Application of Event Sourcing in Research Data Management

Jedrzej Rybicki
Juelich Supercomputing Center (JSC)
Juelich, Germany
Email: j.rybicki@fz-juelich.de

***Abstract*—Event sourcing is an architecture pattern successfully applied in modern microservice-oriented web applications. It enables better scalability, integration, and traceability by changing the way in which data are handled in those distributed systems. There are many differences, however, between data used by commercial applications and research data. In this paper, we examine if and how event sourcing can be applied in the field of research data management and what ramifications and benefits can it bring. One of the most important rules of the pattern is to record and publish all the changes ever done to a data item. Therefore, not only the current version of the item exists, but also older versions and all modifications can be traced back in time. As we will show, it opens new avenues to work with research data. The publication of the changes makes it easy to replicate the data, and collaborate on them without a central authority. All these are features often required in the modern data-driven science. The concept, its suitability, ramifications, and initial performance evaluations are presented in two real world usage scenarios. The preliminary results corroborate the assertion of suitability of event sourcing in this particular field.**

***Keywords*–Data Management; Event sourcing; Replication; Performance evaluation.**

I. Introduction

The research data come in all kinds and flavors. Spanning from small items like single measurement of a parameter value, through all kinds of documents, or recordings, up to large size genome sequences or results of astrophysical observations. Data can be raw and unprocessed or curated to form highly processed secondary data. Also, there is a high veracity with regard to the intended use of the data: some of them are expected to be just safely stored (archived), some are shared (downloaded) or collaboratively edited by researchers spread all over the world. Finally, there is an option for processing the data with High-Throughput Computing (HTC) or High-Performance Computing (HPC) facilities. The variety along all the dimension, suggest that there is not a single optimal storage solution, but rather one has to decide on case-by-case basis which solution best suits the particular data and envisioned usages.

One common feature of research data is the time dimension. It can either be explicit like in case of subsequent measurements, but also implicit like different versions of curated document. Closely related to this subject is the question if the research data change at all. Does a new, modified version of the data really substitute the old one, or should the old one be kept? One could argue that for the sake of data understandability and research transparency both versions should exist with a link (or other indicator) between them in the logical data access layer. In that sense, the research data are immutable, i.e., they never change but rather new versions emerge along the old ones. Each new version or each new measurement is yet another *event* on the time axis, that should be recorded and stored to enable better understanding of the current versions.

If the data are used as input for scientific processing, researcher often requires means to define exactly the set of inputs. A *snapshot* composed of many data objects could be a useful abstraction for that. In many cases, it might also be relevant how the output of the processing changes depending on the selected data. In that case, the snapshots must be parameterized, e.g., to include only measurements from given region or time interval. If the processing supports the notion of snapshots for inputs it is also possible to create “alternative past models”, for instance to examine what results alternative measurements would led to.

A common problem with research data is the distribution. The data often need to be replicated to make distributed processing more efficient. In some of the cases, *sharding* makes more sense than full replication: on a given location only a partition of data is stored. A reason could be that a local facility is only interested in parts of the data. The problem of data distribution can also be understood as a generalization of snapshotting mentioned above. Local replica is then a snapshot and replication is creation of parametrized snapshots with final location of the data as one parameter. Replication and sharding are hard engineering problems and become even more challenging when data are modified in a distributed fashion. Some coordination is required to keep track of such changes and potentially propagate them to all interested parties. Preferably this should happen with as little overhead as possible, in particular global “locks” as in case of well-known two-phase commit distributed transactions might not be acceptable. Yet there are high expectations in regard to data consistency. As we show latter, the distributed modification of data can be better understood and solved with the notion of events mentioned above.

In this paper, we propose how modern software architecture principles subsumed under the term of event sourcing can be applied to research data. In particular, this approach seems to suit well the distributed, collaborative storing and processing of data. The paper is a work-in-progress report on our experiences with application of event sourcing paradigm in research data management. We will present two distinct use cases of storing research data with the proposed architecture and shed some light on the technical details of our approach. The paper also includes preliminary performance assessments of applications employing the concept.

