Dr. Thorsten Papenbrock
Prof. Dr. Felix Naumann
Information Systems Group
Hasso Plattner Institute

February 11, 2019

Distributed Data Management – Exam

Winter Term 2018/2019 **Musterlösung:**

Musterlösung

Matriculation Number: _____

0	1	2	3	4	5	6	7	Σ
1	10	7	6	7	13	5	21	70

Important Rules:

- The exam must be solved within **180 minutes** (09:00 – 12:00).
- Fill in your matriculation number (Matrikelnummer) above and **on every page**.
- Answers can be given in English or German.
- Any usage of external resources (such as scripts, prepared pages, electronic devices, or books) is not permitted.
- Make all **calculations transparent and reproducible**!
- Use the **free space under each task for your answers**. If you need more space, continue on the **back of the page**. Use the extra pages at the end of the exam only if necessary. **Provide a pointer to an extra page** if it should be considered for grading. The main purpose of the extra pages is for drafting.
- Please write clearly. **Do not use the color red or pencils**.
- If you have any questions, raise your hand.
- The exam consists of 22 pages including cover page and extra pages.
- For any multiple choice question, more or fewer than one answer might be correct.
- Good luck!

Task 0: Matriculation Number

Fill in your matriculation number (Matrikelnummer) on every page including the cover page and the extra pages (even if you don't use them).

Hint: Do it now! **1 point**

Task 1: Distributed Systems

1. Which of the following statements about distributed systems are true? Tick all true statements. **4 points**

   ☐ A distributed system is a group of independent compute nodes that communicate and collaborate to solve a common task.

   ☐ Distributed systems are designed for vertical scaling.

   ☐ Distributed systems can increase their reliability with the help of redundancy and replication techniques.

   ☐ According to the CAP theorem, distributed systems can support BASE but not ACID.

   Musterlösung:

   ☒ That is exactly the definition.
   ☐ They are designed for horizontal scaling. Vertical scaling = upgrade a machine to a faster one; not add more machines.
   ☒ Faults are more likely (because more types of faults exist and there are more system components that can have faults); however, fault-tolerance is more powerful in distributed systems due to redundancy.
   ☐ Many distributed systems drop ACID in favor of performance, but some do support ACID.

   Grading: 1P for each correctly ticked or non-ticked field

2. Assume that you wrote a distributed algorithm in Akka. After careful analysis of the algorithm, you know that 80% of its execution time would profit from parallelization, while 20% of it is non-parallelizable. According to Amdahl's Law, what is the maximum speedup that this algorithm can achieve on an idealized cluster that has no inherent distribution overhead and infinite size? **3 points**

   Musterlösung:

p = 8/10
s = inf
speedup = 1 / ((1-p)+p/s)
speedup = 1 / ((1-8/10)+(8/10)/inf)
speedup = 1 / (1-8/10) = 1 / (2/10) = 10/2 = 5

Grading:

- 2P correct formula
- 1P correct calculation

3. To solve some particular task, it might be necessary that a client node synchronizes its local time with a dedicated time server using the network time protocol (NTP): The client frequently sends a time message to the server to collect four timestamps: t_0, t_1, t_2, and t_3. Assume that for our client node the local clock failed completely after sending a time message. Hence, when the time message returns, t_3 has no value. Fortunately, the node monitors the round-trip delay δ, which had a very stable value of $\delta = 2$ *sec* over the last few hundred time synchronizations. Reconstruct the lost value t_3 and calculate the time shift θ between the server and the client node given the following timestamps:

   t_0 = 10:32:02, t_1 = 10:32:20, t_2 = 10:32:21 (with pattern *hh:mm:ss*) **3 points**

   Musterlösung:

   $\delta = (t_3 - t_0) - (t_2 - t_1)$
   $<=> \delta + (t_2 - t_1) + t_0 = t_3$
   $=> t_3 = 2 + (21 - 20) + 2 = 5sec$
   $=> t_3$ = 10:32:05

   $\theta = ((t1 - t0) + (t2 - t3))/2$
   $=> \theta = ((20 - 2) + (21 - 5))/2$
   $=> \theta = (18 + 16)/2 = 34/2 = 17sec$

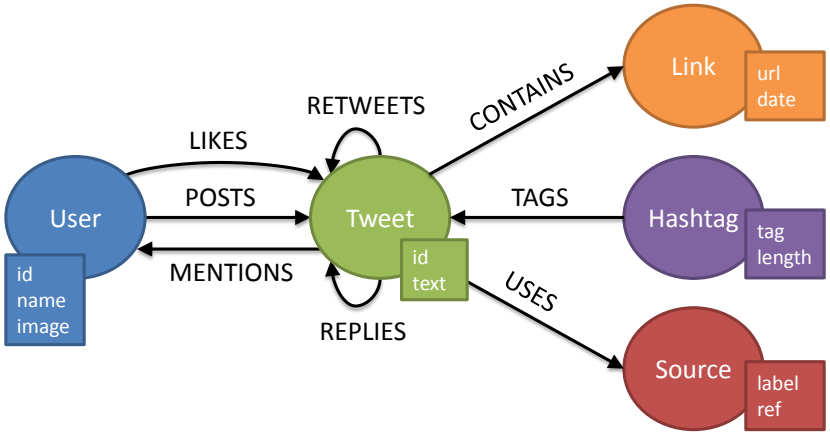
   Set time to t_3 = 10:32:05 + 17 sec

   Grading:

