Data Broker: a Case for Workflow Enablement Using a Key/Value Approach

Lars Schneidenbach, Bruce D'Amora, Claudia Misale, Carlos H. A. Costa, Sara Kokkila Schumacher
IBM T.J. Watson Research Center
Yorktown Heights, New York
{schneidenbach,damora,chcost}@us.ibm.com
{c.misale,saraks}@ibm.com

Thomas Ward
IBM Hursley
Hursley, Winchester, UK
TJCW@uk.ibm.com

Abstract

Complex problems can often only be solved with a workflow of different applications whose progress depends also on the need of shared data. There are different challenges that developers must overcome when sharing data in a workflow; each application may have very different requirements in how they access and consume shared data. For example, data access can be online or offline, data sizes and types may vary as well as the frequency they are consumed and produced by applications in the workflow. Moreover, producers and consumers may not run at the same time scale, thus introducing possible latency. Also, the deployment system itself plays an important role, especially when considering low latency read/write operations and reliability in case of failures.

To facilitate the communication of information within a workflow, in this paper we present the Data Broker, a programming model helping applications to share data in the form of named tuples, relying on the concept of *namespaces* providing software-based data isolation. The client API provides access to the namespaces through mainly put/get primitives, enriched also with namespace management functions and key browsing/querying. Our programming model can support different backends for storing/retrieving the data: in this paper, we will focus on the Redis implementation. By using Redis as backend, we enable the Data Broker with the reliability and infrastructure provided by it, which allows users to have access to a fast local or remote in-memory distributed key-value/object store configured as a set of independent instances or a coordinated cluster.

On top of Redis, we implement data access function, namespace management, asynchronous call with client-side queues per server, and key-space browsing.

CCS Concepts • Information systems → Data management systems; Key-value stores; • Software and its engineering → Distributed systems organizing principles; • Computing methodologies → Massively parallel and high-performance simulations.

Keywords HPC, Distributed Systems, Data Management, Key-Value Store, Scientific Workflows

ACM Reference Format:

Lars Schneidenbach, Bruce D'Amora, Claudia Misale, Carlos H. A. Costa, Sara Kokkila Schumacher and Thomas Ward. 2019. Data Broker: a Case for Workflow Enablement Using a Key/Value Approach. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '19)*, September 30–October 3, 2019, Washington, DC, USA. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3357526.3357572>

1 Introduction

Large scale cloud and traditional data centers offer increasingly greater computing power, memory, network bandwidth, and storage. Large scale systems are being used to solve problems in engineering and science that could not be explored with previous systems. Complex phenomena in physics, chemistry, biology, finance and business analytics, are not easily modeled with a single monolithic application. Application maintenance, portability, and extensibility present additional development challenges [15, 23, 27]. Rather than develop single applications, developers can design a workflow represented by a set of applications, each solving part of the problem and sharing their results with one or more of the other applications composing the workflow. This approach provides several benefits over the traditional monolithic application approach:

- Composability

The workflow may consist of different simulation, AI, analytics, and visualization applications. For example, different problems may require replacing an existing simulation application with a different one that has

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

MEMSYS '19, September 30–October 3, 2019, Washington, DC, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-7206-0/19/09...\$15.00

<https://doi.org/10.1145/3357526.3357572>

been developed for the particular domain being explored.

- **Extensibility**

Adding new functionality to a workflow can be accomplished by adding a new application rather than modifying an existing one, which may have design constraints preventing the required extensions.

- **Cloud Native Portability**

Cloud native systems are optimal for microservice deployment. A complex workflow can be decomposed into a set of applications and services that can be containerized and moved to the Cloud [25, 26, 30]. For example, many workflows have a visualization component that can be deployed as a service and shared by multiple workflows.

- **Reusability**

A workflow composed of multiple applications, each targeted for a specific purpose, results in a higher probability of reusing applications across different workflows.

- **Maintainability**

Generally, smaller applications are easier to maintain than large monolithic ones, with complex conditional logic.

Although there are benefits to developing workflows versus single applications to solve complex problems, there are also challenges. Multiple applications composing a workflow need to be coordinated and synchronized in how they produce and consume data. A traditional approach is to leverage posix file I/O to share data between the applications. There are scenarios where this is not only appropriate but potentially optimal. For example, for very large scale data exchanges, i.e terabytes to exabytes, file I/O is optimal. Other cases where applications are leveraging a pre-defined data model such as HDF5 [28] or netCDF [19], it may be a simpler solution to leverage the APIs and tools that exist to manipulate the data. But when there are communication latency requirements between producers and consumers, finer granularity of data exchanges, multiple applications consuming the same data, and/or system limitations on the number of files that can be created, other solutions are needed. An alternative to posix file I/O, is to establish point to point connections between applications that require exchange of data. This can be represented as MPMD or Multi-Program Multi-Data design pattern and can be accomplished by establishing direct socket connections or MPI programming paradigm. In this case, the workflow can leverage the network to exchange data between applications rather than the longer latency, smaller bandwidth of file I/O. There are disadvantages to this approach: 1) Coordination of multiple applications, where multiple socket connections or MPI sub-communicators need to be managed between multiple applications. This can be cumbersome and complex if there

are many applications sharing the same data; 2) Dynamic adding and removing of applications, where adding applications to the workflow is difficult as connectivity has to be known a priori; 3) Resilience of single or multiple application failures, where application failures will cause the entire workflow to fail.

