Object Proxy Patterns for Accelerating Distributed Applications

J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Alexander Brace, André Bauer, Kyle Chard, and Ian Foster, *Fellow, IEEE*

***Abstract*—Workflow and serverless frameworks have empowered new approaches to distributed application design by abstracting compute resources. However, their typically limited or one-size-fits-all support for advanced data flow patterns leaves optimization to the application programmer—optimization that becomes more difficult as data become larger. The transparent object proxy, which provides wide-area references that can resolve to data regardless of location, has been demonstrated as an effective low-level building block in such situations. Here we propose three high-level proxy-based programming patterns—distributed futures, streaming, and ownership—that make the power of the proxy pattern usable for more complex and dynamic distributed program structures. We motivate these patterns via careful review of application requirements and describe implementations of each pattern. We evaluate our implementations through a suite of benchmarks and by applying them in three meaningful scientific applications, in which we demonstrate substantial improvements in runtime, throughput, and memory usage.**

***Index Terms*—Distributed Computing, Futures, Streaming, Memory Management, Open-source Software**

I. Introduction

TASK-BASED programming paradigms, such as function-as-a-service (FaaS) and workflows, have emerged as vital methods for achieving computational flexibility and scalability. Applications are written as compositions of many distinct components, referred to as *tasks*, and FaaS platforms and workflow systems, collectively referred to as *execution engines*, abstract the complexities of executing tasks in parallel, whether across personal, cloud, edge, and/or high-performance computing (HPC) systems [1]–[6]. Such execution engines have enabled a wide variety of innovative applications.

Yet as the scale and ambition of task-parallel applications grows, they increasingly encounter difficulties due to the use of shared storage for the exchange of intermediate data among tasks—an approach commonly employed both by workflow systems (e.g., Parsl [5], Pegasus [7], Swift [8]) and cloud-hosted FaaS systems (e.g., AWS Lambda [4], Azure Functions [9], Google Cloud Function [3]). Such uses of shared storage can fail or become prohibitively expensive as the number of tasks, the geographic distribution of tasks, the quantities of data exchanged, and the required speed of data exchange grow.

J. Gregory Pauloski, Valerie Hayot-Sasson, Alexander Brace, Kyle Chard, and Ian Foster are with the Department of Computer Science, University of Chicago, Chicago, IL, USA (e-mail: jgpauloski@uchicago.edu).

J. Gregory Pauloski, Valerie Hayot-Sasson, Logan Ward, Alexander Brace, Kyle Chard, and Ian Foster are with the Data Science and Learning Division, Argonne National Laboratory, Lemont, IL, USA.

André Bauer is with the Department of Computer Science, Illinois Institute of Technology, Chicago, IL, USA.

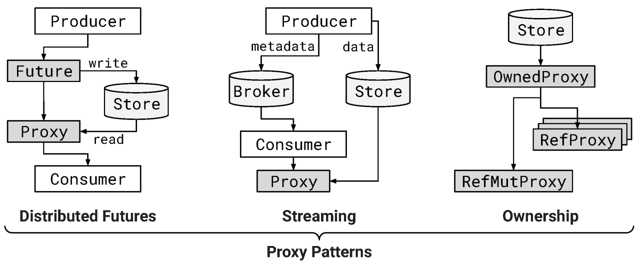


Fig. 1. Overview of the three proxy-based data flow patterns we design.

Many researchers have investigated alternative mechanisms for distributed and wide-area data management that circumvent these limitations of shared storage. For example, Linda's tuple space model provides unified access to a shared distributed memory space [10], DataSpaces provides a similar model for large-scale applications [11], [12], and peer-to-peer systems like the InterPlanetary File System provide decentralized content-addressed file sharing [13]. Another approach to simplifying data sharing is the *object proxy paradigm*, which provides transparent access and management for shared objects in distributed settings. This mechanism, long used with Java's Remote Method Invocation (RMI) [14], is also supported in Python via the ProxyStore system [15]. ProxyStore's transparent object proxies provide lightweight, wide-area references to objects in arbitrary data stores—references that can be communicated cheaply and resolved just-in-time via performant bulk transfer methods in a manner that is transparent to the consumer code. Recent work has shown how by decoupling data flow complexities from control flow-optimized execution engines [16]–[20], the object proxy paradigm can simplify implementations of dynamic application structures such as ML model training and inference.