   - 1P correct formula or calculation strategy for δ
   - 1P correct formula or calculation strategy for θ
   - 0.5P correct calculation of t_3
   - 0.5P correct calculation of θ

Task 2: Data Models and Query Languages

The following graph describes the schema of a database storing twitter users and their tweets. Each circle describes a possible label for node instances and each edge their possible relationships to other nodes. Assume that the graph/schema is not completely shown here and further nodes and edges might exist (open world assumption). The database was created in Neo4J and you have to use Cypher to query its content.



1. Write a query that reports the ids of all `User` nodes that posted at least one `Tweet` tagged with the `Hashtag` tag "#ClimateChange". **3 points**

   Musterlösung:

   MATCH (user:User)-[:POSTS]->(:Tweet)<-[:TAGS]-(:Hashtag {tag:"#ClimateChange"})
   RETURN user.id

   Grading:

   - If bracket types are not correct or the syntax is a little bit wrong, we do not mark that as an error
   - 0.5P MATCHing User
   - 0.5P MATCHing Tweet
   - 0.5P MATCHing Hashtag
   - 0.5P RETURNing user.id
   - 0.5P arrows follow the join path correctly
   - 0.5P correct edge and node labels

2. We are interested in replies to `Tweet`s that talk about Michael Stonebraker's project *Tamr*. A `Tweet` talks about this project, if it mentions the `User` "Michael Stonebraker" and contains a `Link` to the url "www.tamr.com". Write a query that returns all `Tweet` nodes that either directly or indirectly (via up to 5 replies) reply to such a project tweet. **4 points**

   Musterlösung:

   MATCH (:User {name:"Michael Stonebraker"})<-[:MENTIONS]-(tweet:Tweet)-[:CONTAINS]->(:Link {url:"www.tamr.com"})
   MATCH (reply:Tweet)-[:REPLIES*1..5]->(tweet)
   RETURN reply

   Grading:

   - If bracket types are not correct or the syntax is a little bit wrong, we do not mark that as an error
   - 1P MATCHing tweet to user Michael Stonebraker
   - 1P MATCHing tweet to link www.tamr.com
   - 0.5P MATCHing tweet to reply
   - 1P *1..5 reply hops
   - 0.5P RETURNing reply

Task 3: Replication and Partitioning

1. Assume you have a cluster of 800 nodes. The cluster runs a leaderless replicated, distributed database. Quorum reads and writes are used to ensure consistency and a gossip protocol ensures that all updates will eventually spread to all nodes in the cluster.

   Which read (r) and write (w) values do we need to define in our quorum q(r,w) if writes should return as fast as possible and every successfully written value should reach all cluster nodes in expectedly not more than five rounds of gossip? **4 points**

   Musterlösung:

   with 0 gossip rounds: 800
   with 1 gossip rounds: 800 / 2 = 400
   with 2 gossip rounds: 800 / 2 / 2 = 200
   with 3 gossip rounds: 800 / 2 / 2 / 2 = 100
   with 4 gossip rounds: 800 / 2 / 2 / 2 / 2 = 50
   with 5 gossip rounds: 800 / 2 / 2 / 2 / 2 / 2 = 25

   in general:
   $w * 2^{rounds} >= 800$
   $w >= 800/2^{rounds} = 800/2^5 = 800/32 = 25$

   consistency: r + w > n
   r > n - w = 800 - 25 = 775

   answer: q(776,25)

   Grading:

   - 1P w = 25 correctly calculated
   - 1P r = 776 correctly calculated
   - 1P gossip protocol correctly understood
   - 1P quorum consistency condition r + w > n

2. Assume you have a dataset that is stored on a cluster with 3 nodes. For partitioning, the space of key hashes is split as follows:

   Now a new node N_4 enters the cluster and wants to get its share of the dataset. Add node N_4 to the partitioned key space of Figure 1 by drawing it into the image. Use *consistent hashing* and the *fixed number of partitions per node* strategy. **2 points**

   Musterlösung:

   Grading:

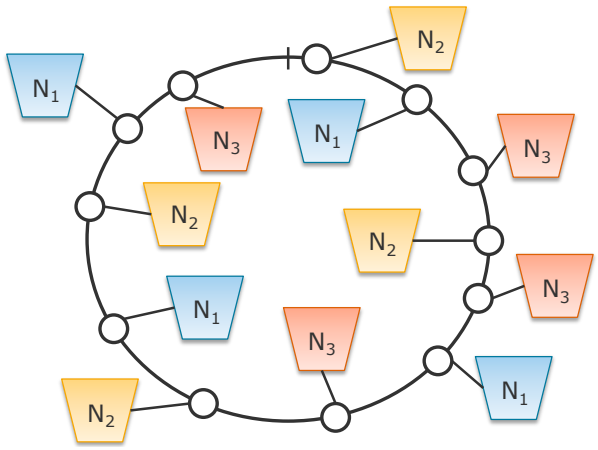
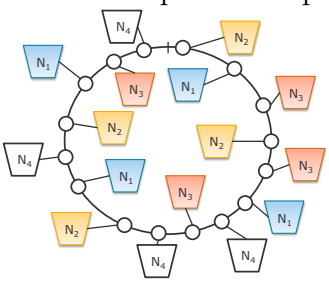


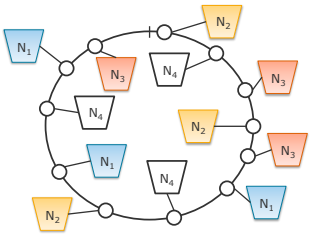
Abbildung 1: A partitioned space of key hashes

Fixed number of partitions per node



- 4 new hash points
- halving previous partitions

Fixed number of partitions overall



- 3 partitions taken over by N4
- one from N1, N2, N3 respectively

- -1P if more or less than 4 new nodes labeled N_4 are placed on the ring
- -0.5P if N_4 does not halve the partitions it steals from
- -1P if existing nodes are moved or removed
- not less than 0 points

Task 4: Failure Detection

The ϕ accrual failure detector method calculates a suspicion level ϕ for monitored processes from their heartbeat history. One important part of that calculation is the formula $P_{later}(t) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \int_t^{+\infty} e^{-\frac{(x-\mu)^2}{2 \cdot \sigma^2}} dx$.

1. What does the formula for $P_{later}(t)$ calculate, i.e., what does it mean? **1 points**

   Musterlösung: -> The probability that a heartbeat will arrive more than t time units after the previous one

   0.5P for probability
   0.5P heartbeat after the previous one

2. Consider the probability density function $f(t) = \frac{1}{\sigma \cdot \sqrt{2 \cdot \pi}} \cdot e^{-\frac{(t-\mu)^2}{2 \cdot \sigma^2}}$ of heartbeat arrival times that is used to calculate $P_{later}(t)$. Figure 2 depicts the graph of $f(t)$ for some mean μ and variance σ of heartbeat arrival times. How does $f(t)$ change if the variance σ of the heartbeats increases due to higher network traffic? Draw one curve with higher variance σ into Figure 2 and describe the change in a few words. **2 points**

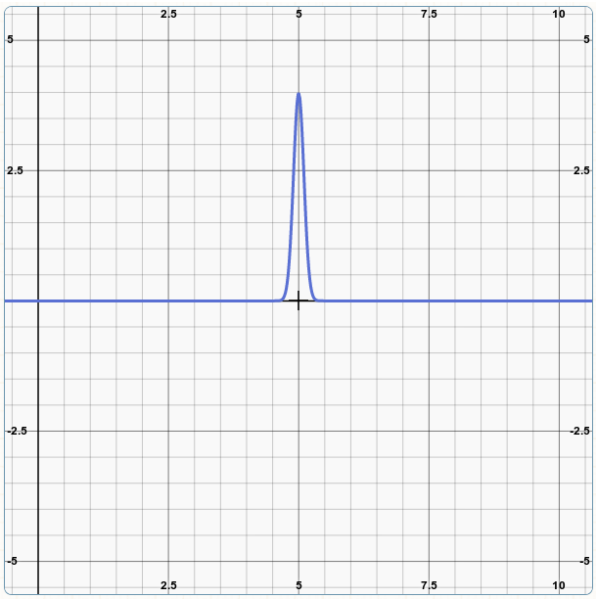


Abbildung 2: The probability density function $f(t)$

Musterlösung: -> the graph gets flatter and more bellied; the area under the curve stays the same

1P "flatter"
1P "more bellied" or "area stays the same"
curve needs to show the same; the visualization should show these two aspects, i.e., if the curve gets flatter but the area definitely gets smaller, only half of the points can be given

3. How can we turn the monotonically decreasing $P_{later}(t)$ into the suspicion level $\phi(t_{now})$, given that T_{last} denotes the arrival time of the last heartbeat and t_{now} the current time? Write down the formula. **2 points**