In this paper, we introduce a programming model, the Data Broker, for applications to exchange data in the form of named tuples, relying on the concept of namespaces to provide software-based data isolation. Interacting applications in workflows can access data in namespaces through put/get primitives. The Data Broker provides access to a shared memory space where multiple processes can share data both synchronously and asynchronously. The advantages of the proposed system are the following:

- High level programming model hiding low level communication and connectivity semantics
- Multiple programming APIs that can be integrated into applications
- A common simple data model
- Simplified synchronization and communication among processes
- API supporting both synchronous and asynchronous operations
- An architecture that can support multiple data stores including hardware acceleration
- Fast in-memory distributed data storage

The Data Broker supports different back-end data stores. In this paper, we focus on Redis [7], an in memory data structure store, used as a database, cache and message broker, which is used as one of the backend options for the Data Broker. With Redis, we are able to provide an in-memory, lightweight back-end data store which is scalable, offers backup/recovery, and failover features, as well as automatic resharding of data. Redis supports cluster mode distributing a set of Redis servers over a number of compute nodes with each server managing a portion of the key-value hash slots. Applications have access to all the Redis server instances transparently to the user via the Data Broker programming model. The Data Broker adapter for Redis supports Redis single node and cluster mode deployment options transparently to the user. The implementation supports some cases of fail-over recovery.

With Data Broker, it is possible to build a customizable in-memory store, which can be either local or remote, paired with an easy-to-use client API. The advantage of that is twofold. First, having an in-memory store it is possible to take advantage of low latency offered by read/write operations in memory. Second, it is possible to customize the placement and configurations of the back-end store servers, in order to place them close to processes accessing them, and hence take advantage of a reduced network traffic.

The paper is organized as follows. Section 2 provide an overview of the efforts made in the community to support workflows and applications for efficient data sharing, coordination and communication, and on object/key-value storage. In Section 3 we define and describe the Data Broker architecture and implementation, as well as the semantics of its client API. An example of its execution model is described in Section 4. Preliminary benchmark results and a use case are exposed in Section 5. Finally, section 6 concludes the paper, describing also the future directions we are taking.

2 Related Works

In this section we provide an overview of the efforts made in the community to support workflows and applications for efficient data sharing, coordination and communication, targeting distributed systems and large scale deployments. We will focus on frameworks providing communication and coordination primitives to workflow developers, and on object/key-value storage.

Data sharing and coordination for parallel applications running in distributed systems has always been a challenging research topic, coupled with the need of providing a high level APIs easing the development of parallel and distributed applications. The use of distributed data structures and a programming model resembling programming in shared memory, would be the most natural way. We can find implementations of Partitioned Global Address Spaces (PGAS), such as GASNet project [13] as a language and compiler translating high level annotations into RDMA data transfer routines. The programming model is not generic, but specific to SPMD applications (i.e., MPI programs). The Data Broker provides a backend implementation based on GASNet, so that it is possible to enable named tuple exchange on a global address space, but we remark that the client API provided by the Data Broker is independent from the programming model.

An alternative to message passing and global address spaces has been proposed in the Linda programming model [14]. Linda can be defined as a coordination and communication model, where parallel processes read and write objects into a shared, virtual, associative memory. Linda is basically a *coordination language* providing primitives operating on ordered sequence of typed data objects called *tuples*. Such tuples are managed by the user with a set of primitives to store into and retrieve from a logically global associative memory, called a *tuplespace*. The tuplespace concept is hence an abstraction of distributed shared memory with several minor distinctions (destructive and non-destructive reads, and different coherency semantics are possible).

One of the motivations behind Linda, is to provide a simplified, high level, general approach for applications in distributed system to communicate by sharing both data and

coordination commands. The Data Broker can be considered an implementation of the Linda programming model, where the tuplespace concept is now the *namespace*, and it can be instantiated on top different memory models (i.e., shared memory, distributed object/key-value stores, hardware based, global address space), provided a backend exists.

An interesting work has been done to try to define a standard for interoperability among high-performance scientific application components, which may be work together in workflows, namely Common Component Architecture (CCA) [12]. The group, composed mainly by various Department of Energy labs and academics institutions, recognized the complexity and challenges arising from enabling interoperability between heterogeneous, HPC applications, and they drove their research focusing on the fact that scientific community needed a formal and standard way to manage the complexity of multidisciplinary simulation workflows, and also a way to address scalability and performance issue when running on parallel and distributed systems. Applications can be modeled as components implementing *Provides* and *Uses* service *Ports*, which are in turn connected at runtime to specify the dependencies and interactions among applications. However, this work does not address how to actually exchange data, and it is left to the implementation.