Yet the object proxy paradigm remains a *low-level* abstraction that can be hard to use in practice due to the complexities inherent in managing many references to remote objects. Thus, we ask: Can we identify common *high-level patterns* that build on the proxy model to accelerate and simplify development of advanced applications? To this end, we review in this paper three computational science applications previously developed for workflow execution engines (*1000 Genomes*, *DeepDriveMD*, *MOF Generation*), identify limitations in the data flow patterns supported by those execution engines, and propose three new programming patterns that extend the proxy model to overcome these limitations (Fig 1):

- A *distributed futures* system for seamless injection of data flow dependencies into arbitrary compute tasks to overlap computation and communication;
- An *object streaming* interface that decouples event notifications from bulk data transfer such that data producers can unilaterally determine optimal transfer methods; and
- An *ownership* model that provides client-side mechanisms for managing object lifetimes and preventing data races in distributed task-based workflows.

Each pattern simplifies building sophisticated task-based applications that are to execute across distributed or remote compute resources (e.g., using FaaS or workflow systems). For each, we discuss its requirements and the protocols used to support it. Our reference implementations extend ProxyStore [15], our prior work, to leverage the existing low-level proxy model within Python, a popular and pervasive language for task-based distributed applications. The implementations are available within ProxyStore v0.6.5 and later, available on GitHub [21] and PyPI [22]. We evaluate our reference implementation for each pattern using (1) synthetic benchmarks across various FaaS and workflow systems and (2) our motivating applications, for which we reduce workflow makespan by 36% in 1000 Genomes, improve inference latency by 32% in DeepDriveMD, and optimize proxy lifetimes in MOF Generation.

The rest of this paper is as follows: Sec II introduces our motivating scientific applications; Sec III gives background on our prior work, ProxyStore; Sec IV outlines the design and implementation of each pattern; Sec V demonstrates synthetic evaluations; Sec VI presents our experiences applying these patterns to our motivating applications; Sec VII provides context about related work; and Sec VIII summarizes our contributions and future directions.

II. Motivating Applications

1000 Genomes: This bioinformatics pipeline [23] identifies mutational overlaps within the 2504 human genomes sequenced by the 1000 Genomes Project [24]. It comprises five stages: (1) fetch files, each containing all Single Nucleotide Polymorphisms (SNPs) in a chromosome, chunk, and process them in parallel to extract SNP variants by individual; (2) merge individuals' results of the prior stage; (3) score and select SNP variants based on their phenotypic effect; (4) compute overlap of selected SNP variants among pairs of individuals and by chromosome; and (5) compute frequency of overlapping variants. Executing scientific workflows in a FaaS setting may be preferred when access to specialized hardware, such as AI or quantum accelerators, or the ability to rapidly scale up or down is required, but workflow execution on a FaaS system poses challenges because FaaS systems rely on control flow to determine when to submit tasks. From the application perspective, however, the availability of data—the data flow—is the condition upon which tasks can be submitted. We use the 1000 Genomes workflow as an example of the challenges that arise when executing data flow oriented applications on control flow-optimized systems.

DeepDriveMD: Molecular dynamics (MD) simulation acts as a computational microscope [25] to enable the study of complex biomolecular systems. However, many important phenomena are difficult to sample using conventional MD, even with powerful supercomputers [26]. DeepDriveMD [27], [28] implements an emerging HPC paradigm in which machine learning (ML) methods are used to track a simulated state space and guide simulations toward a sampling objective. The DeepDriveMD client submits discrete training, inference, and simulation tasks and receives their results. This pattern causes two challenges. First, all data must flow through the client which limits performance at scale (e.g., data volume or task frequency), so a mechanism is needed to alleviate data flow burdens from the client when possible. Second, repeated tasks perform redundant work. For example, each inference tasks loads the latest ML model from disk, infers using the input batch, and compiles the results which will later become the input to a simulation task. This is inefficient because the same model is loaded multiple times across tasks, tasks may execute on different workers negating cache benefits, and every task incurs non-trivial overheads for scheduling and execution.

Metal-Organic Framework (MOF) Generation: This workflow [29] uses molecular diffusion models [30] to generate organic ligands, assemble MOF candidates, and employ physics models to identify candidates best suited for storing CO_2. The workflow uses a central process, referred to as a thinker [16], to determine which tasks to execute, and with what parameters. A core computational challenge is ensuring that the thinker has timely data, such as the latest diffusion model results, when deciding the next task. Object proxies have been used to improve thinker response time in similar applications [16], [31], but knowing the lifetime of proxied data is challenging in sophisticated workflows where the types of tasks to be executed are not know ahead of time.

