EPJ Web of Conferences **214**, 04035 (2019) https://doi.org/10.1051/epjconf/201921404035
CHEP 2018

Storage events: distributed users, federation and beyond

A. Paul Millar[1,*], *Olufemi* Adeyemi[1], *Vincent* Garonne[3], *Dmitry* Litvintsev[2], *Tigran* Mkrtchyan[1], *Albert* Rossi[2], *Marina* Sahakyan[1], and *Jürgen* Starek[1]

[1]Deutsche Electron Synchratron (DESY)
[2]Fermilab National Laboratory
[3]University of Oslo

Abstract. For federated storage to work well, some knowledge from each storage system must exist outside that system, regardless of the use case. This is needed to allow coordinated activity; e.g., executing analysis jobs on worker nodes with good accessibility to the data.
Currently, this is achieved by clients notifying central services of activity; e.g., a client notifies a replica catalogue after an upload. Unfortunately, this forces end users to use bespoke clients. It also forces clients to wait for asynchronous activities to finish.
dCache provides an alternative approach: storage events. In this approach the storage systems (rather than the clients) become the coordinating service, notifying interested parties of key events.
At DESY, we are investigating storage events along with Apache OpenWhisk and Kubernetes to build a "serverless" cloud, similar to AWS Lambda or Google Cloud Functions, for photon science use cases.
Storage events are more generally useful: catalogues are notified whenever data is uploaded or delete, tape becomes more efficient because analysis can start immediately after the data is on disk, caches can be "smart" fetching new datasets preemptively.
In this paper we will present work within dCache to support a new event-based interface, with which these and other use cases become more efficient.

1 Introduction

The behaviour of all network accessible storage systems is constrained by the data access protocol(s) they implement. Multiple protocol support may enhance the features and flexibility of storage systems.

A feature of almost all storage protocols is the request-response paradigm. In this paradigm, the client issues a request to the server and the server gives a single response, which indicates the result of some action attempted on behalf of the client.

For some protocols, certain responses are a composite. They contain multiple elements that describe different parts of the clients request (e.g., the response to a WebDAV PROPFIND [1] request, or an FTP MLSD [2] request). However, even in these cases, the response is finite, bounded, and the direct result of a specific request.

*e-mail: [paul.millar@desy.de]

There are operations that may take a long time for the server to complete. Such requests are often processed asynchronously. This is where the client makes an initial request that creates a job within the server, which contains the details of the desired operation. This initial request returns quickly, providing the client with some unique ID for the job.

The client can then poll the server to discover when that job has completed. This involves the client requesting the current status of the job by including that job's ID in the status request. The server quickly replies with the job's current status, in particular whether or not the job has completed. The SRM v2.2 protocol [3, 4] includes such asynchronous operations.

2 Problems with standard approach

There are several problems with the current approach to storage interaction.

Storage systems are generally scientific domain agnostic. The protocols that deal directly with data access and data management generally operate in terms of stored bytes, rather than more abstract domain-specific concepts. Therefore, any interactive browsing capability provided natively by the storage server (e.g., an interactive directory explorer) is generally too limited for scientific exploration.

Scientists wishing to explore data wish to do so using domain-specific views. These views typically support sophisticated queries that select from the available data using domain-specific metadata, either extracted from the stored data or supplied from some external source. In either case, the domain-specific view must know when the available data changes, to update its catalogue of available data and potentially to extract and store domain-specific metadata from any new data.

This requires a level of synchronisation between the domain-specific view and the storage system which is currently not possible. The domain-specific view must somehow learn of any changes in the storage system. It could periodically scan the storage system. This is relatively slow, introduces a latency in changes being propagated, and could potentially hurt the storage system's performance if done too aggressively. Alternatively, the client that uploads fresh data, deletes old data, or renames existing data could also notify the domain-specific view of these changes in a robust and timely fashion. This approach brings several benefits, but requires that data is always managed by a custom client. Also, handling operational issues (such as lost data, due to hardware failures) becomes problematic.

The same problem exists in federated environments, such as WLCG. In order for a client of a federated storage system to read a specific file, it must first learn in which storage systems that file is stored. This, in turn, implies some central catalogue that knows where files are located. If data is uploaded to a storage system without the corresponding entry in the file catalogue being added then the data will consume capacity but is otherwise inaccessible, so-called "dark data". Likewise, if data is removed but the corresponding entry in the file catalogue is not removed then attempts to read the file will fail, so called "dangling links".