DataSpaces [16] is a framework providing high level interaction and coordination abstractions that can support dynamic and asynchronous data-intensive coupling patterns required by application workflows. It provides the abstraction of a specialized, distributed, virtual shared space accessible by workflow components and services. DataSpaces has been embedded in ActiveSpaces framework [17] to give an higher level of abstraction to the end user. ActiveSpaces is a data management framework that allows the application developers to define data-processing routines, deploy and execute them in the staging area built on top of the DataSpaces framework.

Applications can store and retrieve data objects at runtime using semantically meaningful descriptors (e.g., geometric regions in a discretized application domain). Its programming model is based on the concept of processing routines, namely *data kernels*, that can be deployed and executed on staging areas managed by DataSpaces. The binary code associated with data kernels is transported by the runtime to the staging areas and executed there in parallel, where they access the data directly. After executing a data kernel, result is sent back to the application. Similarly to the Data Broker, ActiveSpaces provides applications a way to collaborate and share data at runtime by insert data in the shared space, but it also provide a way to offload computation to the data, thanks to a runtime supporting it. At the current implementation status, the Data Broker does not implement yet remote execution, but this is a feature that is already in the API specification.

Today, both industry and academia frequently use remote in-memory key-value or object store to share data and potentially coordinate the execution flow across multiple applications. Examples of such in-memory store are Redis [7], Memcached [5], and others [1–3, 6, 18, 20–22]. A key property of those in-memory store is that they provide a simple API based mainly on put/get primitives, but more importantly they provide very high throughput and low latencies remote operations on the actual storage and provide automatic recovery and auto-sharding of data. Those are also aspects we considered when choosing an object store as backend.

In a recent work [11], authors present an alternative approach to remote in memory key-value stores when used as cache and/or store for stateless applications. The alternative model they propose is named *Link*, linked in memory key-value store, that is a stateful application server model, which is tightly coupled to the applications that needs to access data. Their claim is to make developers build *stateful application servers*, that is, servers that couple application code and long-lived in-memory state in the same process, and to build a custom in-memory store when application servers are not feasible (i.e., when state is shared by multiple applications or languages). The custom in-memory store would expose a domain-specific API and leverage the fact of being at network distance.

Instead of taking a single direction (either local or remote store), the Data Broker is acting like a stateful application server, but it can use a remote store locally, hence avoiding the implementation of resharding, high availability, consistency, etc., and providing a unified API that allows the user to choose among different backends without the need of re-implementing their applications.

3 Data Broker

We define the Data Broker (DBR) as a programming model to help applications, which are executing as part of a workflow, share data and exchange messages in the form of named tuples, hence abstracting out the underlying data model. The DBR relies on the concept of *namespaces* to provide software based isolation so that processes can access namespaces synchronously and asynchronously as well as online and offline. This approach makes it possible to enable any producer/consumer pattern. Each namespace is a *multiset* of named tuples stored in FIFO order (Fig. 1). The DBR provides an application programming interface (API) to create and manage the underlying namespaces from the user applications.

Architecturally, the DBR is a layer of middleware that exists between the application and an in-memory data store of named tuples organized by namespace. The DBR design points allow for support of different data stores via the DBR system library. For example, any key-value store, object store as well as software-based global address space can be supported by writing a DBR system library that maps the DBR

Figure 1. Representation of a Namespace.

client API to the backend data store (see Sec. 3.1). This architecture facilitates support for different data backing stores without recompiling the application.

Performance and reliability of the Data Broker relies both on the middleware and on the backend’s runtime. As stated earlier, in this paper we are focusing on the Redis based implementation.

Figure 2 shows a typical utilization of the Data Broker. In the following, we demonstrate how to use the Data Broker, by showing code snippets implemented with the Python client API binding.

```

1 dbr_name = "DBRtestname"
2 dbr_hdl = ffi.new('DBR_Handle_t*')
3
4 # attach to or create namespace
5 dbr_hdl = dbr.attach(dbr_name)
6 if dbr_hdl == None
7     level = dbr.DBR_PERST_VOLATILE_SIMPLE
8     group_list = ffi.new('DBR_GroupList_t')
9     dbr_hdl = dbr.create(dbr_name, level,
                          group_list)

```

Listing 1. Creation or connection to a namespace with the Data Broker.

