private void initChannel(final channel)                throws {
62          assert !Objects.isNull(channel);
63
64          channel.bind(null);
65          channel.setOption(StandardSocketOptions.IP_MULTICAST_IF,this.networkInterface);
66      }
67
68                                  DatagramChannel               IOException
        private void doBroadcast(final channel)                throws {
```

```
69          assert !Objects.isNull(channel);
70
71                                      .multicastGroup,
            Pulse.broadcast(this.id,thischannel);
72      }
73  }
```

- line 14: we create a ScheduledExecutorService with the purposes of scheduling the multicast heartbeat pulse to the multicast group
- line 15: we create a InetSocketAddress which will be the multicast group to which we will send our heartbeats
- line 18: we create a NetworkInterface which will encapsulate the interface through which our multicast heartbeats will travel
- line 34: we initialize our DatagramChannel
- line 35: we schedule our heartbeat thread
- line 48-58: represents the schedule task that is run, this is quite simply a `send(...)` operation on the DatagramChannel to the InetSocketAddress which represents our multicast group
- line 64: allow any socket address to be bound to the socket – does not matter
- line 65: ensure we set the NetworkInterface to be used for the multicast heartbeats that are sent. We don't set the TTL for the multicast, although you could if you want.

5. Subscriber

Subscriber

```
001                                 ScheduledChannelOperation
    final class Subscriberimplements {
002
003                 String
        private final id;
004                 ScheduledExecutorService
        private final scheduler;
005                 NetworkInterface
        private final networkInterface;
006     private final InetSocketAddress hostAddress;
007                 InetAddress
        private final group;
```

```
008                     ConcurrentMap<String, Pulse>
        private final pulses;
009
010                         String          String          String
        Subscriber(final id,        final ip,       final interfaceName,       final int port,
             poolSize)
    final int {
011         if
    (StringUtils.isEmpty(id) && StringUtils.isEmpty(ip) ||
    StringUtils.isEmpty(interfaceName)) {
012                                                 "required id, ip and
                throw new IllegalArgumentException(interfaceName"                       );
013         }
014
015             .id =
            thisid;
016             .scheduler =
            thisExecutors.newScheduledThreadPool(poolSize);
017             .hostAddress
            this=             new InetSocketAddress(port);
018             .pulses
            this=         new ConcurrentHashMap<>();
019
020         try {
021                 .networkInterface =
                thisNetworkInterface.getByName(interfaceName);
022                 .group =
                thisInetAddress.getByName(ip);
023               (SocketException | UnknownHostException e)
            }catch {
024                                       "unable to start            ,
                throw new RuntimeException(broadcaster"                 e);
```

```
025          }
026      }
027
028      @Override
029            ScheduledExecutorService getService()
         public {
030          return this.scheduler;
031      }
032
033           run()
         void {
034          try (final
     DatagramChannel channel =
     DatagramChannel.open(StandardProtocolFamily.INET);                    final
     Selector selector = Selector.open())
     {
035
036                                  "Starting subscriber    ,
                 System.out.printf(%s"                     id);
037              initChannel(channel, selector);
038              doSchedule(channel);
039
040                    (!Thread.currentThread().isInterrupted())
                 while {
041                    (selector.isOpen())
                     if {
042                              numKeys =
                       final int selector.select();
043                       (numKeys    )
                       if >         0{
```

```
044                         handleKeys(channel,
                            selector.selectedKeys());
045                     }
046                 }else {
047                     Thread.currentThread().interrupt();
048                 }
049             }
050                 (IOException e)
            }catch {
051                                         "unable to run           ,
                throw new RuntimeException(subscriber"               e);
052         }finally {
053             this.scheduler.shutdownNow();
054         }
055     }
056
057                                     DatagramChannel             Selector
        private void initChannel(final channel,               final selector)        throws
    IOException
    {
058                 !Objects.isNull(channel) &&
            assert Objects.isNull(selector);
059
060         channel.configureBlocking(false);
061         channel.setOption(StandardSocketOptions.SO_REUSEADDR,true);
062         channel.bind(this.hostAddress);
063         channel.setOption(StandardSocketOptions.IP_MULTICAST_IF,this.networkInterface);
```

```
064          channel.join(this.group,this.networkInterface);
065          channel.register(selector,
             SelectionKey.OP_READ);
066      }
067
068                                      DatagramChannel              Set<SelectionKey>
         private void handleKeys(final channel,                 final keys)
           IOException
     throws {
069                !Objects.isNull(keys)  &&
             assert !Objects.isNull(channel);
070
071               Iterator<SelectionKey> iterator =
             final keys.iterator();
072               (iterator.hasNext())
             while {
073
074                    SelectionKey key =
                 final iterator.next();
075               try {
076                     (key.isValid()  &&  key.isReadable())
                    if {
077                       Pulse.read(channel).ifPresent((pulse)  ->
                          {
078                                .pulses.put(pulse.getId(),
                             thispulse);
079                       });
080                    }else {
081                                                              "key not
                       throw new UnsupportedOperationException(valid."          );
```