Another problem is with long-running jobs. If the client wishes to learn about changes to the submitted job in a timely fashion, it will need to query the current status often. In most cases, the status will not have changed from the previous request. Processing such requests consume resources unnecessarily.

This problem is exacerbated if the asynchronous job is a composite (or "bulk") operation, where the job targets several files in a similar fashion. For such bulk requests, the size of the response will be proportional to the number of targets within the job. As the number of targets increases, so will the size of the job's status, making polling requests more expensive to calculate, and causing the response to consume more network bandwidth.

3 Storage Events

In this paper, we introduce a new way of interacting with scientific storage: storage events. The core concept is that events are dispatched when something happens within the storage system. They are dispatched so that clients receive interesting events in a timely fashion.

This approach is radically different from existing interactions. Instead of the regular pattern of a client issuing a request and the server replying with the response, the event is sent from the storage server to the client spontaneously, with minimal delay after the triggering activity within the storage system.

The lack of delay allows the client to track changes within the storage system in near real-time. There is no polling delay, where the client only learns of changes when it asks if something has changed. This allows a client to maintain a "live" view of the current storage, where changes in the storage are seemingly reflected instantaneously.

Domain-specific views may be kept up-to-date by receiving events when data is uploaded, deleted or renamed. On receiving the event, the view can update its internal representation, and can even update the visual representation on the fly. An example of this is shown in Figure 1.

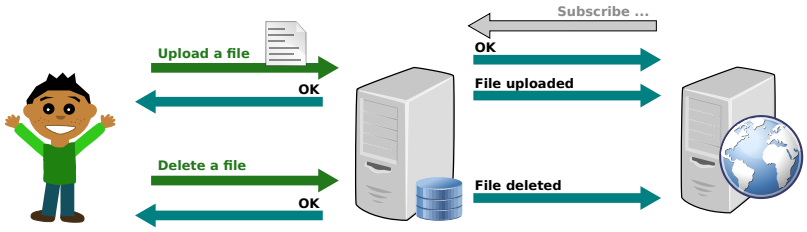

Figure 1. Illustration of how a domain-specific web view or file catalogue is kept up-to-date through storage events. This example shows a client uploading a file and deleting a file. The catalogue receives corresponding events.

Similarly, in federated storage environments, file catalogues are notified when files are uploaded, deleted or renamed. They can then update their inventory accordingly. This allows users to employ standard clients for data uploads without creating catalogue synchronisation problems.

Storage events also solve the scaling problem with polling. As events are dispatched when there is a change in state, the client does not need to poll to learn when the job has changed state. Therefore, the majority of the requests are no longer needed.

The use of events to describe changes in storage has been available in the commercial cloud environments for some time; for example, Amazon Web Services (AWS) supports AWS Lamda [5], Google provides Google Cloud Functions [6], and Microsoft provide Azure Functions [7].

Although storage events are not widely supported in open-source storage solutions, event processing is a common paradigm. There are open-source projects that focus on event processing, with projects like Apache Storm [8], Apache Spark [9], Apache Kafka [10], Apache Nifi [11], Apache Samza [12], Apache OpenWhisk [13] and Kubeless [14] all providing pieces that may be combined to build a compatible, coherent event-processing environment.

4 dCache implementation

In the dCache project [15, 16], we now provide two approaches for supplying storage events: Kafka and Server-Sent Events (SSE). Both approaches allow clients to receive events, triggered by a range of activity within dCache, with minimal delay.

The two approaches are not equivalent, but provide complementary solutions. For a given scenario, one or the other may be most appropriate.

The following sections describe both approaches, with a final comparison describing where each may be of use.

4.1 Kafka

The Apache Kafka project [10] is an open-source project to provide a highly scalable distributed streaming platform. It is used for building real-time streaming data pipelines that get data reliably between systems or applications, optionally transforming or reacting to those events.

Within dCache the billing service accepts dCache internal messages, each describing significant data-lifecycle activity: data being uploaded, read, flushed to tape, staged back from tape, and deleted. Upon receiving a message, the billing service appends this information to a log file or into a database.

Adding Kafka support to dCache involved updating the services that send dCache-internal messages to the billing service so that they now can also send an event to a Kafka cluster. The Kafka event is a JSON object that contains all the details sent to the billing service. All events are sent to a single topic called "billing".