Every access (read or write) to a tuple requires a valid namespace. A valid namespace can be either created, or a process can attach to an existing one (`dbr.create()` and `dbr.attach()`, respectively). Namespaces must be unique, that is, a namespace creation will fail if there is already an existing namespace with the same name. Functions for creating or attaching to a namespace, will call their corresponding implementation in the system library backend adapter. We can consider those calls, `dbr.create()` and `dbr.attach()`, as `cmd` when referring to fig. 2, and they take actions against the namespaces. A new namespace is hence defined by a

Figure 2. Communication pattern with the Data Broker.

unique name, a persistence level¹ and a list of groups². Persistence levels and groups are related to the backend. In the case of Redis, those can be set to zero.

```

1 test_in = "Hello World!"
2 group = dbr.DBR_GROUP_EMPTY
3 res=dbr.put(dbr_hdl, test_in, "testTup", group)
4
5 value,ret = dbr.read(dbr_hdl, "testTup", "",
6     group, dbr.DBR_FLAGS_NONE)
7 print('Read returned: ' + value + " - status: "
8     + dbr.getErrorMessage(ret))
9
10 value,ret = dbr.get(dbr_hdl, "testTup", "",
11     group, dbr.DBR_FLAGS_NONE)
12 print('Get returned: ' + value + " - status: "
13     + dbr.getErrorMessage(ret))

```

Listing 2. Example of storing and retrieving data from a namespace.

The code snippet 2 shows how to put, read and get a simple tuple. In this example, all the calls are blocking. The tuple will be added to the existing namespace and, if there exists already a tuple with the same name, the tuple data will be added to that tuple as a kind of new version of data. Functions storing and retrieving to/from a namespace, will call their corresponding implementation in the System Library back-end adapter. We can consider those calls as data when referring to fig. 2, and they take actions against the tuples in namespaces. Details on the implementation at the System Library level can be found in Section 4.1.

Eventually, an application will either detach from the Data Broker, or delete the instance. Any process that is attached to a namespace can detach from it using `dbr.detach`. The

¹Define the level of persistence of tuples in DBR. It can be in memory or persistent storage depending on the back-end data store chosen. Does not apply to Redis.

²Defines the list of backing memory units (a unit of backing memory can be any slice of memory in RAM, Flash, other NVM,...) that can be used by the DBR in the created namespace.

Figure 3. Architecture stack of the Data Broker.

Data Broker internally uses a reference counter to check how many processes are attached to a namespace, and a detach call will decrement the counter.

```

1 print('Delete Data Broker')
2 res = dbr.delete(dbr_name)
3 print('Exit Status: '+dbr.getErrorMessage(res))

```

Listing 3. Final detach and deletion of a Data Broker instance.

When the user issues a `dbr.delete` (listing 3), the Data Broker will mark the namespace as *erasable* and check the reference counter: when 0, then the deletion will proceed and clean up all remaining data of that namespace. If there are still clients connected to that particular namespace, the last detach will also issue a clean up of all remaining data and then delete the namespace.

3.1 Architecture

Figure 3 shows a representation of the DBR architecture stack. We can identify three main layers acting: Client Library, System Library and Back-end Storage. At the top level, the Client Library exposes the Data Broker API to the user,

which is used to access the Data Broker from the application. C and Python API bindings are available. Also the System Library is exposed to the user through its API, and it offers the interface to implement other back-end adapters. The next paragraphs provide more details about the architecture.

Data Broker Client Library The *Client API* level is exposed to the user who wants an application to communicate via the DBR. The user API exposes **management** and **access** functions. Management functions are intended to let the user create, destroy, manage one or more namespaces, while access functions are intended to access the namespace content to read and write tuples, in both blocking and non-blocking fashion. It also provide Scatter-Gather API extensions. Tables 1 and 2 report the functionalities supported in the API.

Table 1. List of functions for managing DBR and underlying namespaces.

Data Broker Management Functions	
dbrCreate	Create a new named Data Broker instance with an associated empty namespace
dbrDelete	Delete an existing Data Broker and its namespaces
dbrAttach	Attach to an existing Data Broker by name
dbrDetach	Detach from an existing Data Broker by name
dbrQuery	Query information about an existing Data Broker
dbrTest	Test the status of an asynchronous call
dbrCancel	Cancel an asynchronous call

Table 2. List functions to read and write on a namespace of the DBR.

Data Broker Access Functions	
dbrPut/PutA	Writes a named tuple into a namespace, blocking/non blocking
dbrRead/ReadA	Performs a non-destructive read from namespace, blocking/non blocking
dbrGet/GetA	Performs a destructive read from namespace, blocking/non blocking
dbrMove	Move a named tuple from a source namespace to a destination namespace
dbrDirectory	List keys in a namespace with a given pattern
dbrTestKey	Check the existence of a tuple by its key/-name

Table 2 shows the access functions to the Data Broker. Let us consider, for instance, reading a tuple t from a namespace given a key k from the namespace ns in a DBR (DBR_{ns})

running instance. The function `read` takes in input a key k to be searched and a value t to be filled with the value copied from DBR_{ns} (thus `dbrRead` is not destructive). The call does not return until a value for key k becomes available in the namespace or a timeout is reached, hence the `read` call is synchronous and blocking. An asynchronous implementation `dbrReadA` call, has the same semantics of `dbrRead` in the sense that it is not destructive, but the function returns immediately after posting the call. The completion of any asynchronous read can be checked with the `dbrTest` function.

