1. UDP Sockets

Im ersten Schritt sollen die Machinensensoren Informationen liefern. Dazu sollen z.B. Temperatur, Luftfeuchtigkeit, Helligkeit, etc. von jeweils einem Sensor erfasst bzw. simuliert werden. Die Sensorinformationen sollen sich ständig ändern. Weiterhin soll der IoT-Gateway die Sensor-Informationen in regelmäßigen Zeitabständen mittels eines zum implementierenden Request-Reply-Protokolls über UDP abfragen und auf der Standardausgabe ausgeben. Jeder Sensor als auch der IoT-Gateway sollen als eigenständiger Prozess (bzw. als eigenständiger Docker Container) laufen.

Wir haben uns für eine $asynchrone\ Methode$ mittels eines $DatagramChannels$ entschieden, der in Synergie mit einem $Selector$ beliebig viele Anfragen gleichzeitig verarbeiten kann. Ein $Selector$ funktioniert so, dass er spezielle $Keys$ anlegt, die entweder $read$ oder $write$ Berechtigungen haben. Standartmäßig wird ein Key immer als $readable$ initialisiert, um auf Anfragen reagieren zu können. Außerdem öffnet jeder $Selector$ einen eigenen $DatagramChannel$, um gleichzeitige Anfragen auch gleichzeitig bearbeiten zu können. In unserem Beispiel sieht das ganze so aus:

```
1.  /**
2.   * Run client method.
3.   * @throws IOException if something went wrong.
4.   */
5.   public void start() throws IOException {
6.       while (true) {
7.           this.selector.select();
8.           final Iterator<SelectionKey> selectedKeys =
9.           this.selector.selectedKeys().iterator();
10.
11.              while (selectedKeys.hasNext()) {
12.                  SelectionKey key = selectedKeys.next();
13.                  selectedKeys.remove();
14.
15.                  if (!key.isValid()) continue;
16.
17.                  if (key.isReadable()) read(key);
18.                  if (key.isWritable()) write(key);
19.              }
20.         }
21.  }
```

Weiterhin gibt es eine $read$ Methode, die folgender Maßen aussieht:

```
1.  /**
2.   * Read incoming requests with specific key.
3.   * @param key as identifier.
4.   * @throws IOException if something went wrong.
5.   */
6.   private void read(SelectionKey key) throws IOException {
7.       final DatagramChannel client = (DatagramChannel) key.channel();
8.       final Buffer buffer = (Buffer) key.attachment();
9.       buffer.setSocketAddr(client.receive(buffer.getReq()));
10.      buffer.getReq().flip();
11.
12.      int limits = buffer.getReq().limit();
13.      byte[] bytes = new byte[limits];
14.      buffer.getReq().get(bytes, 0, limits);
15.      final String request = new String(bytes);
16.      System.out.println(request);
17.      generateData(request, buffer, key);
18.    }
```

Um eine Anfrage zu beantworten musst auf demselben $DatagramChannel$ zurückgeschrieben werden. Dies sieht wie folgt aus:

```
1.  /**
2.   * Write data to datagram with specific key.
3.   * @param key as identifier.
4.   * @throws IOException if something went wrong.
5.   */
6.   private void write(SelectionKey key) throws IOException {
7.       final DatagramChannel client = (DatagramChannel) key.channel();
8.       final Buffer buffer = (Buffer) key.attachment();
9.       client.send(buffer.getResp(), buffer.getSocketAddr());
10.
11.      key.interestOps(SelectionKey.OP_READ);
12.  }
```

Der Server, welcher die $Pullrequests$ an die $Clienten$ schickt, arbeitet mit demselben Prinzip. Der einzige Unterschied: Die Anfragen laufen in $separaten\ Threads$, da der Main-Thread nicht durch eine andere Aktion unterbrochen werden soll. Wir haben dies mit einem $ThreadScheduler$ realisiert.

```
1.  final ScheduledExecutorService executorService1 =
    Executors.newSingleThreadScheduledExecutor();
2.  executorService1.scheduleAtFixedRate(this::request, 0, INTERVAL_MS,
    TimeUnit.MILLISECONDS);
```

Weiterhin warden geschickte Anfragen in einer Wrapper-Klasse gespeichert. So können wir herausfinden, ob ein Sensor einen Timeout hat und somit nicht erreichbar ist.

```
1.  /**
2.   * Send requests to all clients.
3.   */
4.   private void request() {
5.       Arrays.stream(sAddresses).forEach(address -> {
6.           try {
7.               checkTimeout(address);
8.
9.             this.server.send(ByteBuffer.wrap("get_data"
10.                .getBytes(StandardCharsets.UTF_8)), address);
11.           } catch (IOException io) {
12.                io.printStackTrace();
13.           }
14.       });
15.       System.out.println("Database size: " + this.data.size());
16.   }
17.
18. /**
19.  * Check for connection timeout.
20.  * @param address of the client.
21.  */
22.  private void checkTimeout(InetSocketAddress address) {
23.      if (System.currentTimeMillis() - this.timers.getTimer(address) <
24.      this.INTERVAL_MS + 1000) return;
25.
26.      this.logger.warn("Client ("
27.              + address.getAddress().getHostAddress()
28.              + " | " + address.getPort()
29.              + ") is offline!");
30.      saveData(false, null, address);
31.      this.timers.resetTimer(address);
32.   }
```