The rest of the paper is structured as follows. In Section II, we shortly summarize some of the previous work on event sourcing and its application in research. Then, follows a section describing the different flavors of event-driven architectures to finally formulate event sourcing pattern and give some insights on how it can be implemented with help of Apache Kafka. The main part of this paper is the evaluation of event sourcing for managing research data, this is done based on two real-world inspired use cases which we describe in Section V. The results obtained in our preliminary evaluations are discussed before the papers ends with a conclusion and outlook on future work.

II. Related Work

Event sourcing was and is successfully applied in the domain of commercial applications [1]–[3], because of the performance, distribution, and ease of integration (especially in microservice-oriented systems) it offers. There are many evaluations of using event sourcing with particular languages or frameworks [4] [1] [5]. There is not much work, however, on how this kind of architecture approach can be applied in research data management. Of course, emerging research infrastructures should follow best development and architectural practices obtained in the commercial setting. On the other hand, there are some unique aspects of managing research data that need to be addressed. The most prominent difference seems to be caused by the Open Data Movement [6]. Research data are expected to become open, and accessible, at least in in the long term. Also, the transparent provenance standards resulting from the need of reproducibility are probably higher in the academic context. In this paper we show how event sourcing can help in achieving these features.

Müller [7] showed how event sourcing can enable retroactive computing, i.e., to examine how alternative chains of events could lead to a different final state of the system. This interesting use case can be realized if the research data are stored in an event-driven fashion and is therefore orthogonal to our work. Another application of event sourcing in broad context of research data comes from Erb and Kargl [8]. They analyzed how event sourcing architecture can be incorporated into discrete event simulations to make them better to understand, debug, and evaluate. The work is less focused on managing research data (although it is relevant in this context) but provides important insights and use cases for the event sourcing architecture.

One of the use cases discussed in this paper is replication of research data repository. State-of-the art research data management solutions like iRODS [9] or Fedora Commons [10] store data on the backend file system and metadata in a relational database. iRODS provides its own means to facilitate replication. Upon ingest of new data they can be copied to remote repository with proprietary transport protocol. To replicate an existing repository, its backend filesystem and metadata database has to be copied to a remote location by other means. To create such a consistent snapshot, original repository has to be either stopped or set into read-only mode. In this paper we will show that event store has some clear advantages over the mixture of database and file system in this scenario. In particular, it enables replication during normal operation of the original repository. By playing back all the past operations on the original repository it can be guaranteed that the replica is in a consistent (even if not most up-to-date state) state. The messaging features of event stores, guarantee that the future updates will be propagated to existing replicas. Lastly, the event stores allow for replication in the time which are convenient to the receiver, e.g., during the night hours.

The popularity and usability of event-oriented approaches in the modern microservice architectures led to availability of at least few products that can be used to built event sourcing solutions. Apache Kafka [11] is one of the most popular and we used it as a backbone of our solution. Furthermore, we already have used Kafka in context of research infrastructure, namely to extend Swift [12] with flexible namespaces [13]. Thus, some initial experiences with this product that we could build upon were available. The basic workings and most prominent features of Apache Kafka will be explained later in this paper.

For our experiments we selected Kafka as an event store. The idea of event sourcing is to store all the changes done to entities as events. Thus, to obtain a valid state of an entity, it is required to replay all the events that happened to it. We did this by retrieving the events from event store, an alternative approach would be to use one of the streaming platforms like Samza [14] or Spark [15] or streaming features of Kafka. These platforms can read data from Kafka and direct process the data, e.g., to obtain an aggregate or entity state. Such a processing is usually done in close to real-time manner. This was not required in our scenarios. In opposite we were interested in the past versions of entities. Thus, streaming platforms may combine well with our approach but their application is depending on the user requirements.

III. Event-driven architectures

Before we explain what kinds of event-driven architectures are common in distributed systems, we shall first define some basic terminology. In the remainder of this paper, we will use terms entity, aggregate, and event as defined by domain-driven design [16]. In short, *entity* is characterized by the possession of identity, a group of entities can form an *aggregate*, and changes in state of entities or aggregates are called *events*. Especially the last definition is not so common in the event-driven approaches as we will see later in this section.