Data Broker System Library The System Library provides the implementation of the mapping between each DBR API call and the backend. It provides an API to implement additional back-end adapters. Any back-end adapter is required to implement 6 API calls that cover functionality for posting, testing, and cancelling user requests as well as initialization and termination of any backend state as needed. The post and test operations should be non-blocking to allow the user library and the application to continue in case the user requires a non-blocking operation.

Our main focus was on an implementation for Redis. An additional proof of concept implementation was created for GasNet. It stores data in the memory of a distributed GasNet process. There are further plans for a hardware accelerated backend of the Data Broker library.

4 Implementation

The Data Broker Redis adapter is implemented on top of the Redis Modules API to guarantee best performance and because we only need a small subset of the rich Redis API.

To store the tuple data, the `list` type and API of Redis is used because it provides the FIFO functionality required to implement the semantics for tuples with the same key. The user issuing a read on a specific key k will trigger a read of the head element of the list associated to k . Similarly, the user issuing a get will trigger a pop command on the top of the list. The implementation is guaranteed to be race free by the Redis runtime, which serializes the client requests arriving from different clients. There is no ordering guarantee for requests. This applies to requests from both a single client and from multiple clients.

While the natural mapping for namespaces to Redis would be the database, it was not possible to implement it this way because the number of databases is limited to 16 and Redis cluster mode only supported one database. Therefore, the current backend uses the namespace as a prefix for the keys.

4.1 Execution Model Example

In this section, we provide a simplified example of the execution model of the Data Broker runtime, when a `dbrGet` call is issued to a distributed Redis Cluster backend instance. We can partition the runtime in two parts: *Sender* and *Receiver*

processing components³. The *Sender* component handles any data and other requests that need to be sent to the Redis, while the *Receiver* component manages and parses any responses from Redis.

User issues the call The client library pushes the `dbrGet` request from the user space into the work queue, namely the `RequestQueue`, managed by the *Sender*. The *Sender* then checks the internal `RetryQueue` for requests that are uncompleted yet: those might be repeated, multi-stage or redirected requests that are pushed by the *Receiver* for their completion. The `RetryQueue` has priority over the `RequestQueue` to make sure pending requests complete before additional request get processed. However, the *Sender* is trying to prepare and send multiple requests to reduce the amount of switching between *Sender* and *Receiver*. Therefore, new user requests have a high chance of getting processed even with several incomplete requests pending in the `RetryQueue`.

Request creation The *Sender* analyzes the request and determines which connection to use, i.e., to which Redis server the request has to be sent. The *Sender* component manages an internal hash-map storing the mapping between Redis server IDs and their IP addresses. The CRC16-based hash value of the key is mapped to the Redis server connection where the key might be retrieved. The *Sender* is now in charge of creating the protocol string based on Redis protocol syntax. Instead of assembling the string in a contiguous buffer, it is assembled as a list of scatter-gather elements (SGE) also known as `struct iovec` from Posix file I/O. This helps to avoid unnecessary copies of the user data⁴. See [9] for details on the Redis communication protocol. As an example of a protocol string sent to Redis, let us consider retrieving and consuming the tuple having *Ziggy* as its key. The string will be:

```
*2\r\n$4\r\nLPOP\r\n$5\r\nZiggy\r\n
```

Where `*2` declares the command is composed as an array of two elements: the first entry is the actual command `LPOP` and the second part is the key both represented as a Redis bulk string. A bulk string starts with `$` followed by the length, the separator/termination `\r\n`, the text and another termination symbol. Once the request has been created, it is sent to the correct Redis server.

Check for responses The *Receiver* component of the Data Broker runtime checks for request messages from any active connections to Redis servers by multiplexing activity via `libevent` [24]. Those request messages can be part of a multi-staged request that needs more stages to be completed. In

this case, the request stage is processed and then pushed to the `RetryQueue` to hand it over to the *Sender* component. Other reasons for returning a request to the send queue can be:

- A redirect (ASK) or relocate (MOVED) response from Redis, so the hash-to-connection mapping has to be updated and the request has to get sent to the new location.
- A get or read request has not found an entry. This requires the system to retry until either the data becomes available or a timeout is met and the request gets canceled.

If the request is complete, or it caused an error, it is pushed into a completion queue and the completion is fetched and processed by the client library.

4.2 Redis Single Node and Cluster Mode

It is possible to deploy Redis in different ways: single node or cluster mode, where each server can be deployed with replicas for redundancy. Single node mode is straightforward: it deploys a Redis server instance on a host. An application can attach to the Redis server via an IP address and port. Each Redis server is independent from the others, i.e. each server manages all the hash slots. Redis cluster deploys a multi-node Redis server installation where data is automatically sharded across the nodes. Each server may have replicas for redundancy, and in this case the servers are called *masters*, while replica servers are considered *slaves*. The servers do not forward requests to the correct location in case a client contacts the wrong Redis instance. Instead the client receives a response containing the information about the server to contact. It is up to the client to create and maintain a mapping or always rely on the redirect response from the server.