III. ProxyStore

In software design, a *proxy* is an object that functions as an interface to another object [32]. A simple proxy will forward operations on itself to the real or *target* object, but often a proxy is used to provide extra functionality such as caching or access control, in addition to forwarding operations [15]. For example, distributed applications can use a proxy to invoke methods on a remote object, and data-intensive applications can use a virtual or *lazy* proxy which will perform just-in-time resolution of large objects (i.e., load the object from a remote location into local memory when first needed).

Lazy transparent object proxies can be used to communicate objects efficiently in distributed applications [15]. Here, a proxy refers to a target object stored in an arbitrary mediated communication medium (e.g., an object store, database, file system). The proxy forwards all operations on itself to the target, but importantly is totally transparent in that the proxy is an instance of the same type as the target. In Python, this means that `isinstance(p, type(t))` is true for a proxy *p* and its target *t*. The proxy is lazy in that it performs just-in-time resolution of the target. The target is not copied from the mediated storage into local memory until an operation is

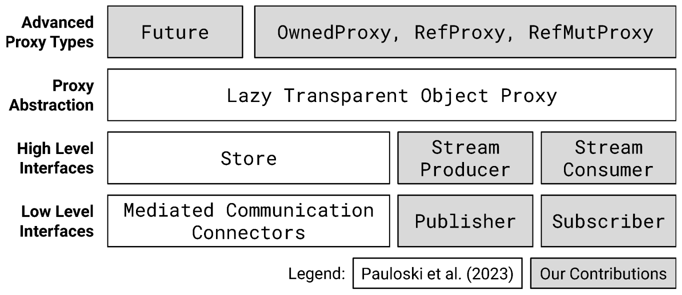


Fig. 2. Overview of the ProxyStore interface and abstraction stack with our contributions included in the shaded boxes.

invoked on the proxy. This proxy paradigm has both pass-by-reference and pass-by-value semantics; unused copies of the target object are not made when the proxy is passed between processes but the actual consumer of the proxy is given a copy.

The benefits of moving data via proxies are numerous: pass-by-reference reduces transfer overheads, no external information is required to resolve a proxy, shims or wrapper functions are eliminated, just-in-time resolution amortizes communication costs and avoids costs associated with unused objects, and proxies enable automatic access control. As such, this paradigm has been used to build a diverse suite of robust and scalable scientific applications [15]–[19], [31], [33], [34].

ProxyStore [15] implements this proxy paradigm which we use as the basis for our patterns' reference implementations. ProxyStore defines the *factory*, *connector*, and *store* constructs. The *factory* is a callable object that returns the target object when invoked. ProxyStore creates a unique factory for each target object containing the metadata and logic necessary to retrieve the target from a remote location. This factory is used to initialize a proxy, and a proxy is *resolved* once it has invoked its factory to retrieve and cache the target locally.

The *connector* is a protocol that defines the low-level interface to a mediated communication channel. A mediated channel is one where the communication between a producer and consumer is indirect, such as via a storage system [35]. This indirection is important because the process that creates a proxy and the process that resolves a proxy may not be active at the same time, in which case they could not communicate via direct mechanisms. ProxyStore provides many connectors, including interfaces to external mediated channels such as shared file systems, object stores (Redis [36] and KeyDB [37]), and peer-to-peer transfer systems (Globus Transfer [38], [39] and ProxyStore Endpoints [15]) and bespoke mediated channels that can leverage high-performance networks through the UCX [40] and Margo [41] libraries.

The high-level *store* interface, initialized with a connector, is used to create proxies of objects. A proxy p can be created from a target t by calling `Store.proxy(t)`. This method (1) serializes t using the default ProxyStore or user-provided serializer [42], (2) puts the serialized t in the mediated channel via the connector, (3) creates a factory with the appropriate metadata about t and the store/connector used, (4) initializes a proxy with the factory, and (5) returns the proxy. This process incurs some overhead but is trivial for larger objects. Prior work [15], [16] found the performance benefits of proxies to outweigh proxy creation and resolution overhead for objects larger than ∼10 kB; the exact threshold depends on many factors (e.g., connector choice, execution engine).