Fowler presented an excellent discussion of the different kinds of event-driven architectures [3], we include only a short summary of his arguments. One of the first examples of an event-oriented approach were systems using notifications distributed through a common Enterprise Service Bus (ESB) [17]. This solution was used for enterprise-wide integration of services. In particular, upon a change within the system, a notification was sent to the bus, and then distributed to all interested parties. This was achieved in publish/subscribe fashion. The messages were usually simple in form and to actually get information about what changed, the interested party had to contact the message originator. Soon, this subsequent communication was identified as a bottleneck and new architectures emerged where messages included sufficient description of the actual change to eliminate the need for additional communication. Usually, it would include an identifier of the entity or aggregate and new values of its attributes. Such approaches were subsumed under the term event-carried state transfers, they were still relying on publish-subscribe message buses and were oriented on costly immediate information delivery.

The final evolution of message-driven architectures, and the one that is relevant for the rest of the paper is called event sourcing. Here again each change in the entity or aggregate state is recorded as an event and published. But unlike the previous approaches the publication is done with help of an event store. This component stores the events for longer period of time and beside supporting publish-subscribe interface for immediate event distribution, it also provides access to stored events from the past. Thus, the subscribers can not only just express their interest for the future events (like in approaches described above), but also request the past events. Events follow time order, and each consumer can request all the messages starting from a given time offset. At the first glance, the change might not seem to be very substantial but it has a lot of implications. In particular, each service in the system can maintain its own state replica, and thanks to the notifications the state can be kept up-to-date. Importantly, changes of the state can be done in a consistent manner even without costly distributed transactions or central coordination in form of global locking. Further, the local state can be erased and rebuild from the past events. As all the changes in the system are stored as events, it is only required to play all the events in timely order to recreate the final state.

We believe that event sourcing can be successfully applied to manage research data. Since modification of an entity is done by publishing a "modification" event, no information is lost and old versions of the entities can always be retrieved. It is much more meaningful, for instance, to see that a value was added and then removed from a stream of measurements, rather than have just the final version without the given value. Furthermore, the event sourcing removes the need for central coordination when working with research data. Each researcher can pick and choose the events she is interested in, publish her modifications, or withheld modifications from others in her local version. This is not only a technical argument but also a social one as it plays well with the open nature of modern data-driven research.

It should be stressed that event sourcing is not a plug-and-play solution that can be easily included in the existing systems. It is an architecture decision that strongly influence the design of the system especially in data access layer.

IV. Implementation of event sourcing

In this paper, we aim at evaluating the general suitability of event sourcing architectures to manage research data. We are more focused on general insights rather than rigorous performance evaluations (which might follow up this paper).

A. Apache Kafka

Kafka is a "distributed streaming platform" [18], its functionality boils down to three main aspects:

1) publish/subscribe system,
2) fault-tolerant distributed storage,
3) processing of streams.

From our perspective the first two are the most important. Kafka uses notion of *records*, that are roughly equivalent to events from our previous definition. The records are very flexible structures comprising of a key, value, and timestamp. Both key and value are basically streams of bytes handled over to Kafka. Records belong to *topics*, i.e., categories of events. For efficiency Kafka divides them into *partitions*. Partition is an immutable sequence of records with strict time-based ordering. Upon publication of a new record *Publisher* assigns it to a partition. This can be done in a programmatic way (e.g., based on hash of the message or any other information about the structure of records). *Consumers* can subscribe to given topic and will receive records in the same order as they were published. Each consumer can, also, select its offset in the topic, i.e., the position of the next message she would like to receive. It could be the newest one, but also the oldest one in case of rebuilding of an entity. The producer, on the other hand, only appends to new records to the partition log (which does not require random access to the storage and is very beneficial for the overall performance). If required, consumers can form groups per topic. In such case each record will be delivered to exactly one consumer from a group in a round robin fashion.

