# Group 43 — Report for Assignment 4

Rodrigues, Diogo
MTK 03770446
diogo.rodrigues@tum.de

Silva, Tiago
MTK 03770544
tiago.silva@tum.de

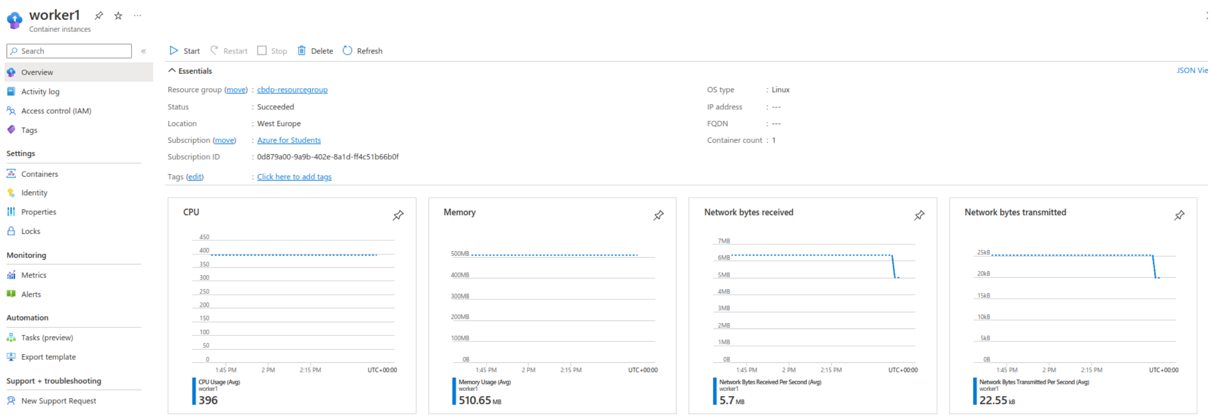
Martins, João Lucas
MTK 03770536
joao.martins@tum.de

TUM – Cloud-Based Data Processing 2022/23
21st December 2022

## Part 1: Azure Deployment and Testing

1. After you have finished with the Azure tutorial, measure the time it takes for the Assignment 3 query to run on Azure. What do you notice?
   **A:** Assignment 3 was deployed and measured with three workers, three times, yielding the following execution times: $28\,103$ ms, $27\,747$ ms, and $30\,070$ ms. These values are similar to the results obtained in the previous assignment.

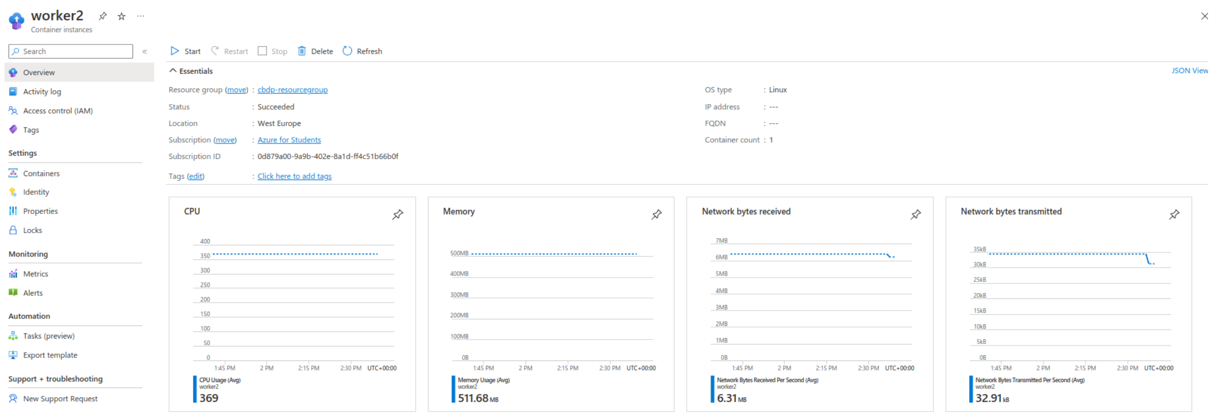
2. Go to the Azure monitoring panel for your containers: explain what is the bottleneck that increases query execution time. Include screenshots if needed.



