Verteilte System - Übungen

Patrick Jungk

28. September 2021

Inhaltsverzeichnis

1 Einführung

1.1 Input/Output

1.1.1 Aufgabe

Erstellen sie ein Programm, dass...

1. einen Text (lesbare Zeichen) in eine Textdatei schreibt.
2. einen Text (lesbare Zeichen) aus einer Textdatei einliest.
3. dem Texteintrag einen Zeitstempel hinzufügt
4. einen neuen Eintag hinzufügt.

1.1.2 Hilfe

Hilfsklassen (JAVA) und Hinweise

- java.util.Calendar/java.time.Instant: Zeitformatierungen und Zeitausgabe
- java.io.∗: Schreiben und Lesen von u.A. Dateien

2 Lokale Nebenläufigkeit

2.1 Multi-Prozess/Multi-Threaded In/Out

2.1.1 Aufgabe

Erweitern Sie ihr Programm um folgende Punkte:

1. erweitern Sie Ihr Programm, damit jede Instanz (Prozess und Thread) eine eindeutige Bezeichnung bekommen kann und schreiben Sie diese Bezeichnung in die Textdatei pro Zeile analog
2. es sollen im weiteren ca. 100 Einträge erstellt werden, damit die Datei nicht allzu groß wird.

```
Prozess 1: Test text 10.10.2020 9:35:12.345
Prozess 2: Test text 10.10.2020 9:36:12.345
                    ...
```

3. 2 Prozesse erstellen asynchron Texteinträge in eine gemeinsame Datei (starten sie dazu Ihr Programm zweimal)
4. lassen Sie jeden Pozess mit zufälliger Pause Einträge erstellen und lesen (und ausgeben)
5. stellen Sie das Programm auf 2 Threads um (auch hier mit zufälliger Pause)
6. was passiert, wenn Sie eine feste Pause (z.b. 250) einstellen?

Was fällt Ihnen bei Prozessen auf und was bei Threads?

2.1.2 Hilfe

Hilfsklassen (JAVA) und Hinweise

- java.lang.Thread: Threads, die gestartet werden können (mittels start()); können überschrieben werden.
- java.lang.Runnable: Interface; abgeleitete Klassenobjekte können den Threads im Konstruktor übergeben werden.

Eine Klasse kann als „Runnable“ definiert werden.

```
/**
 * This class is used as an example for Threading
 */
package eu.boxwork.dhbw.examples;

public class MyRunnableClass implements Runnable {
/**
```

```
 * the run method will be called by the thread
 * when we use the start()-Method
 */
 public void run()
 {
  System.out.println("Am I a threaded method?");
 }
}
```

Während der Thread diese übergeben bekommt und die Run-Methode implizit aufruft.

```
/**
 * This class is used as an example to all a Runnable
 */
package eu.boxwork.dhbw.examples;

public class Application {
/**
 * we simply start the main
 */
 public static final int main(String[] args)
 {
  // create our runnable class object
  MyRunnableClass runnableObject = new MyRunnableClass();
  // create a Thread
  Thread runner = new Thread(runnableObject);
  // we can setup additional settings, like e.g. name
     and priority
  //..

  // now we start the runner
  runner.start(); // this calls the run() Method in the
     runnable class object

  // as a main, we wait for the thread to finish
  runner.join();
 }
}
```

2.1.3 Herausforderung

Lassen Sie die Threads eine eindeutige laufende Nummer (streng monoton steigend; ohne Lücke, Doppelung und in korrekter Reihenfolge) den Texteinträgen (in einer gemeinsamen Datei) hinzufügen. Stellen Sie auch hier einmal eine zufällige, eine feste und kein Pause ein.

1. Was fällt Ihnen auf?
2. Was passiert bei 2 Anwendungen (Prozessen)?
3. Was passiert bei 2 Anwendungen (Prozessen) mit je 2 (oder mehr Threads)?
4. Ist die Nummer noch wie gefordert?