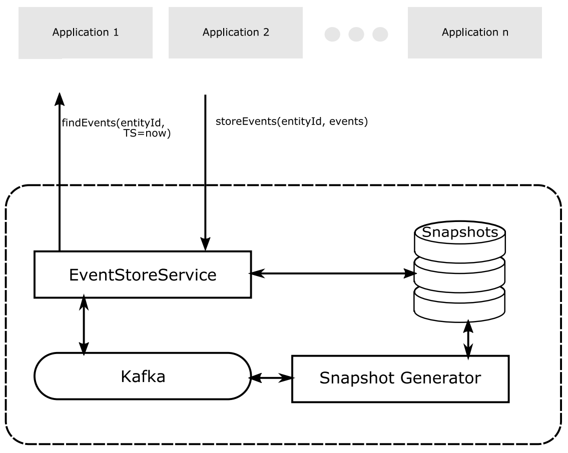

Figure 1. Architecture overview of our solution

As already mentioned, Kafka is more than yet another publish/subscribe system. Its strength is the fact that the records are written to disk for defined periods of time in a fault-tolerant way. In particular partitions of a topic can be replicated and have their own retention policies. To increase data safety, consumer upon publication of a record, can wait for a defined number of acknowledgments from all replica managers.

A good intuition of what Kafka is and how it differs from other messaging and storage systems is to see it as a system that allows access to both past and future data [18]. Storage repositories, like filesystems or databases, are providing the data stored in the past. Whilst messaging systems allow to subscribe for the future data, i.e., data that will become available in the future will be distributed to all clients that subscribed.

B. Kafka deployment

There are many ways in which Kafka can be deployed. In general, Kafka servers can form a cluster and coordinate through Apache Zookeeper [19] (which can also be deployed as cluster). For our experiments we used, however, one host deployment, with one instance of Kafka and one instance of Zookeeper residing on the same host. The basis for our deployment were Docker images provided by Confluent Platform [20]. Docker enables rapid provisioning of software, thus this setup can be further extended towards cluster setting. With one exception that will be described later we used the pre-configured defaults of Kafka and Zookeeper as defined in the images.

Both Kafka and Zookeeper were deployed on a host with 1 VCPU, 2 GB RAM and 100 GB storage using Ubuntu 16.04

LTS and Docker in version 1.13.1. We used Confluent platform version 3.3.1 which included Kafka 0.11.0.1 and Zookeeper 3.4.10. To communicate with Kafka we used Python library `kafka-python` version 1.3.4.

C. Programming interface

As mentioned above, event sourcing is not a plug-and-play extension that can be added to an existing systems. It requires some changes in the persistent layer. Rather than performing fetches from a "source of truth" central database system to obtain current state of an entity, each service has to work with streams of events rather to build their states.

For a better understanding of the mental model required to work with events as persistence layer, let us discuss following code example.

```
# get current version
pastEvents = findEvents(entityId)
entity = new()
applyEvents(entity, pastEvents)

# do something with it
newEvents = processCommand(entity,...)
applyEvents(entity, newEvents)

# publish changes
storeEvents(entityId, newEvents)
```

In the first lines, a substitute for fetch command from traditional database-based approaches is presented. Crucial is the function `applyEvents` which applies all the events (stream of modifications) to a newly created, pristine entity. The modification to an existing entity is done with `commands`, which rather than modifying it directly, produce a list of events that need to be applied to the existing entity. Lastly, to make the changes induced by a command persistent, it is required to store the list of events.

To improve performance of such workflows it is possible to use snapshots. Snapshots would be entities which are rendered with `applyEvents` function and persisted together with a timestamp. The same function (without modifications) can be used to update snapshot or produce an entity for a timestamp of higher value than the timestamp of the snapshot.

A rudimentary overview of our architecture is depicted in Figure 1. Kafka is abstracted by an event source service which can also use snapshots if required. On the top level applications constituting user interfaces reside. In our case there are two web applications created. One is a simple research data repository and the other is a measurement display. These emulate a means in which data from event store would be served to end users.

V. Use cases

We defined two use cases inspired by real-world usage of research infrastructures. The use cases are implemented with the application of event sourcing. In this section we will describe the use cases and how we implemented them.

A. Measurements storage backend

The inspiration to this use case was a common practice of collecting research data from a distributed network of sensors.