Resource group (move) : cbdp-resourcegroup
Status : Succeeded
Location : West Europe
Subscription (move) : Azure for Students
Subscription ID : 0d879a00-9a9b-402e-8a1d-ff4c51b66b0f
Tags (edit) : Click here to add tags
OS type : Linux
IP address : ---
FQDN : ---
Container count : 1

CPU Usage (Avg) worker1 396
Memory Usage (Avg) worker1 510.65 MB
Network Bytes Received Per Second (Avg) worker1 5.7 MB
Network Bytes Transmitted Per Second (Avg) worker1 22.55 kB

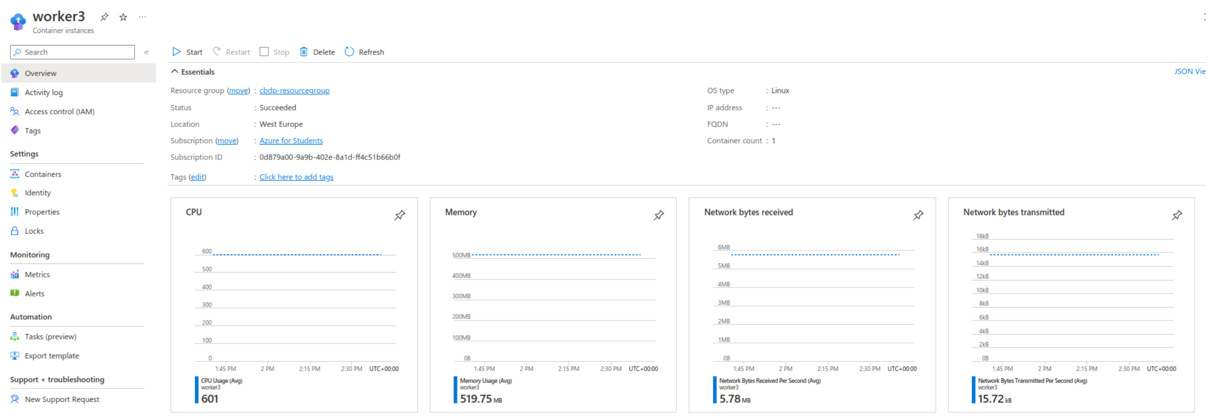
Figure 1: Worker 1 Monitoring Panel



Resource group (move) : cbdp-resourcegroup
Status : Succeeded
Location : West Europe
Subscription (move) : Azure for Students
Subscription ID : 0d879a00-9a9b-402e-8a1d-ff4c51b66b0f
Tags (edit) : Click here to add tags
OS type : Linux
IP address : ---
FQDN : ---
Container count : 1

CPU Usage (Avg) worker2 369
Memory Usage (Avg) worker2 511.68 MB
Network Bytes Received Per Second (Avg) worker2 6.31 MB
Network Bytes Transmitted Per Second (Avg) worker2 32.91 kB