Adding replica servers to each of the Redis masters improves the availability and robustness of the data backend. However, unless configured to return after data has been replicated, there is no guarantee that written data is preserved in case of a server crash⁵.

The Data Broker adapter for Redis covers all of the above deployment options transparently to the user. The minimum requirement is to provide one location of a Redis server. After that, the Data Broker library will retrieve the other locations from that server including any replica information. It will re-route and any client requests to the correct Redis server instance automatically.

The implementation supports some cases of fail-over recovery by trying to recreate broken connections, retrieving updated Redis cluster information, and switching connections to replicas. For cloud deployments of Redis, the system library is able to stay connected to a cluster that gets re-sharded or changes its size (scale up/down).

³Currently sender and receiver are processed in the same thread as the calling application. The two might become separate threads later.

⁴Complete avoidance of copies might depend on the `libc` implementation of `sendmsg` or the OS kernel. Each might create a contiguous send buffer from the SGE anyway. This was not the case in our test environment since we measured improvements of the write throughput.

⁵This configuration setting comes with a significant performance penalty.

4.3 libfuse integration

We implemented a libfuse (File System in Userspace) server [4]. The following is simplified source code snippet demonstrating writing to Data Broker through the POSIX file system API and reading the same data using the Data Broker API (listing 4).

```

1 // File write/databroker read test case
2 #include <libdatabroker.h>
3 // #include ...
4 #define NAMESPACE "interop2"
5 #define KEY "mykey2"
6 // write data on a file system mount point
7 void fs_write(){
8     int fd=open("/tmp/mnt/"NAMESPACE "/"KEY,
9         O_WRONLY|O_CREAT,S_IRWXU);
10    if (fd>=0){
11        char *buffer="mydata";
12        ssize_t actlen=
13            write(fd,buffer,strlen(buffer));
14        close(fd);
15    }
16 }
17 // issue a dbrRead from the namespace
18 void db_read(){
19     //init connection to Data Broker
20     //buffer init
21     DBR_Errorcode_t ret =
22         dbrRead(nspace_hdl,buffer,&buflen,key,
23             "",0,DBR_FLAGS_NONE);
24     fprintf(stdout,"buffer=%s - status %d\n",
25         buffer, ret);
26     dbrDetach(nspace_hdl);
27 }

```

Listing 4. Example of a read/write operation over libfuse with Data Broker.

A libfuse server is a server which runs on the same node as the application which makes the POSIX IO calls, together with a kernel extension which passes the data to this server. Each POSIX IO syscall results in a function call in this server.

The libfuse server is linked with the DBR library. With the Redis backend, there are then one or more Redis servers as described in Section 4.2.

5 Use Case and Experiments

In this section we provide a description of MuMMI, a workflow focusing on molecular dynamics simulations driven by machine learning, in which we used the Data Broker to enhance communication. We also provide a set of benchmarks we conducted to compare Data Broker performance against the official benchmark suite provided by Redis.

Figure 4. MuMMI workflow

5.1 MuMMI Workflow Example

The Data Broker facilitates data sharing in complex workflows. One example is the use of the Data Broker in the Multiscale Machine-Learned Modeling Infrastructure (MuMMI) workflow. MuMMI uses machine learning to couple macro scale biological simulations with high-fidelity molecular dynamics (MD) simulations. Through the use of the Data Broker for data management and a workflow manager, MuMMI connects multiple software components running simultaneously to create a single, massively parallel, biological simulation.

Multiple components in MuMMI share data via the Data Broker. Figure 4 outlines the basic workflow of MuMMI, where each box represents multiple software components. A typical MuMMI simulation uses 2040 nodes and the job layout is optimized to use the *Sierra* supercomputer. *Sierra* is the next high performance computer system developed for the NNSA ASC program and located at Lawrence Livermore National Laboratory. Developed by IBM with POWER 9 and Nvidia GPU technology, it can achieve peak 121 PFLOPs, with a total of 4320 compute nodes, each configured with two IBM POWER 9™ CPUs, four Nvidia Tesla™ V100 GPUs.

The macro scale simulation uses the Moose framework in coordination with ddcMD. MOOSE posts a data message via the Data Broker to ddcMD, which signals ddcMD to begin the next computation. Once ddcMD finishes its computation, it uses the Data Broker to post a response to the MOOSE framework which then proceeds to its next simulation step.

Output from the macro scale model is analyzed by the machine learning tools and selections from the macro scale model are ranked to create a priority queue and are shared with the workflow manager via the Data Broker. As resources become available, the workflow manager embedded in MuMMI, uses the ranking information to schedule micro scale setup, micro scale production simulations, and micro scale analysis jobs. The micro scale setup uses Gromacs [10] to generate initial MD configurations for the production level micro scale simulations. All of the data generated by the production level micro scale simulations is stored in the data broker and a backup is saved to IBM® General Parallel

File System (GPFS^{TM}). This data is accessed by the micro scale analysis scripts while the simulation is running.