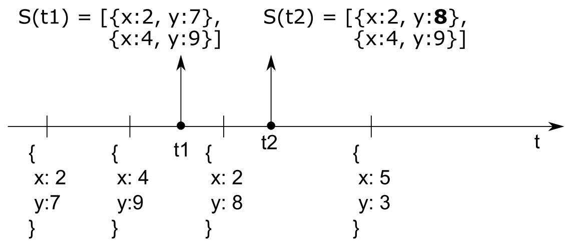
S(t1) = [{x:2, y:7}, {x:4, y:9}]
S(t2) = [{x:2, y:8}, {x:4, y:9}]
t1
t2
t
{ x: 2 y:7 }
{ x: 4 y:9 }
{ x: 2 y: 8 }
{ x: 5 y: 3 }

Figure 2. Modeling measurement stream as series of events. Aggregated states at $t1$ and $t2$.

We assumed that each station in the network periodically measure some values and uploads those to the central sink for further processing. We, also, wanted to account for a possibility to upload corrections of the previous measurements (e.g., upon detection of sensor malfunction). A simple domain analysis suggest a model in which each station in the network is an entity and each measurement would be an event. Hence, the state of the entity at given time would be a stream of measurements (and corrections) collected up to this moment. In fact, we are somehow close between discussing aggregates and entities but in general, this is not relevant here.

We implemented this use case in such a way that a process produces measurements of two values (x, y) which are then uploaded with a timestamp and station identifier to the central repository (i.e., Kafka server). The state of the entity is given by a $f(x) = y$. Therefore, if a measurement arrives with a value of x which is already present it would be treated as a correction. This approach is depicted in Figure 2. State of the entity requested at $t = t1$ would include values $x : 2, y : 7$ and $x : 4, y : 9$. Later, a correction of the first measurement arrives, thus the entity returned at $t = t2$ is built of values $x : 2, y : 8$ and $x : 4, y : 9$. It is worth noticing that the corrected measurement is still present in the event store and thus it is always possible to request state at $t1$.

Because the timestamps are also recorded it is possible to generate previous views of the entity state (e.g., before a correction). We implemented this functionality in form of a web application that connects to Kafka, pull all the relevant measurements and presents them in form of a x, y plot. Since building of the final state of a given entity boils down to replaying all the events (measurements), an obvious optimization would be to store previous states as snapshots and only apply events that happened between the snapshot time and the requested time. Many different strategies for generating snapshot are conceivable. A simple strategy is to store a snapshot depending on the number of events that needs to be read from Kafka to rebuild the requested entity state. If the number is higher than some pre-defined threshold a snapshot will be stored. This strategy aims at optimizing the overhead of communicating with the event store. An alternative like time-based snapshotting is less effective in this, especially when events are not evenly distributed in time. Two states of an entity, which are distant in time, might not differ very much in number of events that happened to them. All the code we developed can be found in the GitHub repository [21] for further analysis.

A system built in this manner would have to provide good performance on at least two fields. Firstly, a high throughput

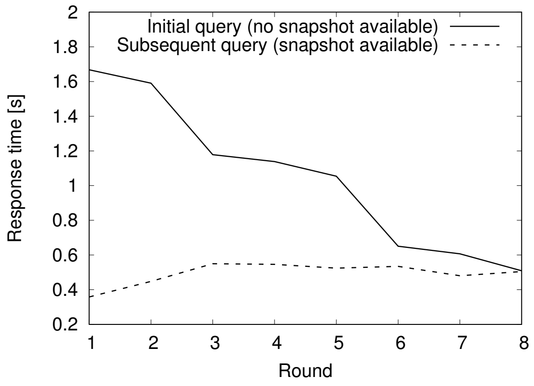

Figure 3. Querying measurements storage: In each round aggregated entity for decreasing timestamp is requested twice.

with regard to the uploading measurements must be granted. Secondly, the entity states delivered by aggregating the measurements should be produced quickly. The throughput was measured by uploading repeatedly large batches of measurements directly to Kafka. We tested batch sizes of 5 000, 10 000, and 50 000. The experiments were divided into 10 rounds, in each round a defined number of measurements were uploaded and time measurement was conducted. We summarize the obtained results in Table I. Our very rudimentary experiments suggest that it is possible to arrive at the throughput of almost 6 000 messages per second. Given, rather simple single-node deployment, we consider this performance sufficient.