   $\phi(t_{now}) =$

   Musterlösung: -> $\phi(t_{now}) = -log_{10}(P_{later}(t_{now} - T_{last}))$

   1P for *minus some logarithm*
   1P for P_{later} of $t_{now} - T_{last}$

4. The suspicion level ϕ needs to fulfill four important properties for being used by a failure detector. Name two of them. **2 points**

   Musterlösung:

   1. **asymptotic completeness** or *if the monitored process is faulty, ϕ becomes infinitely large*
   2. **eventual monotony** or *if the monitored process is faulty, ϕ is monotonically increasing*
   3. **upper bound** or *if the monitored process is correct, ϕ has an upper bound*
   4. **reset** or *if the monitored process is correct, ϕ always resets to 0 at some future point in time*

   0.5P for each correct point

Task 5: Batch Processing

Suppose you are given three datasets: A `students` dataset that contains general information about students, a `courses` dataset that describes available courses for the students, and an `enrollment` dataset, which is basically a join table between `students` and their `courses`. The following code snippets read the three datasets into Spark `Datasets`:

```
val students = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/students.csv")
  .toDF("ID", "Name", "Semester", "Supervisor")
  .as[(String, String, String, String)]

val enrollments = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/enrollments.csv")
  .toDF("StudentID", "CourseID", "Credits")
  .as[(String, String, String)]

val courses = spark
  .read
  .option("quote", "\"")
  .option("delimiter", ",")
  .csv(s"data/courses.csv")
  .toDF("ID", "Title", "Teacher", "Topic")
  .as[(String, String, String, String)]
```

Use the three `Datasets` to solve the following tasks. You may use Spark's *Dataset* and/or *DataFrame* API but no SQL! Also have a look at the *Dataset* API documentation at the end of this task. If you are not sure about how a particular interface, call, or class works, make a good guess and provide a comment on how you *think* it works.

1. Write a Spark transformation pipeline that starts with the `students Dataset`, then filters all those students that are supervised by "Prof. Dumbledore", maps these students to their `ID` and `Semester`, and finally displays the results in tabular form on the standard output. **3 points**

   Musterlösung:

   ```
   students
     .filter(s => s._4 == "Prof. Plattner")
     .map(s => (s._1, s._3))
     .show()
   ```

   Grading:

   - 1P filter
   - 1P map
   - 1P show
   - Because "filter all those students that are supervised by Prof. Dumbledore" is ambiguous, both checks == and != are OK for the filter.

2. Translate the following SQL query into a Spark transformation pipeline. **5 points**

   ```
   SELECT *
   FROM {
       SELECT DISTINCT Title
       FROM courses
       WHERE Teacher = "Prof. Snape"
     } INTERSECT {
       SELECT DISTINCT Title
       FROM courses
       WHERE Teacher = "Prof. Moody"
     }
   ORDER BY Title
   ```

   Musterlösung:

   Grading:

   - 2 * 0.5P filter
   - 2 * 0.5P map
   - 2 * 0.5P distinct
   - 1P intersect

```
val coursesSnape = courses
  .filter(c => c._3.equals("Prof. Snape"))
  .map(c => c._2)
  .distinct()
val coursesMoody = courses
  .filter(c => c._3.equals("Prof. Moody"))
  .map(c => c._2)
  .distinct()
coursesSnape.intersect(coursesMoody).sort().show()
```

- 1P sort
- action does not matter here, because the SQL query also does not imply an action
- alternative solution: groupBy(Title) + mapGroups(those that contain Snape and Moody)

3. Implement a join between the `students` and the `enrollments` dataset as a Spark batch job without using Spark's `joinWith()` transformation. The result should be a `Dataset` with two columns: one column containing the student record and one column containing the enrollment record. Join on `students.ID = enrollments.StudentID` and finalize the pipeline with some action of your choice. Do *not* implement the join as a broadcast join or any map-side join, because both relations are large. **5 points**

   Hint: Remember how reduce-side joins work in the MapReduce framework.

   Musterlösung:

```
val sMapped = students.map(t => (t._1, 1, Array(t._1, t._2, t._3, t._4)))
val eMapped = enrollments.map(t => (t._1, 2, Array(t._1, t._2, t._3)))

sMapped
  .union(eMapped)
  .groupByKey(t => t._1)
  .mapGroups((key, iterator) => {
      val group = iterator.toList

      val side1 = group.filter(t => t._2 == 1).map(t => t._3)
      val side2 = group.filter(t => t._2 == 2).map(t => t._3)

      side1.flatMap(t => side2.map(s => (t, s)))
  })
  .flatMap(t => t)
  .show()
```

   Grading:

   - 1P map to key
   - 1P union
   - 1P groupBy

- 1P mapGroups and crossproduct in each group
- 0.5P flatMap after grouping
- 0.5P some final action (show, collect, foreach, ...)
- Geänderte Bewertung: 2P union and map to same schema does not matter, because one could assume that Tuple4 and Tuple3 have the same super class Tuple (which they do not, but the assumption is ok) and then simply union Tuples would work. One can then figure out if a tuple is from students or enrollments by the size of the tuple and select the join attribute accordingly.