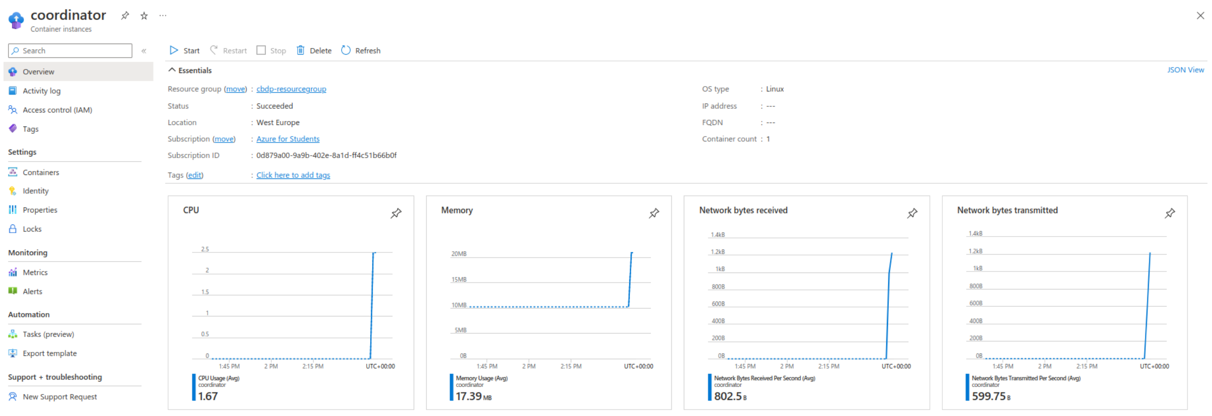
Figure 2: Worker 2 Monitoring Panel



Essentials

Resource group (move) : cbdp-resourcegroup
Status : Succeeded
Location : West Europe
Subscription (move) : Azure for Students
Subscription ID : 0d879a00-9a9b-402e-8a1d-ff4c51b66b0f
Tags (edit) : Click here to add tags
OS type : Linux
IP address : ---
FQDN : ---
Container count : 1

Figure 3: Worker 3 Monitoring Panel



Essentials

Resource group (move) : cbdp-resourcegroup
Status : Succeeded
Location : West Europe
Subscription (move) : Azure for Students
Subscription ID : 0d879a00-9a9b-402e-8a1d-ff4c51b66b0f
Tags (edit) : Click here to add tags
OS type : Linux
IP address : ---
FQDN : ---
Container count : 1

Figure 4: Coordinator Monitoring Panel

**A:** Figures 1, 2, and 3 contain worker monitoring panels during one of the conducted tests. Figure 4 represents the same statistics, but for the coordinator node. These figures showcase that the load on all worker nodes is superior to the load on the coordinator instance. Furthermore, worker statistics show that CPU, memory and network usage remain constant throughout the job execution, with a particular high incoming network usage, around $48\,\text{Mbit/s}$. The group believes this resource is the bottleneck of the system, which is expected when considering the network-heavy nature of the job.

3. Let's get faster: Pre-upload the data partitions and fileList inside Azure blob storage. Adapt your solution to read from there. What is the speedup you observe? How is it explained?
   **A:** Contrary to expectations, there was no speedup when data partitions and fileList is stored inside blob storage. Three different tests were conducted with three workers, resulting in the following execution times: $29\,011$ ms, $36\,707$ ms, and $27\,562$ ms. The group was expecting that, due to the network being the limiting factor in the solution, these results would be faster.

## Part 2: Managing Shared State - Design Questions

1. Give a brief description of your solution.
   **A:** In our solution, we start the coordinator first, and then the workers try to connect to the coordinator. Although the coordinator is already running, it is not accepting connections yet, because we want processing to start when we manually trigger it, so we can appropriately measure processing time when all workers are ready to receive work. When all workers are up and trying to connect to the coordinator, we start accepting connections and processing by manually notifying the coordinator via a pipe.
   We start by sending the workers `SPLIT` commands. A `SPLIT` command tells a worker to grab a partition $p$, count the domains in that partition, and then partition domains by hash into $S$ subpartitions ($S$ defaults to 3).
   When all subpartitions are done, the coordinator starts sending `MERGE` commands. A `MERGE` command instructs a worker to merge a list of subpartition with countings, to get a single partial result with the 25 most popular domains in that partition (according to our hash-based partitioning of domains).
   Finally, when all partial results are ready, the coordinator aggregates them.

2. What was the query execution time in Azure? Include screenshots, logs if needed.
   **A:** The measured query execution time for three workers and three subpartitions has an average of 151303.

3. Which partitioning method did you choose for the calculation of the partial aggregates? Why?
   **A:** We chose hash partitioning. So, to decide what partition a domain $d$ belongs to, we hash it $h(d)$ and then obtain the partition by using the modulus $h(d) \mod S$. So the partial result that domain $d$ will end up is `h(d) % S`.