Additionally, the Data Broker is restarted after the MuMMI application is completed for post analysis of the micro scale simulations stored in the Data Broker. Finally, the analyzed micro scale simulation data is used to provide feedback to update the macro scale simulation parameters. Overall, the typical key size used in MuMMI was 50 bytes and the value size ranged from 0.3 KB to 10 MB.

5.2 Microbenchmarks

While the current implementation has not been thoroughly optimized for performance yet, we conducted a number of experiments to show IOPS and throughput for a range of data sizes (see Figures 5 and 6). Benchmarks were executed on a cluster of IBM Power9^{TM} diskless nodes, each equipped with six Nvidia Volta^{TM}, 512GB DRAM, and two Mellanox ConnectX-5 EDR Infiniband network adapters. Nodes are network attached to the IBM Elastic Storage servers configured with IBM Spectrum Scale parallel filesystem to provide 1.5PB of storage. The results were measured by running 100 Redis servers spread evenly over 10 nodes. The parallel benchmark application was run on the same 10 nodes using 40 MPI processes. The start and end of the measurements are synchronized between the client processes and the results are accumulated from all processes.

Figure 5. IOPS performance for dbrPut and dbrGet.

5.3 Comparison to Redis Benchmark

Redis includes the redis-benchmark [8] utility to measure performance of basic commands and simulate a number of different scenarios. Because of limitations of the Redis benchmark to work in cluster mode, we compare only the baseline performance when running a single client process with a single connection to a single Redis server and no request pipelining. Figure 7 shows the IOPS comparison between Redis and the Data Broker on top of Redis. The results indicate that the overhead introduced by the library

Figure 6. Throughput for dbrPut and dbrGet.

is marginal and performance is on par with what the Redis server provides.

Figure 7. IOPS Comparison of Redis and Data Broker.

6 Conclusions and Future Works

In this paper we presented the Data Broker, a programming model helping applications to share data in the form of named tuples, relying on the concept of *namespaces* providing software-based data isolation. The client API provides access to the namespaces through put/get primitives, enriched also with namespace management functions and key browsing/querying. By using Redis as backend, we enable the Data Broker with the reliability and infrastructure provided by the backend, which allows users to have access to a fast local or remote in-memory distributed key-value/object store configured as a set of independent instances or a coordinated cluster.

Currently, we are exploring several potential enhancements to the Data Broker. We have done initial prototyping with libfuse to provide a posix file I/O interface to the Data Broker for applications needing to maintain posix I/O semantics, but achieve the potential speedups of exchanging data in an in-memory key value store.

Integration with Data Analytics Pipelines In emerging workflows, rich and scalable data analytics support for in-situ analysis is key to enable real-time extraction of actionable information and post-processing of large amounts of data. While the open source big data analytics ecosystem provides a variety of analytics frameworks, its evolution with distinct design goals and software technologies makes the integration of different tools into a pipeline challenging.

We are leveraging the Data Broker as a mechanism to efficiently integrate popular data analytics tools into heterogeneous workflow. We are focusing on Apache Spark and its vast ecosystem of libraries for data processing and machine learning; Pandas, the Python data analysis library; and Jupyter Notebook, a front-end for interactive computing.

With the Data Broker, we create a shared and distributed data space based on a standardized language-independent in-memory format (based on Apache Arrow [2]) that allows efficient data exchange among different computing and interactivity tools.

Object Storage Integration In preparation for deployment in public or private cloud systems, we are exploring the ability to map key-value storage to object storage frameworks that are deployed by the majority of public and private cloud providers. Our future work involves evaluating data broker performance while using object stores as backend storage. One of the most popular object storage system CEPH [29], seems useful and compatible to Data broker since it separates data from metadata and uses key/value access mechanism to distribute it over multiple nodes. In addition, using object storage will bring many advantages to the system, to name a few, the easy scalability of the system and reliability of data.

Cloud Deployment As future work, we are considering to create an infrastructure to make easy and automatic the deployment of Data Broker and Redis (either cluster or single node) on cloud platforms. Moreover, we will provide more API bindings (e.g., Golang) to allow more application to use the Data Broker.

Acknowledgments

This work has been supported under the CORAL contract with Lawrence Livermore National Laboratory, LLNS Subcontract No. B604142.

We would like to thank the following people who contributed to the success of this project.

IBM T.J. Watson Research Center Data Centric Systems team: Philip Jacob, Philip Strenski, Charles Johns, Yoonho Park, Hui-fang Wen, James Sexton.