Typed transformations

def **as**(alias: String): Dataset[T]
Returns a new Dataset with an alias set.

def **distinct**(): Dataset[T]
Returns a new Dataset that contains only the unique rows from this Dataset.

def **except**(other: Dataset[T]): Dataset[T]
Returns a new Dataset containing rows in this Dataset but not in another Dataset.

def **filter**(func: FilterFunction[T]): Dataset[T]
(Java-specific) Returns a new Dataset that only contains elements where `func` returns `true`.

def **filter**(func: (T) ⇒ Boolean): Dataset[T]
(Scala-specific) Returns a new Dataset that only contains elements where `func` returns `true`.

def **flatMap**[U](f: FlatMapFunction[T, U], encoder: Encoder[U]): Dataset[U]
(Java-specific) Returns a new Dataset by first applying a function to all elements of this Dataset, and then flattening the results.

def **flatMap**[U](func: (T) ⇒ TraversableOnce[U])(*implicit* arg0: Encoder[U]): Dataset[U]
(Scala-specific) Returns a new Dataset by first applying a function to all elements of this Dataset, and then flattening the results.

def **groupByKey**[K](func: MapFunction[T, K], encoder: Encoder[K]): KeyValueGroupedDataset[K, T]
(Java-specific) Returns a KeyValueGroupedDataset where the data is grouped by the given key `func`.

def **groupByKey**[K](func: (T) ⇒ K)(*implicit* arg0: Encoder[K]): KeyValueGroupedDataset[K, T]
(Scala-specific) Returns a KeyValueGroupedDataset where the data is grouped by the given key `func`.

def **intersect**(other: Dataset[T]): Dataset[T]
Returns a new Dataset containing rows only in both this Dataset and another Dataset.

def **joinWith**[U](other: Dataset[U], condition: Column): Dataset[(T, U)]
Using inner equi-join to join this Dataset returning a `Tuple2` for each pair where `condition` evaluates to true.

def **map**[U](func: MapFunction[T, U], encoder: Encoder[U]): Dataset[U]
(Java-specific) Returns a new Dataset that contains the result of applying `func` to each element.

def **map**[U](func: (T) ⇒ U)(*implicit* arg0: Encoder[U]): Dataset[U]
(Scala-specific) Returns a new Dataset that contains the result of applying `func` to each element.

def **sort**(sortExprs: Column*): Dataset[T]
Returns a new Dataset sorted by the given expressions.

def **sort**(sortCol: String, sortCols: String*): Dataset[T]
Returns a new Dataset sorted by the specified column, all in ascending order.

def **union**(other: Dataset[T]): Dataset[T]
Returns a new Dataset containing union of rows in this Dataset and another Dataset.

Untyped transformations

def **col**(colName: String): Column
Selects column based on the column name and return it as a Column.

Actions

def **collect**(): Array[T]
Returns an array that contains all rows in this Dataset.

def **show**(numRows: Int, truncate: Int): Unit
Displays the Dataset in a tabular form.

def **foreach**(f: (T) ⇒ Unit): Unit
Applies a function `f` to all rows.

def **reduce**(func: ReduceFunction[T]): T
(Java-specific) Reduces the elements of this Dataset using the specified binary function.

def **reduce**(func: (T, T) ⇒ T): T
(Scala-specific) Reduces the elements of this Dataset using the specified binary function.

class **KeyValueGroupedDataset**[K, V] extends Serializable

def **mapGroups**[U](f: MapGroupsFunction[K, V, U], encoder: Encoder[U]): Dataset[U]
(Java-specific) Applies the given function to each group of data.

def **mapGroups**[U](f: (K, Iterator[V]) ⇒ U)(*implicit* arg0: Encoder[U]): Dataset[U]
(Scala-specific) Applies the given function to each group of data.