TABLE I. MEASUREMENTS UPLOAD THROUGHPUT.

Batch size	Throughput (measurements/s)
5 000	5819.39
10 000	5938.39
50 000	5500.44

Our second heuristic was the performance of delivering the aggregated states of entities. For that we have conducted a two-phased experiment. Firstly, about 50 000 evenly distributed measurements were uploaded to Kafka. Subsequently, entity states for decreasing timestamps (i.e., from the most current downwards) were requested and response times were recorded. For each timestamp we performed two requests, thus the first response was produced solely based on measurements from Kafka and resulted in storing a new snapshot. It was in turn used to answer the subsequent query. The results are show in Figure 3. Solid line depicts the initial queries where no snapshot was available, subsequent query in each round benefited from the just-created snapshot and, thus, were answered quicker (dashed line). It can be seen that snapshots indeed improve the response times and are crucial especially for more complex entities aggregating large number of measurements (left side of the plot). This experiment was intended to show the best possible gains obtained by creating snapshots. The initial query always required full list of events for the requested entity. It is worth mentioning that we stored the snapshots in a simple in-memory store, so the difference in response times is mainly caused by retrieval of events from Kafka and process of rebuilding of the requested entity.

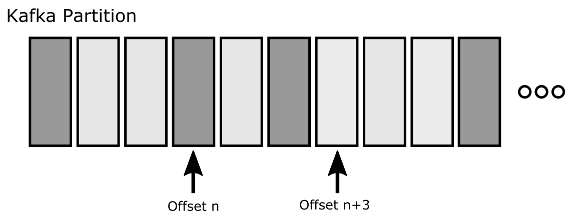

Figure 4. Storing data objects (dark gray) and files (light gray) in Kafka.

B. Replication of research data repository

Research data often need to be replicated. The reasons for that might be the data-preservation policies that require multiple copies, or efficiency considerations. To this end, we want to examine how event sourcing can be used to implement such replication.

For our tests we used data from a real-world research data repository EUDAT B2SHARE [22]. It is part of the EUDAT research data infrastructure built to serve researchers from all across Europe [23]. The data model used by B2SHARE defines data objects as a set of metadata, persistent identifier (PID), and a list of files. Current model does not support versioning of the objects and files. B2SHARE is based on open-source Invenio system [24] which offers an API that we used to download all the data from the repository. We obtained 538 objects and total size of data amounted to about 40 GB. Subsequently, we uploaded the data objects and files to our Kafka instance. Data objects were just JSON documents as downloaded from B2SHARE, they included basic metadata and names of files attached to the object, we used their PIDs as keys in Kafka. Files were uploaded with filenames as keys, and binary content as values. Larger files needed to be split into chunks of 40 MB. Such large records required a change in the default configuration of Kafka. All the data were put into one partition in Kafka and objects were put before files they referred to. We used creation time to sort the data objects, i.e., newer data objects have higher offsets in the partition. The code we developed for both scrapping the original data, as well as replicating it can be found in GitHub repository [25].

It might not be immediately clear from the above description what are the entities and events in this scenario. The entities in the system are the data objects, and events are modifications (or creations) of metadata descriptions which are stored in Kafka and uploads (or modifications) of files belonging to the data objects. The approach is shown in Figure 4. The dark gray rectangles depict events of uploading metadata descriptions of objects, and light gray rectangles are the upload of files corresponding to the objects. For example at offset n in partition there is a metadata object, at offset $n+3$ starts a list of three files belonging to an object at $n+2$ (it could also be three chunks of a larger file).

The evaluation of this use case comprised of two phases. First, the time required to upload the content of the B2SHARE repository to Kafka was measured. Afterwards, the repository was restored at the target site. These two phases simulate full replication of the research data. In total, 4267 events were generated to upload the content. The results are summarized in Table II.

There is apparently not much difference in performance of uploading and downloading. For comparison we copied

TABLE II. REPLICATION TIMES.