Kafka clients can subscribe to this billing Kafka channel. Such clients will receive notification of all billing events.

Although Kafka supports client authentication and an authorisation model for events, dCache currently does not make use of this. Admins may be able to restrict access to this topic; however, if a client can see any billing events then it can see them all. This by-passes dCache's namespace permissions model. For example, the names of files uploaded into a private directory will be visible to all Kafka clients.

4.2 Server-Sent Events

Server-Sent Events (SSE) [17] is a standard protocol for delivering events over HTTP. In general, it is widely supported. All mainstream web-browsers have built-in support, and all major programming languages have libraries that provide SSE clients.

The SSE protocol has some support for reliable event delivery, where clients can request delivery of any events that might have happened while the client was absent. In dCache, there is limited support for this using an in-memory circular buffer. This allows a client to receive any events that occurred while disconnected, provided that the client reconnects relatively quickly.

Although SSE provides a mechanism where clients can obtain events, it does not provide an API to allow clients to choose which events are of interest. dCache therefore has implemented a proprietary interface that allows clients to register for an SSE endpoint and subscribe to events.

Internally, dCache uses a pluggable interface to allow the kind of events to be extended. dCache comes with two plugins: a plugin that generates test events at a regular interval and a plugin that provides *inotify*-like events. *Inotify* is an API provided by the Linux kernel that allows programs to learn of changes to the filesystem. The dCache *inotify* plugin closely

follows the Linux *inotify* semantics, allowing clients to subscribe to files or directories and receive events describing activity.

4.3 Comparison between Kafka and SSE

Although both Kafka and SSE-based event notification provide a mechanism for delivering events, the respective functionality is complementary.

Kafka provides access to dCache billing events. The target audience of billing information is dCache administrators or equivalent IT infrastructure building and monitoring roles. dCache provides an abundance of information, some of which may be personal or confidential. Therefore, the Kafka interface is best suited for site-level integration, offering combined services that, together, support that site's community in their scientific research.

This is complemented by SSE. The SSE interface only accepts authenticated users. This, together with a compatible event authorisation model ensures users only receive events they are allowed to see; for example, they cannot learn the existence of files located in directories they are not allowed to list. Because SSE is safe, it is enabled by default to all dCache users, allowing ad-hoc integration, potentially with remote (off-site) services receiving events.

5 Storage event demonstrators

In this section we provide two concrete examples that demonstrate the power of storage events: processing data on ingest and automated replication. These two use cases are useful individually, but may also be combined.

5.1 Processing data on ingest

This section provides a description of data processing on ingest. This is a mechanism where new data is processed automatically; for example, to extract metadata or to build summary or derived data that is then uploaded into the storage system.

This mechanism may be applied quite generally. Any scientific field where new data is processed in some automated fashion may be supported. For example, one natural application is to process data as it comes off of some scientific instrument, such as a telescope, microscope, or other detector.

A specific example of how to support data processing on ingest is shown in Figure 2. In this example, dCache publishes billing events into a Kafka instance. A Kafka agent consumes these events, filtering for those that have the signature of incoming data (e.g., matching target directory and filename). When such an event is found, macaroons [18, 19] are requested that allow the bearer to download the new file, and upload the derived data. A new event that contains macaroon-embedded URLs is sent to a different topic in the Kafka instance.

An OpenWhisk instance provides the Function-as-a-Service (FaaS) computational platform, which is backed by Kubernetes [20] running on top of an OpenStack [21] cluster. The OpenWhisk project includes integration support for Kafka, where a Kafka topic is an OpenWhisk feed and Kafka events are OpenWhisk triggers that OpenWhisk rules then map to function invocations. Using this support, the topic containing the macaroon-URL events activates a containerised function. A prepared (containerised) image downloads the new data, processes it to generate the derived data, and then upload this resulting data.

Note that other computational paradigms may be used to support processing data on ingest. For example, data ingest could trigger submission of a job on a batch farm that downloads the data, processes it and uploads any resulting data.