Task 6: Stream Processing

Consider the following stream analytics program written in Flink:

```
import org.apache.flink.api.java.tuple.*;
import org.apache.flink.streaming.api. *;
import org.apache.flink.util.Collector;
import java.util.Date;

public class SensorProcessing {

  public static void main(String[] args) throws Exception {
    StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

    DataStream<Tuple3<Integer, Date, Float>> readings = env
      .socketTextStream("localhost", 9999)
      .map(SensorProcessing::lineToReading)
      .assignTimestampsAndWatermarks(new AscendingTimestampExtractor<Tuple3<Integer, Date, Float>>() {
          @Override
          public long extractAscendingTimestamp(Tuple3<Integer, Date, Float> element) {
            return element.f1.getTime();
          }
        });

    readings
      .filter(r -> (r.f2 > -50.0f) && (r.f2 < 50.0f))
      .keyBy(r -> r.f0)
      .timeWindow(Time.days(1), Time.days(1))
      .apply(new StandingQuery())
      .print();

    env.execute("Sensor Processing");
  }

  public static class StandingQuery implements WindowFunction
      <Tuple3<Integer, Date, Float>, Tuple2<Integer, Date>, Integer, TimeWindow> {

    @Override
    public void apply(Integer key, TimeWindow window,
                Iterable<Tuple3<Integer, Date, Float>> values,
                Collector<Tuple2<Integer, Date>> out) {

      for (Tuple3<Integer, Date, Float> reading : values) {
        if (reading.f2 > 30.0f) {
          out.collect(new Tuple2<>(key, reading.f1));
          return;
        }
      }
    }
  }

  // Reading(SensorID, Timestamp, Temperature)
  private static Tuple3<Integer, Date, Float> lineToReading(String line) {...}
}
```

1. What does the program calculate? Describe what events the `StandingQuery` outputs and what these events mean. **2 points**

   Musterlösung: Grading:

   - 1P What: Pairs of sensor ID and timestamp
   - 1P Mean: The sensor measured a temperature higher than 30 degrees (at least once) on the day of that timestamp. In other words: It is the first time this sensor measured a temperature higher than 30 degrees on that day.

2. The input stream for the Flink program might start like this:

`SensorID`	`Timestamp`	`Temperature`
#	*dd-mm-yy hh:mm:ss*	$°C$
1	21-05-18 00:00:00	12.1
2	21-05-18 00:00:00	28.0
3	21-05-18 00:00:00	15.4
1	21-05-18 06:00:00	12.3
2	21-05-18 06:00:00	29.8
3	21-05-18 06:00:00	14.7
1	21-05-18 12:00:00	12.7
2	21-05-18 12:00:00	31.1
3	21-05-18 12:00:00	89.9
1	21-05-18 18:00:00	13.0
2	21-05-18 18:00:00	32.9
3	21-05-18 18:00:00	14.6
1	22-05-18 00:00:00	11.3
2	22-05-18 00:00:00	27.1
3	22-05-18 00:00:00	15.6
...	...	...

   Write down the output of the program after processing all depicted events. **2 points**

   Musterlösung: (2, 21-05-18 12:00:00)

   Grading:

   - -1P for outputting any (1, XX-XX-XX XX:XX:XX), because it's temperature is never >30
   - -1P for outputting (2, 21-05-18 18:00:00), because the standing query returns after its first output
   - -1P for outputting (3, 21-05-18 12:00:00), because its 89.9 temperature is a false reading that is filtered out before

3. Given the start of the stream as depicted above. What is the degree of parallelism for the standing query, i.e., how many windows are processed in parallel on that stream? **1 points**

   Musterlösung: 3 (grouping by sensor ID creates 3 windows, one for each sensor in the stream; the windows do not overlap)

Task 7: Actor Programming

1. The Actor model implementation in Akka makes two guarantees for the messaging (when used in conjunction with the TCP protocol). What are these two guarantees? **2 points**

   Musterlösung:

   1P at-most-once delivery (or exactly-once delivery due to TCP)
   1P sender-receiver ordering (0.5P ordering, 0.5P for sender-receiver pair)

2. Which of the following statements on the actor model and actor programming are *true*? Tick the true ones. **6 points**

   ☐ An `ActorSystem` in Akka is a named hierarchy of actors that also includes components for event streaming, thread dispatching, and message remoting.

   ☐ Actors are created as passive entities that become active only if they receive messages; after processing a message, they return to their passive state.

   ☐ Actors can open dedicated mailboxes, which serve as site-channels for large messages such that these large messages do not block the communication that happens on the primary mailbox.

   ☐ For parallel message processing, an `ActorSystem` can serve one actor instance with two threads: one thread processes the first message from the queue while the second thread processes the second message.

   ☐ Actor messages usually carry the sender of a message allowing the receiver to respond to that message; for this reason, actors need to be serializable, i.e., implement Java's `Serializable` interface.

   ☐ Blocking ask messages are not supported by the general actor model, but they can be implemented using Java's `Futures` and the *ask pattern*.

   Musterlösung:

   ☒
   ☒
   ☐ The actor model says that there should be only one mailbox for each actor; furthermore, large messages do not block mailboxes – they block the ActorSystems remoting component and its network traffic.
   ☐ No parallelization within one actor! The ActorSystem will never serve the same actor instance with two threads!
   ☐ Actor messages carry an ActorRef (which needs to be serializable) and not the actual actor; hence, actors don't need to be serializable.