Phase	Time (s)
Upload	2170
Download	2282

the same data between the same hosts using Secure Copy Protocol (SCP) which completed the task in about 17 minutes. It should be stressed that the upload and download to Kafka can be done at the same time so that complete replication of B2SHARE data would needed about 30 minutes.

VI. DISCUSSION

To evaluate the applicability of event sourcing in research data management we implemented two use cases. In the first one we emulated gathering and evaluating of measurements from a distributed sensor network. We have identified two critical aspects for performance of event sourcing here. Firstly, the throughput for collecting the measurements. The values obtained in our experiments are pretty high. This comes at no surprise as event sourcing allows for efficient resources usage, there is no need for random access or costly data removal, it only has to support addition of data to the log. The other performance aspect were the response times of the interface serving aggregated entities. Here we have noticed that even a simple snapshotting strategy can substantially improve response times. When discussing this use case it is important to ask a question how hard would it be to change an existing application to support event sourcing. It would clearly require a change in the user-facing application, it must support Kafka as the source of information and being able to reconstruct the state of entity (as a stream of measurements in our case) from recorded events. By offering retrospective views on data it gives direct advantages to the researchers. One further advantage would be the ease of integration with other services and easy way towards replication of the data.

The replication of data constituted the central point of our second use case. We used data from existing repository to make the evaluation results more meaningful. The main challenge here was coping with the large files, they need to be split into chunks before uploading. We measured time for complete replication of the repository. There are many more possible aspects of this use case that might be relevant in the future. First one, is the need of keeping replicas up-to-date this would require changes in the existing repository software so that it would emit information about user actions (upload/modification of objects) and thus inform replicating sites about availability of new data. There could also be possibility of creating shards (i.e., partial replicas of the data), for that a way of defining sharding strategy needs to be put in place. This questions are pretty specific to particular use case, a research infrastructure is addressing and thus were not included into the evaluation. Since we are replaying the events from the oldest one, the target repository remains in a valid (i.e., consistent) if not most up-to-date state during the on-going replication.

There are some common aspects for both use cases. On very high level event sourcing requires a different mindset when dealing with data. In particular, deleting data is almost impossible. It is possible to create a correction (as we discussed), but the original event will remain in the event log. Developers should always be aware of this, especially when dealing with sensitive data like personal health data. Somehow related is the problem of event modeling. In this paper we used very simple modeling that might not be optimal. A solution is always specific to the problem that is to be solved. Another question is the evolution and extension of the event models. It is possible to do this, but it requires changes in the `applyEvent` function.

Event sourcing is not just new approach to data management but also an integration pattern. It might require some changes in the existing applications but as soon as they begin to use event store as persistence layer, it is very easy to add new services that can use the data. The application can run in parallel, use the same data, there is no need for coordination or global locking. The applications can be stateless and thus it is also possible to create multiple instances of the same application for instance to improve response times. In our cases, it would be easy to create one more replica of the repository or spin off one more user interface to present aggregated measurements.

VII. CONCLUSION AND FUTURE WORK

In this paper, we evaluated the application of event sourcing for managing research data. We defined two use cases and examined how they can be implemented in this architecture style and what performance they display. We also showed the functional benefits of this approach: how old versions of the data can be retrieved and how data can be seamlessly replicated. Although, these are preliminary experiences we believe that they can be valuable for both developers and researchers. The results obtained in the performance evaluation indicate the applicability of the approach and particular technology (Apache Kafka) in the real world scenarios. Also, the high-level ramifications of the proposed approach, in particular the concept of making the research data *de facto* immutable is a profound change in way we think about data.

In our future work, we plan to put the gained experiences into practice by implementing event sourcing in the context of research data infrastructures.

REFERENCES

[1] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure. Microsoft patterns & practices, 2013, ISBN: 978-1-62-114016-0.

[2] G. Young, Event Centric: Finding Simplicity in Complex Systems. Addison-Wesley, 2012, ISBN: 978-0-32-176822-3.