4. What number of subpartitions did you choose, and why? If you choose to create 100 subpartitions for each partition, is it a good choice? Is there a scalability limit?
   **A:**

   The bottleneck is probably the amount of data being read/written to the blob container.

   Let us use the following notation:

   - $P = 100$ is the number of partitions that our data is originally partitioned into. We assume we cannot change this number, as we consider the original data partitions to be immutable.
   - $S$ is the number of subpartitions that each partition is split into.
   - $W \approx 3$ is the number of workers that we can spawn, and that can perform work in parallel.
   - $\overline{d} \approx 15\,000$ is the approximate number of distinct domains in the whole dataset.
   - $\overline{p} = 100\,000$ is the number of URLs that each of the $P$ partitions contains.

   The execution times for each phase are:

   - $\lceil P/W \rceil * (\overline{p} + \overline{d} + S)$ for the splitting phase; since each worker will process at most $\lceil P/W \rceil$ split requests, and the time a worker takes to process one split request is $\overline{p} + \overline{d} + S$. This is because the worker has to read one partition of size $|p|$, then create $S$ blobs and then write about $\overline{d}/S$ lines to each of the $S$ blobs, which amounts to writing about $\overline{d}$ lines to subpartitions.
   - $\lceil S/W \rceil * (P * \overline{d}/S)$ for the partial result merging; since each worker will process at most $\lceil S/W \rceil$ merge requests, and the time a worker takes to process one merge request is $P * \overline{d}/S$. This is because the worker has to read $P$ subpartitions that all refer to the same set of $|\overline{d}/S$ domains, and from each of those $P$ files it has to read about $\overline{d}/S$ lines.
   - $25 * S$ for the final aggregation phase at the coordinator, since the coordinator has to read 25 lines from each of the $S$ partial results.

   Therefore, the total execution time is

   $$\left\lceil \frac{P}{W} \right\rceil (\overline{p} + \overline{d} + S) + \left\lceil \frac{S}{W} \right\rceil (P * \overline{d}/S) + 25 * S$$

   If we assume $P, S \gg W$, then we have the execution time

   $$\begin{aligned} \frac{P}{W}(\overline{p} + \overline{d} + S) + \frac{S}{W}(P * \overline{d}/S) + 25 * S &= \frac{P(\overline{p} + \overline{d} + S)}{W} + \frac{S(P * \frac{\overline{d}}{S})}{W} + 25S \\ &= \frac{P(\overline{p} + \overline{d} + S)}{W} + \frac{P * \overline{d}}{W} + 25S \\ &= \frac{P}{W}(\overline{p} + 2\overline{d} + S) + 25S \end{aligned}$$

   So if $S$ is already pretty big, if we increase any further then performance will become worse.

   If we assume that $S \approx W$ or that $S < W$, then $\lceil S/W \rceil = 1$ and we get the following execution time:

   $$\left\lceil \frac{P}{W} \right\rceil (\overline{p} + \overline{d} + S) + \frac{P * \overline{d}}{S} + 25 * S$$

   In the first term, because $\overline{p} = 100\,000$ and we already asserted that $S$ cannot be much larger than $W$ (where we assume we may only spawn a small number $W \approx 3$ of workers), we know that $S$ has only a very minute impact on the first term. Therefore, we only need to worry about the second and third terms.

If we consider the second and third terms, we have the execution time as

$$\frac{P * \overline{d}}{S} + 25 * S = \frac{1\,500\,000}{S} + 25 * S$$

The value of this expression reduces as we increase $S$ until $S \approx 245$, at which point it starts increasing. This means that, when $S$ is small when compared to $W$, our performance increases as we increase $S$.

Thus, assuming that $S \lessapprox W$ we found performance increases as we increase $S$; but if we assume that $S \gg W$, performance increases as we reduce $S$. Therefore, there is an optimal value of $S$ for this dataset that minimizes execution time. We can further speculate from a theoretical standpoint that the optimal value of $S$ is about the same order of magnitude of $W$, but probably slightly larger than $W$.