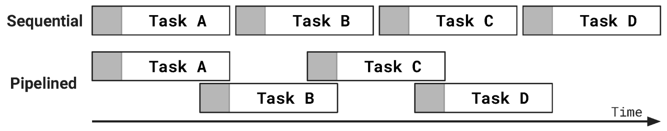


Fig. 3. Four tasks executed in a sequential (above) or pipelined (below) fashion. Each task produces data needed by the following task. The grey region at the start of each task represents startup overhead before the input data can be used. By enabling a successor task to start before its predecessor has finished, futures enable overlapping of startup overhead with computation, a form of pipelining.

```python
from dask.distributed import Client
from proxystore.connectors.foo import FooConnector
from proxystore.store import Store
from proxystore.store.future import Future

def producer(future: Future[str]) -> None:
    future.set_result('value')

def consumer(data: Proxy[str]) -> None:
    assert data is 'value'

with Store('example', FooConnector()) as store:
    client = Client(...)
    future: Future[str] = store.future()

    t1 = client.submit(producer, future)
    t2 = client.submit(consumer, future.proxy())

    t1.result(), t2.result()
```

Listing 1. Example usage of the ProxyFuture interface within tasks executed by Dask. A proxy created from a `Future` will block implicitly on the result of the future when needed. This interface abstracts the low-level communication away from the functions which set the result or consume the proxy.

IV. Proxy Patterns

We describe the design of each of the three advanced programming patterns that build on the aforementioned distributed object proxy base. We discuss the details of our reference implementations that extend ProxyStore, and Fig 2 describes how these patterns fit into the existing ProxyStore stack. These patterns are not mutually exclusive, but we discuss each in isolation for clarity.

A. Distributed Futures

A future represents a value that will eventually be available; the holder of a future can block on it until the value is resolved. Futures simplify writing non-blocking compute (e.g., remote procedure calls, database queries, or FaaS invocations) and I/O (e.g., network requests or file system reads) operations. Execution engines use futures to represent eventual task results, and this is valuable for representing long running remote execution or assembling applications with asynchronous callbacks. However, the distributed futures provided by execution engines have three key limitations: (1) these futures perform control and data synchronization so data flow cannot be optimized independent of control flow, such as to pipeline task execution as in Fig 3; (2) the transfer mechanisms used by the future cannot be optimized based on the type or location of data; and (3) futures produced by execution engines are only usable within the context of that execution engine (e.g., a future from one engine cannot be sent as input to another).

We design a distributed futures system called *ProxyFutures* that (1) supports explicit and implicit usage, arbitrary execution engines, arbitrary distributed memory backends, and seamless injection of data flow dependencies, and (2) addresses a limitation of ProxyStore that a proxy cannot be created before its target object exists. In ProxyFutures, a future f is created for an eventual value x, and f can be used to create any number of proxies p_f.

Consider an application with a main process M, a data producing process P, and a data consuming process C. M dispatches two tasks: T_P to P and T_C to C. T_P is to produce a value x to be consumed by T_C; thus, T_C has a data dependency on T_P. M can create a future f and associated proxy p_f, and pass f and p_f to T_P and T_C, respectively. When T_C first resolves p_f, it blocks until T_P has set the result of f. Importantly, T_C can be started before T_P has finished or even started. M, when creating f, can choose the communication method to be used based on where P and C are located and what communication methods are available between them; thus, the detailed communication semantics are abstracted from T_P and T_C. The implicit nature of p_f also means that the code for T_C can be invoked either on a value directly or on a proxy of the value. This equivalence simplifies code and testing and means that M can inject data flow dependencies via a future into arbitrary third-party functions that expect to receive data directly.

We implement this behavior by extending ProxyStore's `Store` interface to expose a `future()` method that returns a `Future` object. The `Future` class exposes two main methods: `set_result(obj: T)`, which sets the result of the future to an object of type `T` and `proxy()`, which returns a `Proxy[T]`. When a proxy created via `Future.proxy()` is resolved, the proxy blocks until the target value has been set via a call to `Future.set_result()`, as shown in Listing 1. Use of ProxyFutures does not affect when a successor task starts; scheduling is still managed by the execution engine and/or user application. ProxyFutures are best integrated at the application level so that developers can optimize task execution per their application requirements and to express more complex data dependencies than typically supported by execution engines.