```
082                 }
083             }finally {
084                 iterator.remove();
085             }
086         }
087     }
088
089                                     DatagramChannel channel)
        private void doSchedule(final {
090         assert !Objects.isNull(channel);
091
092                                 Runnable()
            doSchedule(channel,new {
093                         run()
                public void {
094                                 .pulses.forEach((id, pulse) ->
                    Subscriber.this{
095                     if
    (pulse.isDead(Constants.Schedule.DOWNTIME_TOLERANCE_DEAD_SERVICE_IN_MILLISECONDS))
    {
096                                                     "FATAL   : %s        ,
                        System.out.println(String.format(removed"            id));
097                         Subscriber.this.pulses.remove(id);
098                     }else if
    (!pulse.isValid(Constants.Schedule.DOWNTIME_TOLERANCE_IN_MILLISECONDS))
    {
099                                                     "WARNING : %s is     ,
                        System.out.println(String.format(down"               id));
100                     }else {
```

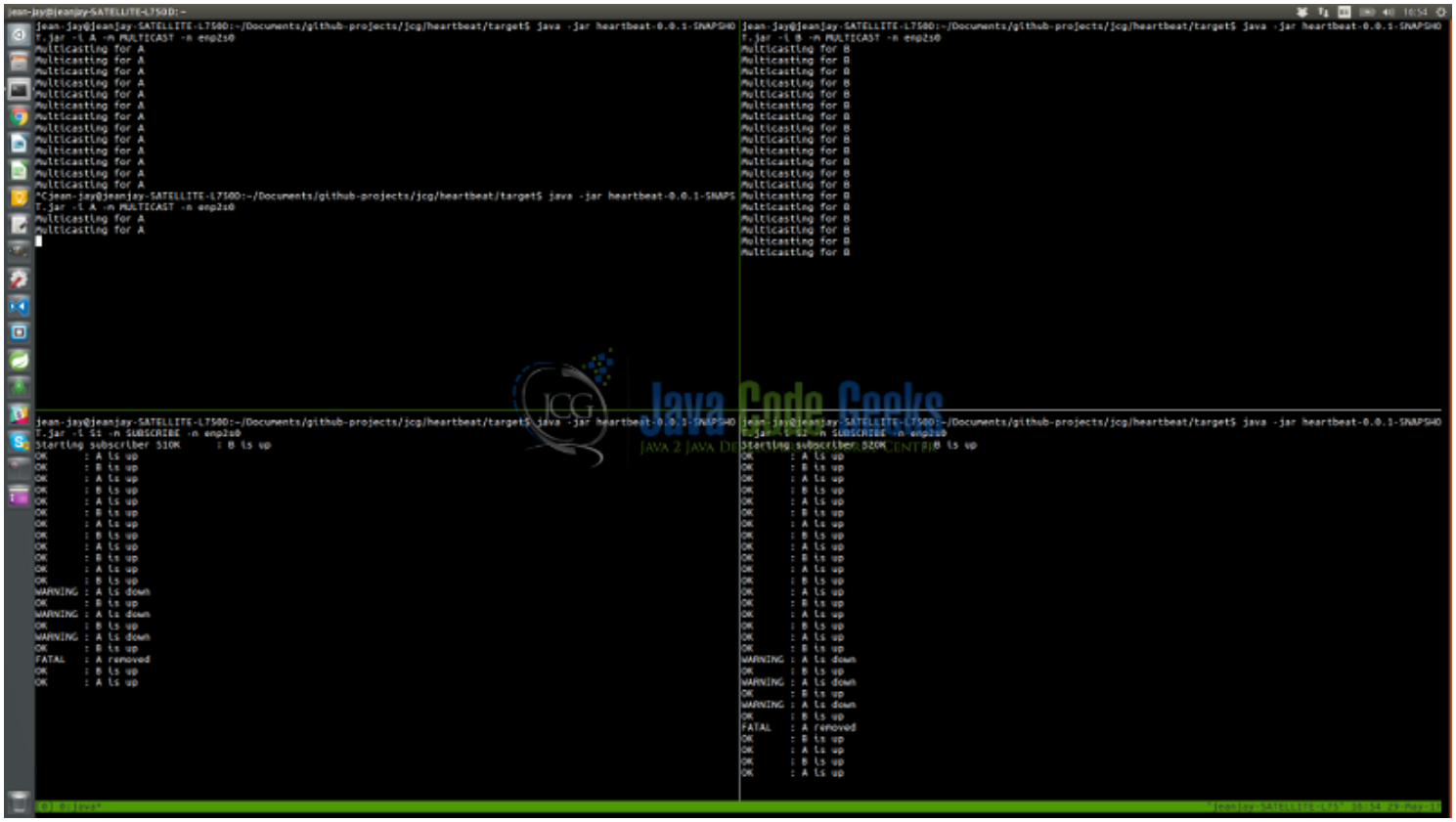
```
101                                                    "OK      : %s is    ,
                        System.out.println(String.format(up"                id));
102                     }
103                 });
104             }
105         }, 0L, Constants.Schedule.PULSE_DELAY_IN_MILLISECONDS,
            TimeUnit.MILLISECONDS);
106     }
107 }
```

- line 16: we create a ScheduledExecutorService to schedule the polling of the heartbeat pulses we have received thus far via UDP multicast
- line 17: we create a InetSocketAddress for the specified port and instantiate it for the “localhost”
- line 21: we create NetworkInterface for the specified interface name, this will be the interface through which the `Subscriber` will receive UDP multicast heartbeat pulses
- line 22: we create a InetAddress representing the multicast group from which we will receive multicast messages
- line 34: we open the DatagramChannel but also specify the `ProtocolFamily` and this should correspond to the address type of the multicast group this Channel will be joining.
- line 37-38: we initialize the Channel and schedule the polling of heartbeat pulses
- line 40-49: while the current thread is still running we utilize the `Selector` and await incoming UDP multicast heartbeats in a non-blocking way.
- line 63-64: we set the multicast interface and join the multicast group using the multicast interface
- line 77-79: we read a Pulse from the UDP multicast packet.

6. Running the program