In order to find the scalability limit, three workers were deployed for five different subpartition numbers. The results of this experiment are shown in table 1.

| Number of subpartitions | Execution time (s) |
|---|---|
| 1 | 159.678 |
| 2 | 153.902 |
| 3 | 149.154 |
| 4 | 251.879 |
| 8 | 353.641 |

Table 1: Execution time (in seconds) of each attempts for 1, 3, 6, and 100 subpartitions.

As can be seen, practical results show that the optimal number of subpartitions is approximately the number of workers.

5. How does the coordinator know which partitions to send to the workers for the merging phase?
**A:** Consider $S = 3$ and that we originally have $P = 2$ data partition, which are named `x` and `y`. The coordinator already knows a priori the scheme that the workers will use to subpartition a partition: a partition with name `x` is split into subpartitions `x.0`, `x.1` and `x.2` (because $S = 3$, we split `x` into three subpartitions); and it knows that, to obtain partial result 1 (`partial1`), it needs to aggregate subpartitions `x.1` and `y.1`.
When a worker is finished processing a `SPLIT`, it returns a list containing the names of the subpartitions it created for partition `x` (which are `x.0`, `x.1`, `x.2`). Upon receiving this response, the coordinator knows that `x.0` will be used to create partial result 0 (`partial0`), `x.1` will be used to create `partial1`, and `x.2` will be used to create `partial2`;
so the coordinator has a data structure `doneSubpartitions` that maps a partial result ID (in this case, the ID may be 0, 1 or 2) to the list of subpartitions that are used to build that partial result and that are ready; thus, `doneSubpartitions[1]` is the list of subpartitions that are required to build `partial1` and that are ready to be used (aka are already in storage because the `SPLIT` that is meant to create it has already finished). Therefore, when the response arrives, the coordinator adds `x.0` to `doneSubpartitions[0]`, `x.1` to `doneSubpartitions[1]` and `x.2` to `doneSubpartitions[2]`.
To know if a `MERGE` can be performed, the coordinator has to find any partial result `i` that has `doneSubpartitions[i].size() == P`, and if so then it just needs to send a command to `MERGE` all files in the list at `doneSubpartitions[i]`; for instance, if `doneSubpartitions[1].size() == 2`, then all requirements to build `partial1` are ready, and a `MERGE` command will be sent to merge the contents of `doneSubpartitions[1]` (which are assumed to be `{"x.1", "y.1"}`).

6. How do workers differentiate from the two tasks they have to perform (partial aggregation, merging subpartitions)?
**A:** With different message types. Partial aggregation is done when a `SPLIT` message is received, and merging subpartitions is done when a `MERGE` message is received. A worker can find the type of the message it is receiving because, when we serialize a message, the first element is the message size, the second element is the message type (`REQUEST` or `RESPONSE`), and the third element is the operation (`SPLIT` or `MERGE`), so then the worker can appropriately know what operation the message is referring to, and correctly deserialize the message as well (since `SPLIT` and `MERGE` messages have different data in the requests and responses).

7. Give a brief sequence diagram to show how the coordinator and workers communicate. Add some details about the communication format.