Internally, communication between a `Future` and any child proxy(s) is handled via the `Store` used to create the `Future`. Thus, a future and associated proxies can be serialized and sent to arbitrary processes on arbitrary machines. In contrast, many standard-library future implementations use non-serializable async, thread, and inter-process synchronization mechanisms (e.g., `std::future` in C++ [43], concurrent and async futures in Python [44]), while RPC-based futures are only resolvable within the RPC framework (Dask futures [1] or Ray `ObjectRefs` [45]). The self-contained properties of the proxy mean that all logic for communication and resolution are embedded within the future and proxy; the future creator chooses communication methods on behalf of the process(es) which might set or consume the result of the future.

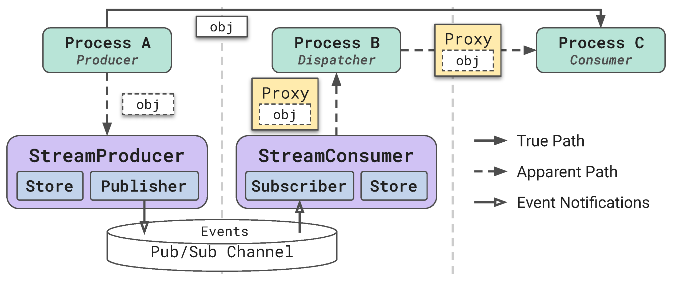


Fig. 4. The `StreamProducer` abstracts low-level communication details from the `StreamConsumer` and transparently decouples metadata from bulk data transfer. Yielding proxies, rather than objects directly, in the `StreamConsumer` enables just-in-time resolution and pass-by-reference optimizations.

```python
from globus_compute_sdk import Executor
from proxystore.stream import (StreamProducer,
    StreamConsumer, KafkaPublisher, KafkaSubscriber)

def producer() -> None:
    store = {'topic': Store(..., FooConnector(...))}
    publisher = KafkaPublisher(...)

    with StreamProducer(publisher, store) as producer:
        for item in ...:
            producer.send('topic', item)

def consumer() -> None:
    subscriber = KafkaSubscriber('topic', ...)

    with StreamConsumer(subscriber) as consumer:
        for item in consumer:
            assert isinstance(item, Proxy)

with Executor('<Endpoint UUID>') as client:
    t1 = client.submit(producer)
    t2 = client.submit(consumer)

    t1.result(), t2.result()
```

Listing 2. Example using the ProxyStream interfaces to stream data between two tasks executed remotely using Globus Compute. A Kafka broker is used for metadata and an arbitrary `FooConnector` for bulk data transfer.

B. *Object Streaming*

High-performance stream processing applications dispatch remote compute tasks on objects consumed from a stream, but task dispatch can quickly become a bottleneck with high throughput streams [27], [28], [33]. Consider the application in Fig 4, where process A is a data generator that streams chunks of data (i.e., arbitrary Python objects) to a dispatcher process B, which for each data chunk dispatches a compute task on a remote process C. Note that while the dispatcher consumes from the stream, it does not need the actual chunk of data; rather, it only needs to know that a chunk is ready (and potentially have access to user-provided metadata) in order to dispatch the task that will actually consume the chunk. We design a system called *ProxyStream* to enable scalable applications of this pattern. At its core, ProxyStream uses a stream of proxies, rather than data chunks. Bulk data are only transmitted between the data generator and the process/node computing on the proxy of the chunk, bypassing the intermediate dispatching process. ProxyStream optimizes for both metadata and bulk data transfer, has broad execution engine compatibility, provides a self-describing data format, and supports various communication modules to take advantage of high-performance networking stacks.

ProxyStream provides two high-level constructs, the

StreamProducer and StreamConsumer, that combine a message stream broker for low-latency event metadata propagation and a mediated communication channel for efficient bulk data transfer. A StreamProducer is initialized with a Publisher and a ProxyStore Store. The Publisher defines a protocol for sending event messages to a stream. We provide shims to many popular event streaming systems (Kafka [46], Redis Pub/Sub and Queues [36], ZeroMQ [47]) which implement the Publisher protocol. When a new object and optional metadata are sent to the StreamProducer, (1) the object is put in the store, (2) a new event containing the user provided metadata and information about where the object is stored is created, and (3) the event is published via the Publisher.

A StreamConsumer is initialized with a Subscriber, which, like the Publisher, defines a protocol for receiving event messages from a stream (Listing 2). The StreamConsumer is an iterable object, yielding proxies of objects in the stream until the stream is closed. Calling next() on the StreamConsumer waits for a new event metadata message via the Subscriber, creates a proxy of the object using the event metadata, and returns the proxy to the calling code. This process is efficient because the bulk object data has not been read at this point; rather, this will be delayed until the resolution of the proxy.