☒

Grading: 0.5P for each correctly ticked or non-ticked field

3. Consider the following actor implementation. What fundamental problem does this implementation have and to what consequences might this problem lead? **2 points**

```
public class BadActor extends AbstractLoggingActor {

    private Map<String, Integer> histogram = new HashMap<>();

    public Receive createReceive() {
       return receiveBuilder()
          .match(String.class, s -> sender().tell(histogram, self()))
          .build();
    }
    ...
}
```

Musterlösung:

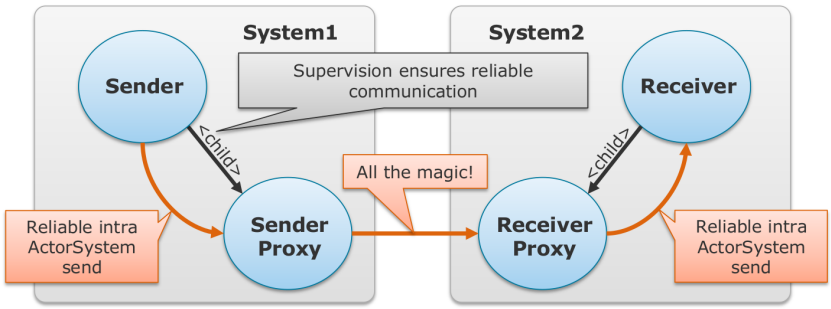
Problem: In response to a String message, the actor sends a reference to its private state (a reference, not a copy), which is then manipulable from the outside!

Consequences: Concurrent modifications of the same data structure; invalid/inconsistent values in the histogram; race conditions

Grading:

- 1P Problem
- 1P Consequence
- if the problem was not seen, 1P for mentioning another, reasonable issue with this implementation (e.g. HashMaps are often large and their data might exceed the maximum message size)

4. The *Reliable Communication Proxy* pattern ensures that all messages that a `Sender` actor tells to a `Receiver` actor are transmitted exactly once and in order. From a conceptual view, the pattern looks as follows:



Implement the two proxies of the *Reliable Communication Proxy* pattern in Akka given the actor templates below. The message that the `Sender` actor wants to tell to the `Receiver` actor is a simple `String`. The `Receiver` should receive this `String` message as if the `Sender` would have told it directly to it. **11 points**

Hint: Use the space on this page for drafting. Then write your solution into the provided gaps. Use the back of the template pages if you need more space.

Musterlösung:

Grading: Do not be strict about the syntax: missing ; or small mistakes are ok

Sender:
0.5P sequenceNumber
0.5P map for pending acknowledgments
1P receive some acknowledgment message
1P create ReliableMessage wrapped in a Cancellable
0.5P increase sequenceNumber
0.5P add to pending acknowledgments
1P handle acknowledgment message by removing and canceling it

Receiver:
0.5P sequenceNumber for lastly forwarded message
0.5P map for waiting messages
1P ReliableMessage correct
1P send acknowledgement
0.5P ignore duplicate messages
0.5P hold message
1P forward message
1P forward also the waiting messages

If the receiverProxy acknowledges every ReliableMessage that it gets, then the Acknowledgment is also a guaranteed to arrive at the senderProxy eventually. Hence, we do not need to care about making the Acknowledgment reliable as well.

Some solutions do the following:
- senderProxy creates a Cancellable for every String message
- receiverProxy waits for the next sequence number and ignores all other messages; if the right message arrives, this one is acknowledged and we wait for the next message
=> exactly-once messaging and ordered-delivery is guaranteed, but under constant message fire; the Cancellable together with the Network are misused as kind of active buffer – not a smart idea; in particular, this is not how the Reliable Proxy Pattern works; therefore: -2P (-0.5 for missing waiting messages queue and -1.5 for not holding and forwarding messages)

Some solutions do the following:
- senderProxy creates one Cancellable for the next String message to be send; it buffers all further Strings in correct order and creates the next Cancellable when the current is acknowledged
- receiverProxy waits for the next sequence number if the right message arrives, this one is acknowledged and we wait for the next message
=> exactly-once messaging and ordered-delivery is guaranteed, but this is also not how the Reliable Proxy Pattern works, because the receiver should ensure the message ordering; the pattern-conform messaging is faster, because ReliableMessages are fired faster when they are not bound to a strict send-ack-send-ack-... schedule; therefore: -1P (-0.5 for missing waiting messages queue in receiverProxy and -0.5 for a too conservative waiting system)

Musterlösung:

Musterlösung:

```java
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;

import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Cancellable;
import akka.actor.Props;
import de.hpi.octopus.ReceiverProxy.ReliableMessage;
import scala.concurrent.duration.Duration;

public class SenderProxy extends AbstractLoggingActor {

    private ActorRef receiverProxy;
    [redacted]

    public SenderProxy(ActorRef receiverProxy) {
        this.receiverProxy = receiverProxy;
    }

    public static Props props(ActorRef receiverProxy) {
        return Props.create(SenderProxy.class, () -> new SenderProxy(receiverProxy));
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(String.class, this::handle)
                [redacted]
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    private void handle(String message) {
        [redacted]
    }

    private void handle([redacted] message) {
        [redacted]
    }

    private Cancellable createCancellableFor(ReliableMessage message) {
        return this.getContext().system().scheduler().schedule(
                Duration.create(0, TimeUnit.SECONDS),
                Duration.create(3, TimeUnit.SECONDS),
                this.receiverProxy,
                message,
                this.getContext().dispatcher(),
                this.self());
    }

    private void stopCancellable(Cancellable cancellable) {
        cancellable.cancel();
    }
}
```

```java
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;

import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Cancellable;
import akka.actor.Props;
import de.hpi.octopus.ReceiverProxy.ReliableMessage;
import scala.concurrent.duration.Duration;

public class SenderProxy extends AbstractLoggingActor {

    private ActorRef receiverProxy;

    private int sequenceNumber = 0;
    private Map<Integer, Cancellable> pendingAcks = new HashMap<>();

    public SenderProxy(ActorRef receiverProxy) {
        this.receiverProxy = receiverProxy;
    }

    public static Props props(ActorRef receiverProxy) {
        return Props.create(SenderProxy.class, () -> new SenderProxy(receiverProxy));
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(String.class, this::handle)
                .match(Integer.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    private void handle(String message) {
        Cancellable cancellable = this.createCancellableFor(
                new ReliableMessage(this.sequenceNumber, message, this.sender()));
        this.pendingAcks.put(this.sequenceNumber, cancellable);
        this.sequenceNumber++;
    }

    private void handle(Integer message) {
        Cancellable cancellable = this.pendingAcks.remove(message);
        this.stopCancellable(cancellable);
    }

    private Cancellable createCancellableFor(ReliableMessage message) {
        return this.getContext().system().scheduler().schedule(
                Duration.create(0, TimeUnit.SECONDS),
                Duration.create(3, TimeUnit.SECONDS),
                this.receiverProxy,
                message,
                this.getContext().dispatcher(),
                this.self());
    }

    private void stopCancellable(Cancellable cancellable) {
        cancellable.cancel();
    }
}
```

```java
import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;

import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Props;

public class ReceiverProxy extends AbstractLoggingActor {

    private ActorRef receiver;

    public ReceiverProxy(ActorRef receiver) {
        this.receiver = receiver;
    }

    public static Props props(ActorRef receiver) {
        return Props.create(ReceiverProxy.class, () -> new ReceiverProxy(receiver));
    }

    public static class ReliableMessage implements Serializable {
        private static final long serialVersionUID = -3254147511955012292L;

    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(ReliableMessage.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    private void handle(ReliableMessage message) {

    }
}
```

```java
import java.io.Serializable;
import java.util.HashMap;
import java.util.Map;

import akka.actor.AbstractLoggingActor;
import akka.actor.ActorRef;
import akka.actor.Props;

public class ReceiverProxy extends AbstractLoggingActor {

    private ActorRef receiver;

    private int sequenceNumber = -1;
    private Map<Integer, String> waitingMessages = new HashMap<>();

    public ReceiverProxy(ActorRef receiver) {
        this.receiver = receiver;
    }

    public static Props props(ActorRef receiver) {
        return Props.create(ReceiverProxy.class, () -> new ReceiverProxy(receiver));
    }

    public static class ReliableMessage implements Serializable {
        private static final long serialVersionUID = -3254147511955012292L;
        public ReliableMessage() {}
        public ReliableMessage(int sequenceNumber, String message, ActorRef sender) {
            this.sequenceNumber = sequenceNumber;
            this.message = message;
            this.sender = sender;
        }
        public int sequenceNumber;
        public String message;
        public ActorRef sender;
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder()
                .match(ReliableMessage.class, this::handle)
                .matchAny(object -> this.log().info("Unknown message: \"{}\"", object.toString()))
                .build();
    }

    private void handle(ReliableMessage message) {
        // Send acknowledgement
        this.sender().tell(message.sequenceNumber, this.self());

        // Ignore message, if it is a duplicate
        if (message.sequenceNumber <= this.sequenceNumber || this.waitingMessages.containsKey(message.sequenceNumber)) {
            return;
        }

        // Hold message, if a previous message is still pending
        if (this.sequenceNumber + 1 > message.sequenceNumber) {
            this.waitingMessages.put(message.sequenceNumber, message.message);
            return;
        }

        // Forward message
        this.sequenceNumber++;
        this.receiver.tell(message.message, message.sender);

        // Forward all messages that were waiting for this message
        while (this.waitingMessages.containsKey(this.sequenceNumber + 1)) {
            this.sequenceNumber++;
            String waitingMessage = this.waitingMessages.remove(this.sequenceNumber);
            this.receiver.tell(waitingMessage, message.sender);
        }
    }
}
```

Extra page 1

Extra page 2

Extra page 3