Lawrence Livermore National Laboratory RAS Project Simulation team:

H. Bhatia, F. Di Natale, J. N. Glosli, H. I. Ingolfsson, T. Oppstrup, F.H. Streitz.

References

- [1] 2019. Amazon DynamoDB. <https://aws.amazon.com/dynamodb/>.
- [2] 2019. Apache Arrow. <https://arrow.apache.org/>.
- [3] 2019. FoundationDB. <https://www.foundationdb.org/>.
- [4] 2019. Libfuse Filesystem in Userspace. <https://github.com/libfuse/libfuse>.
- [5] 2019. Memcached. <https://memcached.org/>.
- [6] 2019. Microsoft Azure Cosmos DB. <https://docs.microsoft.com/en-us/azure/cosmos-db/>.
- [7] 2019. Redis. <https://redis.io/>.
- [8] 2019. Redis Benchmark. <https://redis.io/topics/benchmarks>.
- [9] 2019. Redis Protocol Specification. <https://redis.io/topics/protocol>.
- [10] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. 2015. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX* 1-2 (2015), 19 – 25. <https://doi.org/10.1016/j.softx.2015.06.001>
- [11] Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl. 2019. Fast key-value stores: An idea whose time has come and gone. In *HotOS XVII*.
- [12] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. 1999. Toward a common component architecture for high-performance scientific computing. In *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No.99TH8469)*. 115–124. <https://doi.org/10.1109/HPDC.1999.805289>
- [13] Dan Bonachea, Paul H. Hargrove, Michael Welcome, and Katherine Yelick. 2009. Porting GASNet to Portals: Partitioned Global Address Space (PGAS) Language Support for the Cray XT. (5 2009).
- [14] Nicholas J. Carriero, David Gelernter, Timothy G. Mattson, and Andrew H. Sherman. 1994. The Linda Alternative to Message-passing Systems. *Parallel Comput.* 20, 4 (April 1994), 633–655. [https://doi.org/10.1016/0167-8191\(94\)90032-9](https://doi.org/10.1016/0167-8191(94)90032-9)
- [15] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. 2017. A Characterization of Workflow Management Systems for Extreme-Scale Applications. *Future Generation Computer Systems* accepted (2017). Funding Acknowledgments: DOE DE-SC0012636, DE-AC52-07NA27344.
- [16] Ciprian Docan, Manish Parashar, and Scott Klasky. 2012. DataSpaces: an interaction and coordination framework for coupled simulation workflows. *Cluster Computing* 15, 2 (01 Jun 2012), 163–181. <https://doi.org/10.1007/s10586-011-0162-y>
- [17] Ciprian Docan, Fan Zhang, Tong Jin, Hoang Bui, Qian Sun, Julian Cummings, Norbert Podhorszki, Scott Klasky, and Manish Parashar. 2015. ActiveSpaces: Exploring Dynamic Code Deployment for Extreme Scale Data Processing. *Concurr. Comput. : Pract. Exper.* 27, 14 (Sept. 2015), 3724–3745. <https://doi.org/10.1002/cpe.3407>
- [18] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (USENIXATC'10)*. USENIX Association, Berkeley, CA, USA, 11–11. <http://dl.acm.org/citation.cfm?id=1855840.1855851>
- [19] Harry L. Jenter and Richard P. Signell. 1992. NetCDF: A Public-Domain-Software Solution to Data-Access Problems for Numerical Modelers.
- [20] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 121–136. <https://doi.org/10.1145/3132747.3132764>
- [21] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 137–152.

- <https://doi.org/10.1145/3132747.3132756>
- [22] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33, 3, Article 7 (Aug. 2015), 55 pages. <https://doi.org/10.1145/2806887>
  - [23] Idafen Santana Pérez and María S. Pérez-Hernández. 2015. Towards Reproducibility in Scientific Workflows: An Infrastructure-Based Approach. *Scientific Programming* 2015 (2015), 243180:1–243180:11. <http://dblp.uni-trier.de/db/journals/sp/sp2015.html#PerezP15>
  - [24] Niels Provos and Nick Mathewson. 2019. Libevent—an event notification library. <http://libevent.org/>.
  - [25] Cristian Ramon-Cortes, Albert Servén, Jorge Ejarque, Daniele Lezzi, and Rosa M. Badia. 2018. Transparent Orchestration of Task-based Parallel Applications in Containers Platforms. *Journal of Grid Computing* 16, 1 (01 Mar 2018), 137–160. <https://doi.org/10.1007/s10723-017-9425-z>
  - [26] Kyle M. D. Sweeney and Douglas Thain. 2018. Efficient Integration of Containers into Scientific Workflows. In *Proceedings of the 9th Workshop on Scientific Cloud Computing (ScienceCloud'18)*. ACM, New York, NY, USA, Article 7, 6 pages. <https://doi.org/10.1145/3217880.3217887>
  - [27] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields. 2014. *Workflows for e-Science: Scientific Workflows for Grids*. Springer Publishing Company, Incorporated.
  - [28] The HDF Group. 1997–2019. *Hierarchical Data Format, version 5*. <http://www.hdfgroup.org/HDF5/>.
  - [29] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-performance Distributed File System. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 307–320. <http://dl.acm.org/citation.cfm?id=1298455.1298485>
  - [30] Charles Zheng and Douglas Thain. 2015. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing (VTDC '15)*. ACM, New York, NY, USA, 31–38. <https://doi.org/10.1145/2755979.2755984>