This model has many benefits: (1) communication mechanisms are abstracted from the stream consumer, (2) stream objects are resolved only when actually needed (wherever the proxy is resolved), (3) event message and bulk data transfer are decoupled, allowing the application to better optimize both forms of communication for the given application deployment environment and object characteristics, and (4) it provides a mechanism for implementing stateful actors in a workflow.

The ProxyStream interfaces support any combination of single/multi producer/consumer that is supported by the associated Publisher and Subscriber implementations. The StreamProducer supports mapping different stream topics to Store instances, enabling further optimization of communication mechanisms; batching; and plugins for filtering, sampling, and aggregation. The StreamConsumer support plugins for filtering and sampling. ProxyStream is fault-tolerant provided that the broker and communication channel are fault-tolerant.

ProxyStream can be integrated at the application or framework level. Listing 2 depicts use of ProxyStream within a Globus Compute application; we integrate ProxyStream within the DeepDriveMD framework for the evaluation in Sec VI.

C. Ownership

A limitation of the proxy model is the need to manage explicitly the lifetime of the associated target object. When a proxy is shared with more than one process, it is challenging to know when it is safe to free the target object. A ProxyStore proxy acts like a C/C++ pointer or raw pointer in Rust; thus, one process could prematurely free the target object, causing what is equivalent to a null pointer exception in the other process(es); delay freeing the object causing increased memory usage; or forget to free the object causing a memory leak. ProxyStore provides some guidance on using proxies safely, but ultimately it is up to the programmer to use proxies safely—a situation similar to C pointers.

To address this difficulty, we extend the proxy model with two features not provided by ProxyStore: automatic deletion of objects that have gone out of scope and safe support for mutating objects. Inspired by Rust's borrowing and ownership semantics, our design works in distributed contexts; provides different proxy types that can represent the owned, reference, and mutable reference types; enforces ownership and borrowing rules at runtime based on a proxy's type; and performs automatic dereferencing, coercion, and deletion.

Rust defines three *ownership rules*: (1) each value has an owner, (2) there can only be one owner at a time, and (3) a value is deleted when its owner goes out of scope [48]. A reference allows a value to be borrowed without relinquishing ownership. The *reference rules* are (1) at any given time, a value can have either one mutable reference or any number of immutable references and (2) references must always be valid. The Rust compiler enforces these rules, and the language provides data structures for runtime enforcement for more complex scenarios that the compiler cannot reason about.

Applying these rules in a distributed application, such as a computational workflow, can make memory management significantly easier without the need to perform global reference counting. Computations represented as directed acyclic graphs (DAGs) are particularly well suited to this model. As objects move from a parent DAG node to a child node, ownership can either be transferred to the child or the child can be given a borrowed reference. Thus, a node has full information about what operations are safe on objects that it receives. Ownership transfer means that the recipient node has full control over that object; an immutable reference means that the node can only read the object. A mutable reference means that the node has sole access to modify the object, but the node cannot create and share additional references: i.e., it is not allowed to pass a reference to its own child node.

One challenge of this model is knowing when a reference to an object goes out of scope, because this requires communication between the process that owns the object and the process that has a reference. However, in a task-based workflow, it is easy to reason that a reference passed to a task goes out of scope when the task completes (assuming that the task is well-behaved; an improperly behaved task would be one that, for example, creates and stores a memory-to-memory copy of the reference) and workflow systems already propagate information about task completion.

A second challenge is representing the ownership or borrowing of an object. The Rust compiler and dot operator abstracts much of the nuance of dealing either with objects directly or with their references [49]. In Python, for example, an object T could be wrapped in a Owned[T], Ref[T], and RefMut[T], in a similar manner to some Rust constructs. However, use of these constructs would be cumbersome, as all referencing, dereferencing, or coercion would have to be done manually.