The example project is a maven project and must be built into a “fat” or “uber” jar by issuing the following command `mvn clean install package`. The resulting artifact can be found in the “target” folder located in the project root folder. The project can be run in two modes, one being “MULTICAST” and the other being “SUBSCRIBE”. Obviously the “MULTICAST” mode will publish packets (heartbeats) to the multicast group and the “SUBSCRIBE” mode will receive said heartbeats.

The beauty of the example is that you can spin up as many “MULTICAST” processes as you wish (ensure you give them all unique id’s) and as many “SUBSCRIBE” processes as you wish (ensure you give them also unique id’s). This can be done in random order meaning “MULTICAST” or “SUBSCRIBE” in any order. Simply put, as soon as heartbeats arrive, the subscribers will know about it and begin reporting as shown below:



Running 2 multicast processes and 2 subscribe processes

On top are the two “MULTICAST” processes and on the bottom are the two “SUBSCRIBE” processes. Notice how the subscribers report the other processes being “up” until I terminate one of them (“A” – top left) and then after a configurable tolerance is exceeded, the last packet / pulse is reported as too old, the subscribers notify us that “A is down”. After a short while A is removed and considered dead. We then bring A back up and you see immediately a heartbeat is received and the subscribers begin reporting “A is up” along with “B”, which never went down.

To run this example you need a network interface, ie: you need to be plugged into a network point as the network interface name is required. Although I have been successful in sending packets via the wireless interface as the “BROADCASTER”, to the multicast group, and receiving them in the “SUBSCRIBER” via a LAN interface, I would highly recommend using a LAN (cable) interface for both MULTICAST and BROADCAST processes.The interface will serve as the NetworkInterface on which the multicast group is joined.

The following arguments are required to run the program:

- -i ‘Unique id of the process’ : eg: A or B or S1
- -m ‘Mode’ : can be one of MULICAST or SUBSCRIBE
- -n ‘NetworkInterface name’ : be sure to run `ifconfig` (linux) or `ipconfig` (windows) to find the interface name. eg: eth0
- Running a MULTICAST process: `java -jar heartbeat-0.0.1-SNAPSHOT.jar -m MULTICAST -i A -n eth0`
- Running a SUBSCRIBE process: `java -jar heartbeat-0.0.1-SNAPSHOT.jar -m SUBSCRIBE -i s1 -n eth0`

7. Summary

In this example we demonstrated how to build a simple heartbeat application using UDP and Java NIO. We took it a bit further and leveraged multicasting as a method to publish our heartbeat signals across the network to interested parties. These interested parties can change state, notify persons or even try to resurrect down / faulty services when they become aware of problems via the heartbeat signals.

8. Download the source code

This was a Java Nio Heartbeat tutorial

Download
You can download the full source code of this example here: **Java Nio Heartbeat tutorial**