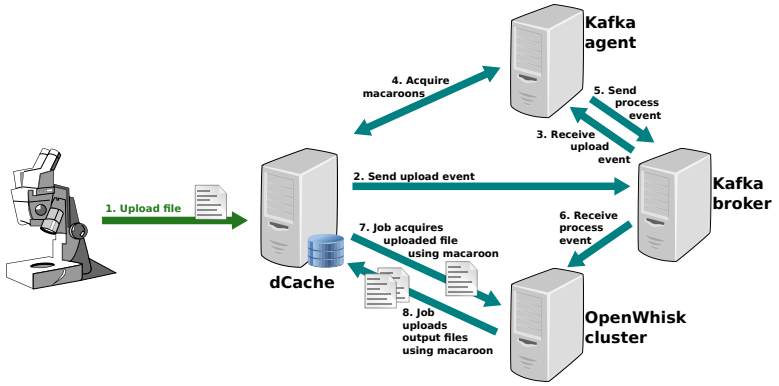

Figure 2. An overview of how to integrate a scalable processing farm that reacts to incoming data. For simplicity, the details of the OpenWhisk deployment are left out.

.

5.2 Automated data replication

This section provides a description of automated data replication. Through this mechanism, a user community can ensure that uploaded data is replicated so as to ensure availability and reduce likelihood of data loss.

The mechanism may be applied more generally. For example, the data management agent could migrate new data away from a storage system, moving data from some instrument-specific storage to more general storage, perhaps located some distance away.

A specific example of how to use storage events to manage data is shown in Figure 3. This example uses Rucio [22, 23] as the data management service which, in turn, uses FTS [24] to orchestrate transfers.

Rucio is a software project that can manage large-scale federated data storage, moving data based on declarative policies. An experimental, new component within Rucio is the panoptes agent. This agent registers itself with dCache for storage events via the SSE interface.

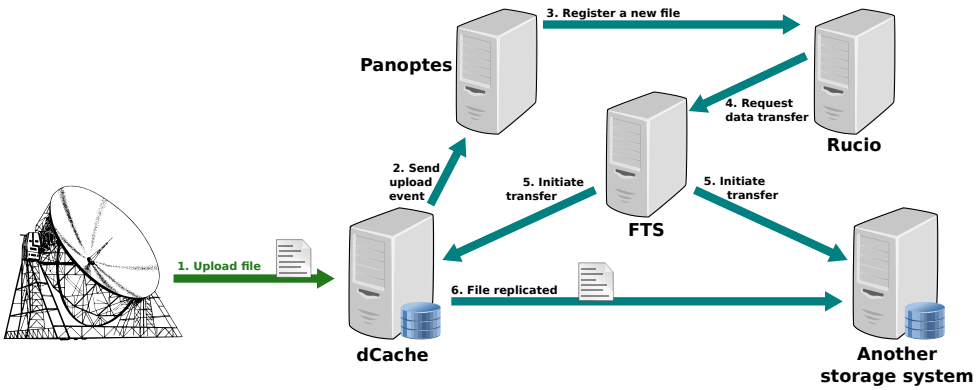

Figure 3. An overview of how to integrate a scalable data ingestion replication with Rucio and FTS. For simplicity, some details within Rucio and FTS are left out.

When data is uploaded, dCache informs the panoptes agent which, in turn, records the new file within Rucio along with the locality policies that apply for this file. In this example, the Rucio policy states that the file should exist on two storage systems. Therefore, once Rucio learns of a new file, it schedules a transfer to the other storage system.

Note that the automated replication is achieved without dCache providing explicit support for Rucio.

5.3 Additional use cases

As mentioned earlier in this paper, there are additional use cases where there is a benefit from adopting storage events, such as avoiding client polling for long-running jobs. Moreover, storage events, as a basic building block, may be combined to build a complex data management and processing system.

As a concrete example, consider a telescope located on top of a mountain that writes data into dCache. The automated replication use case is employed to ensure that any new data is copied from the telescope-local storage to some storage located at a research facility with significant computing resources.

The research facility uses the automated data-processing-on-ingest use case to trigger the analysis of this new data. Site-local computing facilities process the data and generate derived data. This derived data is then written into dCache.

The same automated replication use case then informs the data-management service of the new derived data. This triggers the movement of data off of the storage, copying it elsewhere. This data movement frees up capacity, allowing dCache to receive more data.

6 Conclusions and future work

In this paper we have described storage events, a powerful new concept for managing stored data. The implementation of storage events in dCache was explained, where storage events are available through two complementary channels. The power of storage events was demonstrated through two example use cases: automatic processing of uploaded data and automatic replication of data. These two examples show only a fraction of the use cases that would benefit from adopting storage events.