3 Kommunikation

3.1 Sockets

3.1.1 Aufgabe

Erweitern Sie Ihr Programm, um folgende Punkte

1. erstellen Sie einen Client und einen Server mittels Sockets (Berkley)
2. der Client soll einen Text (inkl. Zeitstempel und laufender Nummer) an den Server schicken, der diesen festschreibt
3. definieren Sie eine Antwort, welche der Server an den Client zurückgibt
4. der Client soll den Server nach dem letzten Eintrag fragen (der Client gibt diesen aus)

3.1.2 Hilfe

Hilfsklassen (JAVA) und Hinweise

- java.net.*
- java.net.ServerSocket
- java.net.Socket
- java.net.DatagrammSocket
- java.io.*

Ein Server kann mittels eines *ServerSockets* initialisiert werden. Der *Socket* wird dann für die Kommunikation verwendet. Hinweis: dieser Socket kann nun auch von einem anderen Thread für das Handling der Kommunikation verwendet werden. Auch dieser kann intern einen Nachrichtenpuffer halten, damit andere Threads hiermit Nachrichten verteilen können.

```
/**
 * This class is used as an example for a Server Socket
 * this could be a own thread
 */
package eu.boxwork.dhbw.examples;
```

```
public class MyServer {
 // this max client definition is not available in "old"
 // implementations
 public static final int maxIncomingClients = 100;

/**
 * this method initialises the server
 * @param dns name like "localhost"
 * @param port port to use
 * @return the created socket after client connected
 */
 public Socket initialise(String dns, int port)
 {
         ServerSocket serverSocket = new ServerSocket(
                port,
                maxIncomingClients,
                InetAddress.byName(dns));
         // no need for an additional bind, but could be
            done here
   Socket clientCommSocket = serverSocket.accept();
   return clientCommSocket;
 }
}
```

Ein Client erstellt einen *Socket* und übergibt dabei das Ziel. Die Kommunikation baut auf diesem Socket auf.

```
/**
 * This class is used as an example for a Client Socket
 * this could be a own thread
 */
package eu.boxwork.dhbw.examples;

public class MyClient {
/**
 * this method initialises the client
 * @param dns distination like "localhost"; can also be
    an IP
 * @param port port to connect to
 * @return the created socket after connection is
    established
 */
 public Socket initialise(String dns, int port)
 {
   Socket clientSocket = new Socket(
```

```
                dns, port);
        // no need for an additional bind, but could be
            done here
   return clientSocket;
 }
}
```

3.1.3 Herausforderung

Client und Server sollen sich die Textliste sichern. Beide Listen sollen am Ende identisch sein. Bricht der Client ab, so soll er an der letzten Stelle wieder weitermachen. Kommt ein neuer Client hinzu, soll dieser auch den letzten Eintrag vom Server bekommen und weitermachen mit dem Prozess.

1. Was fällt Ihnen bei einem Client auf?
2. Was passiert bei 2 Clients: wer hat am Ende die gesamte Sicht auf alle Daten, die geschickt wurden?
3. Was müsste getan werden, damit alle Clients den gesamten Stand der Daten immer haben?

3.2 Nachrichten

3.2.1 Hintergrund

Im verteilten System werden immer wieder Nachrichten verschickt. Diese Nachrichten sind durch u.A. durch Alvisi und Marzullo (1998) charakterisiert durch:

- Senderinformationen
- Empfängerinformationen
- Sequenznummer (um Duplikate zu vermeiden)
- Auslieferungsnummer (um den Zeitpunkt des Auslieferns zu entscheiden)

Weitere Informationen sind machmal sinnvoll, wie:

- Sendezeitstempel
- Status (für die interne Verarbeitung, wie erneutes Senden)
- Type (um mehrere Nachrichtentypen zu unterscheiden)

Immer enthält eine Nachricht einen Payload.

3.2.2 Aufgabe

Verschicken Sie anstatt eines String ein Message-Objekt über den Socket. Erweitern Sie ihr bisheriges Programm um folgende Punkte:

- Senden eines Message Objekts (von Client zu Server)
- Empfangen eines Message Objekts und Ausgabe des Inhalt am Server