**A:**

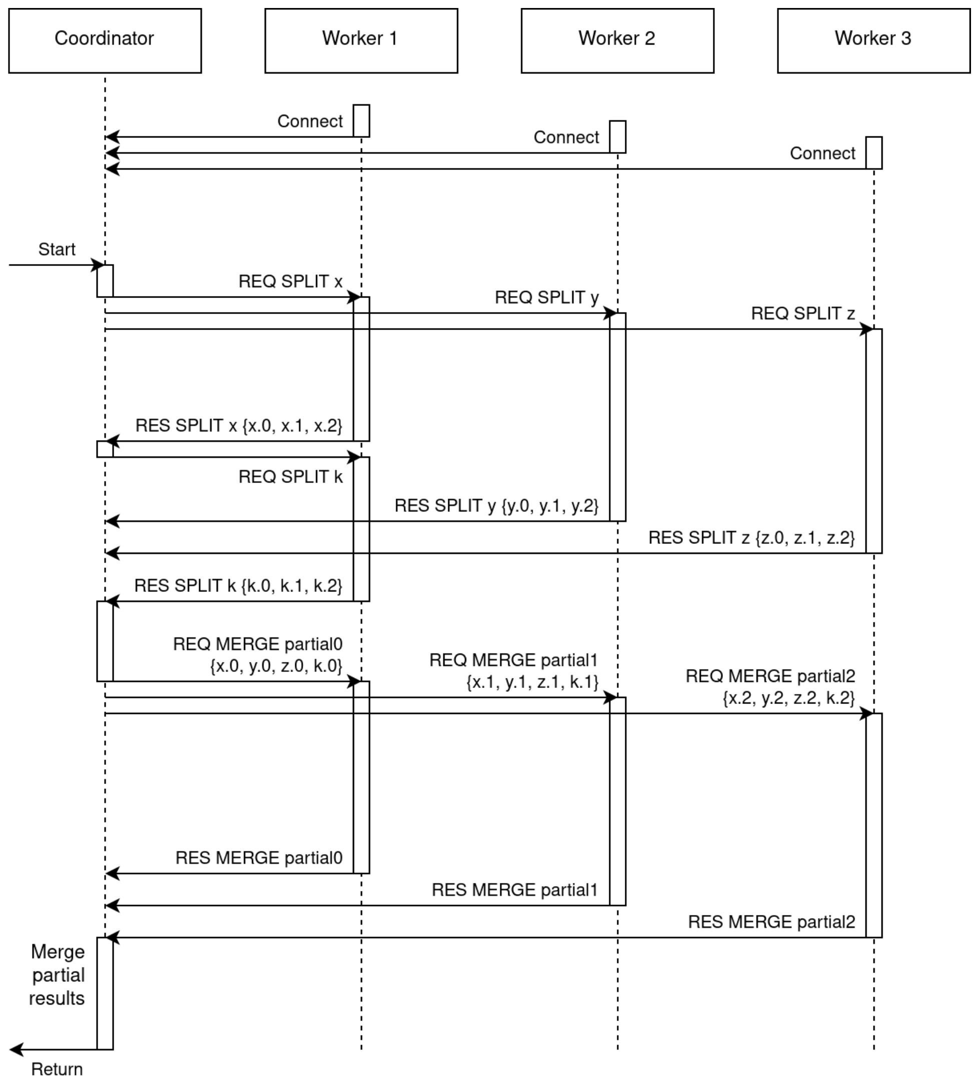

Coordinator
Worker 1
Worker 2
Worker 3
Connect
Connect
Connect
Start
REQ SPLIT x
REQ SPLIT y
REQ SPLIT z
RES SPLIT x {x.0, x.1, x.2}
REQ SPLIT k
RES SPLIT y {y.0, y.1, y.2}
RES SPLIT z {z.0, z.1, z.2}
RES SPLIT k {k.0, k.1, k.2}
REQ MERGE partial0 {x.0, y.0, z.0, k.0}
REQ MERGE partial1 {x.1, y.1, z.1, k.1}
REQ MERGE partial2 {x.2, y.2, z.2, k.2}
RES MERGE partial0
RES MERGE partial1
RES MERGE partial2
Merge partial results
Return

Figure 5: Sequence diagram for a scenario with 3 workers, 4 partitions (`x`, `y`, `z`, `k`) and 3 subpartitions per partition.

`TYPE` is the type of message; `TYPE = 1` corresponds to a request (`REQ`), `TYPE = 2` corresponds to a response (`RES`).
`OP` is the operation; `OP = 2` corresponds to a `SPLIT`, `OP = 3` corresponds to a `MERGE`.