[3] What do you mean by "Event Driven"? [Online]. Available: https://martinfowler.com/articles/201701-event-driven.html [retrieved: Mar., 2018]

[4] B. Nobakht and F. S. de Boer, Programming with Actors in Java 8. Springer Berlin Heidelberg, 2014, pp. 37–53, ISBN: 978-3-66-245231-8.

[5] K. Lee, Event-Driven Programming. Springer London, 2011, pp. 149–165.

[6] M. B. Gurstein, "Open data: Empowering the empowered or effective data use for everyone?" First Monday, vol. 16, no. 2, 2011, ISSN: 13960466.

[7] M. Müller, "Enabling retroactive computing through event sourcing," Master's thesis, University of Ulm, 2016.

[8] B. Erb and F. Kargl, “Combining discrete event simulations and event sourcing,” in Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques SIMUTools ’14, 2014, pp. 51–55, ISBN: 978-1-63-190007-5.

[9] A. Rajasekar, R. Moore, C.-Y. Hou, C. A. Lee, R. Marciano, A. de Torcy, M. Wan, W. Schroeder, S.-Y. Chen, L. Gilbert, P. Tooby, and B. Zhu, iRODS Primer: Integrated Rule-Oriented Data System, ser. Synthesis Lectures on Information Concepts, Retrieval, and Services. Morgan & Claypool Publishers, 2010, ISBN: 978-1-62-705972-5.

[10] D. Wilcox. Stewarding research data with Fedora. [Online]. Available: http://library.ifla.org/id/eprint/1796 [retrieved: Mar., 2018]

[11] J. Kreps, N. Narkhede, and J. Rao, “Kafka: A distributed messaging system for log processing,” in Proceeding of 6th International Workshop on Networking meets Database (NetDB ’11), Jun. 2011, pp. 1–7.

[12] J. Arnold, OpenStack Swift: Using, Administering, and Developing for Swift Object Storage. O’Reilly Media, 2014, ISBN: 978-1-49-190082-6.

[13] B. von St. Vieth, J. Rybicki, and M. Brzeźniak, “Towards flexible open data management solutions,” in Proceedings of the 40th IEEE International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO ’17), May 2017, pp. 233–237, ISBN: 978-9-53-233090-8.

[14] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, “Samza: Stateful scalable stream processing at LinkedIn,” Proc. VLDB Endow., vol. 10, no. 12, Aug. 2017, pp. 1634–1645, ISSN: 2150-8097.

[15] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, “Apache spark: A unified engine for big data processing,” Communications of the ACM, vol. 59, no. 11, Oct. 2016, pp. 56–65, ISSN: 0001-0782.

[16] E. Evans, Domain-Driven Design. Addison-Wesley, 2004, ISBN: 978-0-32-112521-7.

[17] D. Chappell, Enterprise Service Bus. O’Reilly Media, 2009, ISBN: 978-0-59-600675-4.

[18] Apache Kafka. [Online]. Available: https://kafka.apache.org/ [retrieved: Mar., 2018]

[19] F. Junqueira and B. Reed, ZooKeeper: distributed process coordination. O’Reilly Media, 2013, ISBN: 978-1-44-936130-3.

[20] Confluent Platform Docker Images. [Online]. Available: https://github.com/confluentinc/cp-docker-images [retrieved: Mar., 2018]

[21] J. Rybicki. GitHub repository with the source code for measurements use case. [Online]. Available: https://github.com/httpPrincess/measurements2kafka [retrieved: Mar., 2018]

[22] EUDAT B2SHARE. [Online]. Available: https://b2share.eudat.eu/ [retrieved: Mar., 2018]

[23] W. Gentzsch, D. Lecarpentier, and P. Wittenburg, “Big data in science and the EUDAT project,” in Proceedings of the Service Research and Innovation Institute Global Conference, Apr. 2014, pp. 191–194, ISBN: 978-1-47-995193-2, ISSN: 2166-0786.

[24] Invenio. [Online]. Available: http://invenio-software.org/ [retrieved: Mar., 2018]

[25] J. Rybicki. GitHub repository with the source code for replication use case. [Online]. Available: https://github.com/httpPrincess/b2kafka [retrieved: Mar., 2018]