3.2.3 Hilfe

Über Sockets können nur Bytes verschickt werden. Um Objekte zu verschicken, bietet JAVA die Möglichkeit Objekte „flach zu klopfen“. Dafür müssen die Objekte das Interface *Serialisable* implementieren, einen Default-Konstruktor besitzen und für jede Variable ein Getter/Setter Paar haben. Es gibt Listen, welche dieses Interface implementieren. Damit können dann Listen von Objekten verschickt werden.

```
/**
 * This class is a sendable message object, further
    information may be added
 */
package eu.boxwork.dhbw.examples;

public class Message implements Serialisable {
 private String sender;
 private String receiver;
 private Object payload;
 private Instant time = Instant.now();
 private String type; // may be an enum too
 private int sequenceNo = -1;
 /**
  * Default-Constructor
        */
 public Message()
 {}

 /* GETTER - SETTER*/
 public Object getPayload() { return payload; }
 public void setPayload(Object payload) { this.payload =
    payload; }

 public Instant getTime() { return time; }
 public void setTime(Instant message) { this.time =
    time; }

 public String getType() { return type; }
 public void setType(String type) { this.type = type; }
```

```
 public String getSender() { return sender; }
 public void setSender(String sender) { this.sender =
    sender; }

 public String getReceiver() { return receiver; }
 public void setReceiver(String receiver) {
    this.receiver = receiver; }

 public int getSequenceNo() { return sequenceNo; }
 public void setRSequenceNo(int sequenceNo) {
    this.sequenceNo = sequenceNo; }
}
```

Um ein Object zu senden/empfangen, werden die Klassen *ObjectInputStream* und *ObjectOutputStream* benötigt.

```
/**
 * This class is used as an example for reading Object
    from a socket
 */
package eu.boxwork.dhbw.examples;

public class ObjectMessageReader {
/**
 * this method reads objects from a given socket
 * @param socket socket to read an object from
 * @return the message object or null, in case of an
    error
 */
 public Message read(Socket socket)
 {
  Message ret = null;
  try {
   InputStream is = socket.getInputStream();
   ObjectInputStream ois = new ObjectInputStream(is);
   ret = (Message)ois.readObject();
  } catch (Exception e)
  {
   System.err.println(e.toString);
  }
  return ret;
}
```

4 Definitionen

Die Grenzen zwischen Client und Server verschwimmen. Daher eine kleine Definition der Begriffe vorab:

- **Client**: außenstehender Client, der mit den verteilten System agieren will
- **Server**: Knoten, der Nachrichten und Verbindungen entgegennehmen kann. Jeder Knoten ist ein Server
- **Knoten**: Teil des verteilten Systems
- **Master/Leader/Koordinator**: Rolle innerhalb des verteilten Systems. Koordinierende/schreibende Funktion
- **Slave/Follower**: Rolle innerhalb des verteilten Systems. Meist nur weiterleitend, bzw. sekundär aktiv

5 Koordination

5.1 Rollen

5.1.1 Hintergrund

Ein Client (außerhalb des Systems) möchte Nachrichten in einem verteilten System verarbeitet haben bzw. die letzten Nachrichten (z.B. die letzten 10) abrufen. Das System ist in mehrere Rollen aufgeteilt.
Die Rollen innerhalb des System sind essentiell für die Verarbeitung der Daten. In dieser Übung soll das „Rollenverständnis“ geschärft werden. Definiert einen *Master* und mehrere *Slaves*. Die Aufgabe des *Masters* wird es sein, die Liste der bisherigen Nachrichten zu verwalten und zurückzugeben. Der *Slave* hingegen schickt alle Anfragen an den Master und antwortet dem Client mit der Antwort des *Masters* quasi als Proxy.

5.1.2 Aufgabe

Client schickt Nachricht entweder an Slave oder an den Master einen Request. Client kann 2 Aktionen durchführen:

- **WRITE(Text)**: es soll ein *Text* geschieben werden; als Antwort erwartet der Client ein „OK“/“NOK“ mit dem Zeitstempel des Schreibens
- **READ(Anzahl)**: es sollen eine *Anzahl* der letzten Einträge an den Client geschickt werden.

Definieren Sie die Kommandos und die Antworten, sodass der Client und das verteilte System damit umgehen können.

Erweitern Sie ihr Programm um folgende Punkte:

- erweitern Sie Ihr Programm, dass sie beim Start einen Prozess als Master definieren können (z.B. als Start-Parameter)
- jeder weiterere Prozess soll als „Slave“ getaggt und gestartet werden (z.B. als Start-Parameter)
- erstellen Sie einen Client, in der Lage ist die beiden Aktionen durchzuführen (mittels der definierten Kommandos); der Client gibt entsprechend die Antwort des verteilten Systems aus
- Slave und Master sollen entsprechend des Requests agieren
  - der **Slave** soll die Schreib- und Leseanweisungen an den Master weiterleiten und die Antwort an den Client schicken
  - der **Master** schreibt den Text und stempelt die Nachricht als Response mit dem Zeitstempel; der Master gibt die Liste der letzten X Nachrichten zurück

HINWEIS: jeder Slave-Prozess soll später in der Lage sein, als Master zu funktionieren.