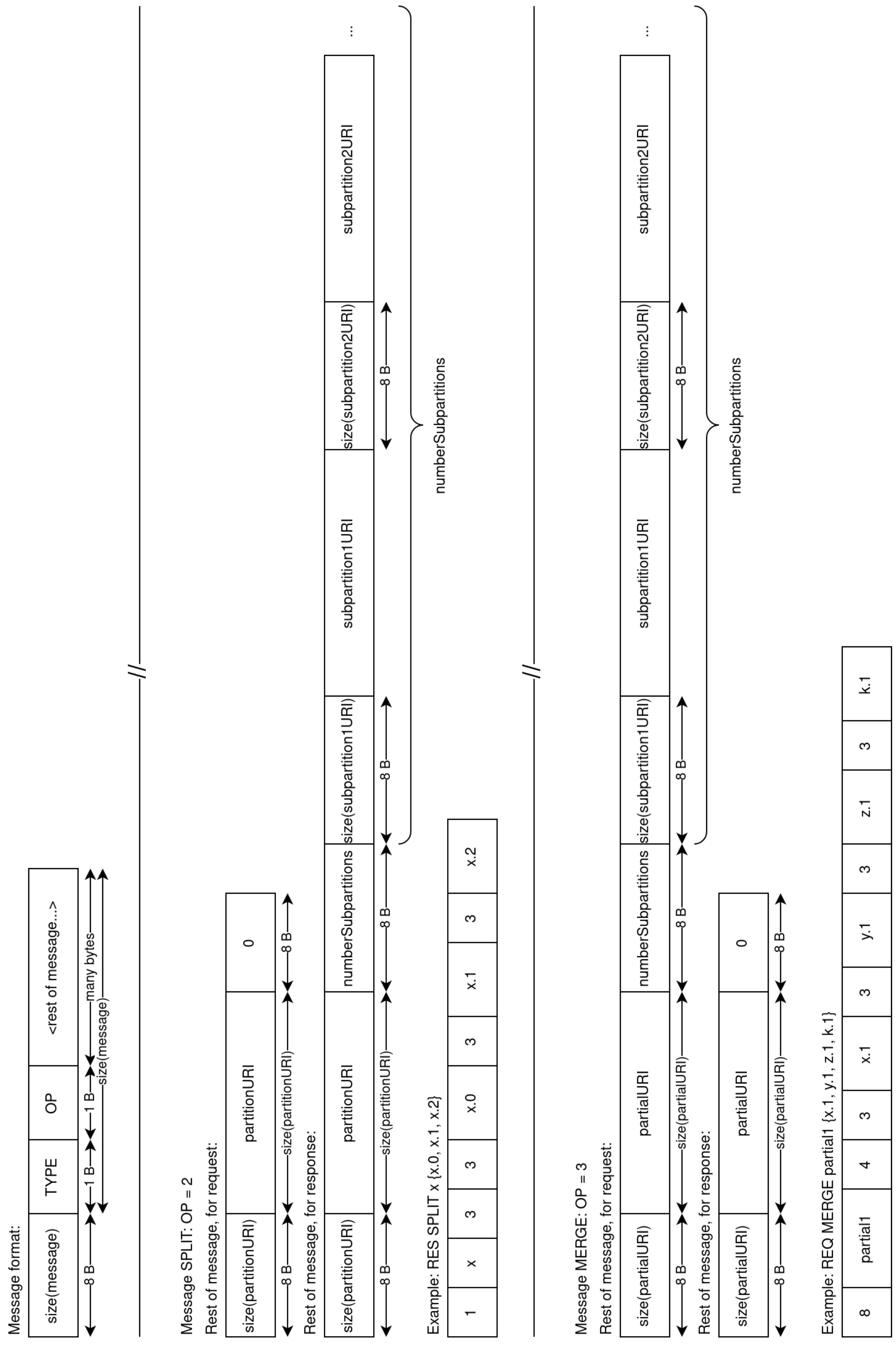


Figure 6: Message formats and examples of serialized messages.

## Part 3: Notes

Currently only one unit test is in use, the runTest.sh script. This script was adapted to the current implementation and its input file size was increased. The old resilience and workload unit tests were removed, as the current solution does not support node failures and the workload being used in the first test is already significant.