Future work includes enhancing the SSE support in dCache by adding additional events. These include notification of tape operations, storage QoS transitions, and transfers. We foresee integration with JavaScript, allowing web clients to maintain up-to-date information.

Work within the eXtreme DataCloud project, in collaboration with the EU projects DEEP and EOSC-Pilot, will further explore automated data workflows and investigate bringing automated processing into production.

The initial integration work with the Rucio team will continue. This will involve exploring how SSE-*inotify* integration may be included in Rucio's mainstream distribution, and how the solution may be deployed at scale.

We will also be working with dCache sites to deploy stored events in their production environments, matching their specific needs and use-cases.

Acknowledgements

This work is supported in part by the eXtreme DataCloud (XDC) project. eXtreme DataCloud is co-funded by the Horizon2020 Framework Program – Grant Agreement 777367

References

[1] E. L. Dusseault, *RFC 4918 HTTP extensions for web distributed authoring and versioning (WebDAV)*, Internet Requests for Comments (2007)
[2] P. Hethmon, *RFC 3659 extensions to FTP*, Internet Requests for Comments (2007)
[3] F. Donno, L. Abadie, P. Badino, J.P. Baud, E. Corso, S.D. Witt, P. Fuhrmann, J. Gu, B. Koblitz, S. Lemaitre et al., Journal of Physics: Conference Series **119**, 062028 (2008)
[4] A. Sim, A. Shoshani, *The storage resource manager interface specification version 2.2*, OpenGridForum (2008), https://www.ogf.org/documents/GFD.129.pdf accessed: 2018-11-29
[5] *AWS Lambda*, https://aws.amazon.com/lambda/, accessed: 2018-11-29
[6] *Google Cloud Functions*, https://cloud.google.com/functions/, accessed: 2018-11-29
[7] *Azure Functions*, https://azure.microsoft.com/en-us/services/functions/, accessed: 2018-11-29
[8] *Apache Storm*, http://storm.apache.org/, accessed: 2018-11-29
[9] *Apache Spark*, http://spark.apache.org/, accessed: 2018-11-29
[10] *Apache Kafka*, http://kafka.apache.org/, accessed: 2018-11-29
[11] *Apache Nifi*, http://nifi.apache.org/, accessed: 2018-11-29
[12] *Apache Samza*, http://samza.apache.org/, accessed: 2018-11-29
[13] *Apache OpenWhisk*, http://openwhisk.apache.org/, accessed: 2018-11-29
[14] *Kubeless*, https://kubeless.io/, accessed: 2018-11-29
[15] P. Fuhrmann, V. Gülzow, European Conference on Parallel Processing **1**, 1106 (2006)
[16] A. Millar, T. Baranova, G. Behrmann, C. Bernardt, P. Fuhrmann, D. Litvintsev, T. Mkrtchyan, A. Petersen, A. Rossi, K. Schwank, Journal of Physics: Conference Series **396**, 32077 (2012)
[17] *Server-sent events*, https://www.w3.org/TR/eventsource/, accessed: 2018-11-29
[18] A. Ashish, A. Millar, T. Mkrtchyan, P. Fuhrmann, G. Behrmann, M. Sahakyan, O.S. Adeyemi, J. Starek, D. Litvintsev, A. Rossi, Journal of Physics: Conference Series **898**, 102009 (2017)
[19] A.P. Millar, O. Adeyemi, G. Behrmann, P. Fuhrmann, V. Garonne, D. Litvinsev, T. Mkrtchyan, A. Rossi, M. Sahakyan, J. Starek, 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP) **1**, 651 (2018)
[20] *Kubernetes*, https://kubernetes.io/, accessed: 2018-11-29
[21] *OpenStack*, https://www.openstack.org/, accessed: 2018-11-29
[22] M. Barisits, T. Beermann, V. Garonne, T. Javurek, M. Lassnig, C. Serfon, A. collaboration, Journal of Physics: Conference Series **1085**, 032030 (2018)
[23] V. Garonne, R. Vigne, G. Stewart, M. Barisits, T.B. eermann, M. Lassnig, C. Serfon, L. Goossens, A. Nairz, the Atlas Collaboration, Journal of Physics: Conference Series **513**, 042021 (2014)
[24] A.A. Ayllon, M. Salichos, M.K. Simon, O. Keeble, Journal of Physics: Conference Series **513**, 032081 (2014)