5.1.3 Hilfe

Sichern Sie sich die Rolle am Knoten. Ein Enum kann hier sinnvoll sein.

```
/**
 * This enum can be used to determine the role
 */
package eu.boxwork.dhbw.examples;

public enum Role {
  UNKNOWN, MASTER, SLAVE;
}
```

Ist ein Knoten noch nicht im Cluster, so ist die Rolle „unbekannt“.

5.1.4 Herausforderung

Wenn er Master ausfällt, kann kein Slave mehr etwas „schreiben“. Auch neue Clients können nichts schreiben, bzw. ausgeben.

1. Mit welchem Algorithmus würden Sie diesem Ausfall begegnen?
2. Wo liegen die Einschränkungen, ab wann dieser Algorithmus nicht mehr funktioniert?

5.2 Cluster

5.2.1 Hintergrund

Im verteilten System ist es immer wieder notwendig die beteiligten Prozesse in ihrer Rolle zu kennen. Diese bilden dann das Cluster. Daher muss das System in der Lage sein, die beteiligten Knoten zu aktualisieren. Essentielle Aktionen sind dabei:

- **JOIN**: ein Knoten tritt einem Cluster hinzu
- **LEAVE**: ein Knoten tritt aus einem Cluster aus
- **UPDATE**: nachdem ein Cluster sich geändert hat, muss den Clusterknoten, der neue Zustand mitgeteilt werden

Es ergibt sich dadurch letztendlich eine Clusterverwaltung.

5.2.2 Aufgabe

Definieren Sie die Kommandos für die Clusterverwaltung. Geben Sie dabei alle notwendigen Informationen für einen Knoten bekannt:

- **Adresse**: IP und Port des Prozesses; ggf. eine ID für den Prozess
- **Rolle**: Master oder Slave

Erweitern Sie ihr Programm um folgende Punkte:

- erstellen Sie eine Liste von Knoten, die jeder Prozess halten wird
- ein neuer Knoten muss sich beim Master anmelden
- nach einer Anmeldung schickt der Master an alle Knoten die neue Cluster-Liste
- jeder Knoten gibt diese in der Konsole (für Debugging) aus
- terminiert ein Knoten, so muss sich dieser zunächst beim Master abmelden

5.2.3 Hilfe

Einige Listen-Implementierungen sind serialisierbar. Objekte in der Liste müssen auch serialisierbar sein, damit die Liste per komplett serialisierbar ist.

```
/**
 * This class is a sendable node object
 */
package eu.boxwork.dhbw.examples;

public class Node implements Serialisable {
 private String ip = "";
```

```
 private int port = -1;
 private int id = -1;
 private Role role = Role.UNKOWN;
 /**
  * Default-Constructor
        */
 public Node()
 {}
 /* GETTER - SETTER */
 public String getIp() { return ip; }
 public void setIp(String ip) { this.ip = ip; }

 public int getPort() { return port; }
 public void setPort(int port) { this.port = port; }

 public int getId() { return id; }
 public void setId(int id) { this.id = id; }

 public Role getRole() { return role; }
 public void setRole(Role role) { this.role = role }
}
```

Um ein Objekt zu senden/empfangen, werden die Klassen *ObjectInputStream* und *ObjectOutputStream* benötigt. Knoten können als Payload einer Message mitgeschickt werden. Dabei kann entweder der Knoten als JSON serialisiert als „String“ hinterlegt werden, oder der Payload einer Message kann generalisiert werden und beim prüfen muss auf den Typ geachtet werden (instanceof).

5.2.4 Herausforderung

Innerhalb eines Clusters können Knoten jederzeit wegfallen

1. Wann müssen wir Änderungen im Cluster erkennen?
2. Wie erkennen wir Änderungen im Cluster, die nicht den Master betreffen?

5.3 Hintergrund

Basiert das Verteilte System in Koordination und Synchronisation auf einem zentralen Ansatz, so muss eine Wahl abgehalten werden, um beim Ausfall des Koordinators einen neuen zu wählen. Basisnachrichten sind dabei wie folgt.

- **ELECT**: ein Knoten startet eine Wahl mit entsprechendem Payload
- **ELECT-RESPONSE**: ein Knoten sendet auf diese Wahlanfrage die Antwort mit entsprechendem Payload
- **ELECT-RESULT**: nach der Wahl wird das Wahlergebnis des neue Koordinators bekannt gegeben

Als erste Amtshandlung könnte der Koordinator nun dem Cluster mitteilen, welche Prozesse aktuell im Cluster vorhanden sind.

Essenziell ist ggf. dass Nachrichten quittiert werden. Der Status der Nachricht muss ggf. gepflegt werden, damit im Fehlerfall entsprechend (z.B. durch Resent) reagiert werden kann. Basisnachrichten sind dabei.

- **AQU**: ein Knoten bestätigt durch solch eine Nachricht, den Empfang einer Nachricht; die Nachrichten - ID wird mitgereicht
- **NAQU**: ein Knoten schickt dem Sender, dass er eine Nachricht nicht bekommen hat; die Nachrichten - ID wird der fehlenden Nachricht, oder der letzten empfangenen Nachricht wird mitgeschickt
- **DONE**: ein Knoten kann die (ggf. asynchrone) Durchführung einer Aktion, die durch eine Nachricht getriggert wurde, als abgeschlossen bestätigen
- **AQUIRE**: ein Knoten signalisiert, dass er eine Resource alleine nutzen will. Die Resource wird benannt. Eine Request-ID wird mitgeschickt

Wählen Sie ***eine*** der folgenden Aufgaben.

5.4 Ring-Algorithmus

5.4.1 Aufgabe

Implementieren Sie den Ring-Algorithmus zur Bestimmung des neuen Koordinators.

- gleich zu Beginn soll der erste Knoten eine Wahl abhalten, auch wenn noch keine Koordinator vorhanden ist
- fällt der Koordinator aus, so hält der erste Prozess, der den Ausfall bemerkt eine Wahl ab

5.4.2 Hilfe

Eine spezielle Wahlnachricht sollte implementiert werden, welche die notwendigen Informationen enthält. Ist die Nachricht serialisierbar, kann diese einfach z.B. über Sockets versendet werden. Um ein Object zu senden/empfangen, werden die Klassen *ObjectInputStream* und *ObjectOutputStream* benötigt.

```
/**
 * This class is as an implementation example for a
    election message (ring algorithm)
 * based on a message
 */
package eu.boxwork.dhbw.examples;

public class ElectionMessage extends Message {
  // node that starts the election
```

```
  private Node startingNode = null;
  // list of nodes to be sent
  private List<Node> electionNodes = new ArrayList<>();

  // default constructor
  public ElectionMessage()
  {
    super();
  }

  /* GETTERS/ SETTERS */
  public void setStartingNode(Node in) {
      this.startingNode = in; }
  public Node getStartingNode(){ return
      this.startingNode; }

  public void setElectionNodes(List<Node> in) {
      this.electionNodes = in; }
  public List<Node> getElectionNodes(){ return
      this.electionNodes; }

  /* methods that make life easier */
  public void addNode(Node in)
  {
    electionNodes.add(in);
  }
}
```

5.5 Bully-Algorithmus

5.5.1 Aufgabe

Implementieren Sie den Bully-Algorithmus zur Bestimmung des neuen Koordinators.

- gleich zu Beginn soll der erste Knoten eine Wahl abhalten, auch wenn noch keine Koordinator vorhanden ist
- fällt der Koordinator aus, so hält der erste Prozess, der den Ausfall bemerkt eine Wahl ab

5.5.2 Hilfe

Eine spezielle Wahlnachricht sollte implementiert werden, welche die notwendigen Informationen enthält. Ist die Nachricht serialisierbar, kann diese einfach

z.B. über Sockets versendet werden. Um ein Object zu senden/empfangen, werden die Klassen *ObjectInputStream* und *ObjectOutputStream* benötigt. Die Sortierung der Knoten muss hier anhand der ID gegeben sein.

```
/**
 * This classes are examples showing the messages needed
    for the bully algorithm
 * based on a message. The ElectionMessage could also be
    used as a message to signalize the handover
 */
package eu.boxwork.dhbw.examples;

public class ElectionMessage extends Message {

  // node that starts the election
  private Node startingNode = null

  public ElectionMessage()
  {
    super();
  }

  /* GETTERS/ SETTERS */
  public void setStartingNode(Node in) {
     this.startingNode = in; }
  public Node getStartingNode(){ return
     this.startingNode; }
}

public class ElectionResponseOK extends Message {
  // node that takes over
  private Node node = null;

  public ElectionResponseOK()
  {
    super();
  }

  /* GETTERS/ SETTERS */
  public void setNode(Node in) { this.node = in; }
  public Node getNode(){ return this.node; }
}
```

5.6 Wahl - Ergebnis

Bei allen Wahl-Algorithmen muss am Ende der Koordinator bekannt gegeben werden, erst dann ist die Wahl am Ende.

```
/**
 * This class is as an implementation example an
    election result
 * based on a message
 */
package eu.boxwork.dhbw.examples;

public class ElectionResult extends Message {
  // node that should be the new coordinator
  private Node coordinator = null;

  /* GETTERS/ SETTERS */
  public void setCoordinator(Node in) { this.coordinator
     = in; }
  public Node getCoordinator(){ return this.coordinator;
     }
}
```

5.6.1 Herausforderung

Innerhalb eines Clusters können Knoten jederzeit so wegfallen, dass das Cluster geteilt wird. Diskutieren sie:

1. Wie gehen wir damit um, wenn das Cluster geteilt wird?
2. Wie können wir eine solche Teilung überhaupt bemerken (automatisiert)?
3. Was machen wir, wenn das Cluster wieder physisch vereinigt wird?

Implementieren Sie entsprechende Strategien um ihr dediziertes Problem zu lösen (je nach Wahl der Aufgabe).

6 Fehlerhandling

6.1 Hintergrund

Bei so viel Kommunikation kann es an vielen Stellen zu Fehlern kommen Nachrichten IDs können dabei helfen den Überblick zu wahren. Doch was im Fehlerfall zu tun ist, hängt stark von der Situation und der Auswirkung ab. Es können Knoten eines Clusters ausfallen und neu hinzukommen. Beim Ausfall kann es zum Datenverlust kommen, der nach Möglichkeit kompensiert werden muss. Sendonce, Sendmany sind hier Ansätze die zu ganz unterschiedlichen Situationen führen können.

Wählen Sie ***eine*** der folgenden Aufgaben.

6.1.1 Fehler: Empfänger

Der Empfänger fällt aus, während eine Nachricht unterwegs ist, bzw. verarbeitet wird. Definieren sie ein Fehlerhandling auf Client Seite je nachdem, wann der Sender ausfällt und implementieren Sie den Ansatz Zeitpunkt des Ausfalls:

- der Sender hat die Nachricht zum Empfänger gesendet, aber es wurde nicht empfangen
- der Sender hat die Nachricht zum Empfänger gesendet, aber der Client stürzt nach der Verarbeitung vor dem Senden einer Antwort ab

6.1.2 Fehler: Sender

Der Sender fällt aus, während er eine Nachricht an den Empfänger sendet. Definieren sie ein Fehlerhandling seitens des Senders je nachdem, wann der Sender ausfällt und implementieren Sie den Ansatz Zeitpunkt des Ausfalls:

- der Sender hat die Nachricht zum Empfänger noch nicht gesendet.
- der Sender hat die Nachricht bereits abgeschickt und stürzt kurz darauf ab; eine Antwort empfängt er nicht

7 Synchronisation und verteilte Daten

7.1 Hintergrund

Für den Zugriff auf gemeinsame Daten ist eine Synchronisation oft wünschenswert. Damit Daten korrekt verarbeitet werden ist ein wechselseitiger Ausschluss im verteilten System mit mehreren Ansätzen möglich.

Wird der Zugriff auf Daten dezentral organisiert, müssen Nachrichten für die Datenverteilung implementiert werden. Abgleiche sind an der Tagesordnung. Der Zugriff auf Resourcen muss ggf. sychronisiert werde um Kollisionen zu erkennen. Nachrichten sind hierbei.

- **AQUIRE**: ein Knoten signalisiert, dass er eine Resource alleine nutzen will. Die Resource wird benannt. Eine Request-ID wird mitgeschickt
- **ACCEPT**: ein Knoten akzeptiert den Versuch eines Knotens, eine Resource zu allokieren. Die Request-ID wird mitgeschickt
- **NOT-ACCEPT**: ein Knoten akzeptiert den Versuch eines Knotens, eine Resource zu allokieren, nicht. Die Request-ID wird mitgeschickt
- **ENTER**: ein Knoten signalisiert, dass er eine Resource nun zu nutzen anfängt

- **EXIT**: ein Knoten signalisiert, dass er mit der Nutzung einer Resource fertig ist und diese somit freigegeben ist

Auch hier kann es essenziell sein dass Nachrichten quittiert werden.
Wählen Sie ***eine*** der folgenden Aufgaben.

7.2 Token-Ring-Algorithmus

7.2.1 Aufgabe

Implementieren Sie den Token-Ring-Algorithmus für den Zugriff aus eine gemeinsame Resource (Datei für Messages).¡

- der erste Prozess, der startet erstellt ein Token
- das Token kreist weiter, wenn ein Prozess seine Schreiboperationen durchgeführt hat
- der Prozess mit dem Token darf schreiben, der Rest nicht
- kann ein Prozess nicht schreiben, kann er eine Message in einem internen Puffer ablegen und diese dann später festschreiben

7.2.2 Hilfe

Interne Message-Queues helfen. Auch lokale temporäre Dateien können als Puffer verwendet werden. Dies hat den Vorteil, dass keine Nachrichten beim schreiben verloren gehen. Wichtig: es gibt eine „Masterdatei“ in der alle Daten am Ende liegen. Als Klassen können bei *Queue* bzw. *LinkedList* verwendet werden. Es bietet sich an, beim Token ein paar Metadaten mitzugeben, wie z.B. Erstellzeit und Cluster. Ggf. ist auch der Ersteller sinnvoll, um das Token managen zu können. Damit können zwei Token, die ggf. fälschlicher Weise kreisen erkannt werden. Auch die Resource kann angegeben werden, um meherere Resourcen parallel zu synchronisieren im selben Cluster.

```
/**
 * This class is an example for a token
 * based on a message
 */
package eu.boxwork.dhbw.examples;

public class Token implements Serialisable {
  // node that created the token
  private Node creator;
  private Instant creation;
  private List<Node> cluster;
  private int resource;

  public Token()
```

```
        {}

  /* GETTERS/ SETTERS */
  public void setCreator(Node in) { this.creator = in; }
  public Node getCreator(){ return this.creator; }

  public void setCreation(Instant in) { this.creation =
      in; }
  public Node setCreation(){ return this.creation; }

  public void setCluster(Node in) { this.creator = in; }
  public Node getCluster(){ return this.creator; }

  public void setResource(int in) { this.resource = in; }
  public int getResource(){ return this.resource; }
}
```

7.3 Verteilter Algorithmus

7.3.1 Aufgabe

Implementieren Sie den verteilten Algorithmus für den Zugriff aus eine gemeinsame Resource (Datei für Messages).

- der erste Knoten, der schreiben will, schickt eine Anfrage an die anderen Knoten und schreibt erst in die Datei, wenn alle Knoten zustimmen
- bevor ein Knoten schreib, schickt er ein „ENTER“ an alle und beim Verlassen ein „EXIT“
- ein Knoten, der ein „OK“ geschickt hat, wird erst selbt den Prozess starten, wenn ein Knoten die Resource freigegeben hat
- jeder Knoten hält sich intern den Status der Resource

7.3.2 Hilfe

Dadruch, dass sich jeder Knoten den Status der Resource hält, kann entschieden werden, ob ein Eintrittsversuch möglich ist. Ist eine Resource schon blockiert muss jeder andere Prozess so lange warten. Hinweis: Es macht Sinn sich die Sttus auf der Konsole auszugeben um den Statuswechsel nachvollziehen zukönnen.

```
/**
 * This class is an example for a state class of a
    resource
 * based on a message
 */
package eu.boxwork.dhbw.examples;
```

```
public enum State  {
  OPEN, AQUIRED, OCCIPIED;
}

public class ResourceState  {
  // node that created the token
  private State state = State.OPEN;
  private int resource;

  public ResourceState()
        {}

  /* GETTERS/ SETTERS */
  public void setState(State in) { this.state = in; }
  public State getState(){ return this.state; }

  public void setResource(int in) { this.resource = in; }
  public int getResource(){ return this.resource; }
}
```

Der Status wird gewechselt beim *AQUIRE*, *ENTER* und *EXIT*.

7.3.3 Herausforderung

Innerhalb eines Clusters können Knoten jederzeit so wegfallen, dass das Cluster geteilt wird. Diskutieren sie:

1. Wie gegen wir mit gesperrten Resourcen um, wenn der Client diese nicht freigibt (weil abgestürzt)?

2. Wie bekommen wir mit, dass eine Resource geblockt ist und nicht mehr freigegeben wird?

Implementieren Sie entsprechende Strategien um ihr dediziertes Problem zu lösen (je nach Wahl der Aufgabe).