The transparent object proxy is well-suited to solve these object scope and reference representation problems. An object that is proxied by a process becomes a shared object that is stored on some global object store accessible by all processes in the distributed environment (Listing 3). The target object is serialized, put in the global store, and an OwnedProxy is

```python
class Store(Generic[Connector]):
    def owned_proxy(obj, ...) -> OwnedProxy: ...

def into_owned(Proxy) -> OwnedProxy: ...
def borrow(OwnedProxy) -> RefProxy: ...
def mut_borrow(OwnedProxy) -> RefMutProxy: ...
def clone(OwnedProxy) -> OwnedProxy: ...
def update(OwnedProxy | RefMutProxy) -> None: ...
```

Listing 3. Proxy ownership model interfaces and functions. Functions are preferred over methods on the associated proxy reference types to prevent unintentionally clobbering a method of the same name on the target object.

```python
from proxystore.connectors import FooConnector
from proxystore.store import Store
from proxystore.store.lifetimes import LeaseLifetime

with Store('example', FooConnector()) as store:
    lease = LeaseLifetime(store, expiry=10)
    proxy = store.proxy('value', lifetime=lease)
    lease.extend(5)
    time.sleep(20)
    assert lease.done()
    # Object associated with the proxy has been removed
```

Listing 4. Example usage of lifetimes when creating a proxy. A `Lifetime` instance represents a physical or logical scope that will clean up all resources (i.e., objects) that were associated with the lifetime when closed.

returned. The `OwnedProxy` contains a reference to the global object and, if the proxy has been resolved, a local copy of the object upon which the proxy forwards operations to.

An `OwnedProxy` enforces the following rules [c.f. Rust's ownership rules]: (1) each object in the global store has an associated `OwnedProxy`, (2) there can only be one `OwnedProxy` for any object in the global store, and (3) when `OwnedProxy` goes out of scope, the object is removed from the global store.

When invoking a task on an `OwnedProxy` (i.e., calling a local or remote function), the caller can do one of four things:

- Yield ownership by passing the `OwnedProxy` to the task.
- Clone `OwnedProxy` and pass the cloned `OwnedProxy` to the task. Cloning an `OwnedProxy` will create a new copy of the object in the global store that will be owned by the callee task while the caller still owns the original object.
- Make a `RefProxy` and pass the `RefProxy` to the task. The caller still retains ownership, and the task can only read the object via the `RefProxy`. The callee task can only mutate its local copy, not the global copy. The caller's `OwnedProxy`, used to create the `RefProxy`, keeps track of the references that it has created. Any number of tasks can be invoked on a `RefProxy` at a time.
- Make a `RefMutProxy` and pass the `RefMutProxy` to the task. The caller still retains ownership (essentially the privilege to delete), but the callee task now has sole access to modify the object in the global store. The caller's `OwnedProxy` marks that it has created a `RefMutProxy` and thus cannot mutate itself until the callee task that has the `RefMutProxy` completes. Only one task can be invoked on a `RefMutProxy` at a time and a `RefMutProxy` and `RefProxy` cannot exist at the same time.

The lifetimes of a `RefProxy` and `RefMutProxy` are strongly coupled to those of the tasks they are passed to. Any violation of these rules, such as an `OwnedProxy` that goes out of scope or is deleted while a `RefProxy` or `RefMutProxy` exists, will raise a runtime error. It is also possible to extend a static code analysis tool to verify correctness prior to execution.

Execution engines typically use futures to encapsulate the asynchronous execution of a task. Thus, we use callbacks on the task result futures to indicate that the references associated with a task have gone out of scope. The primary limitation of this approach is that each execution engine has a different syntax for submitting a task and getting back a future. Rather than modify each engine, we provide a set of shims that appropriately parse task inputs and construct a callback on the task's future that will propagate the necessary information about references going out of scope. The `StoreExecutor`, an interface provided by ProxyStore, wraps an execution engine client (e.g., a Globus Compute, Dask, or Parsl client) and automatically proxies task parameters and results based on user-defined policies and manages references associated with tasks [42]. The `StoreExecutor` is easy to use, but applications requiring more fine-grain control can use the API in Listing 3.

The ownership model is not fault-tolerant when the client crashes in a manner which prevents garbage collection, but the model is compatible with fault-tolerant execution engines such as those that automatically rerun tasks on failure. Since only a single `RefMutProxy` can exist, the ownership model is not optimal for applications with many concurrent writers to the same object; a database, for example, may be more suitable.

So far, we have constricted ourselves to tasks (i.e., function invocations) as the only region of code over which we can define a lifetime; thus, all references to an object are equal to the lifetime of the single task invoked on that reference. Yet a workflow application may employ more complex lifetimes. For example, a lifetime could be assigned to a set of tasks that are a subgraph of the global DAG, and a programmer might want to define references to global objects that are associated with this custom lifetime. Using proxy references is a valid solution but would require additional code to manage and map references to the scopes contextual to the application.

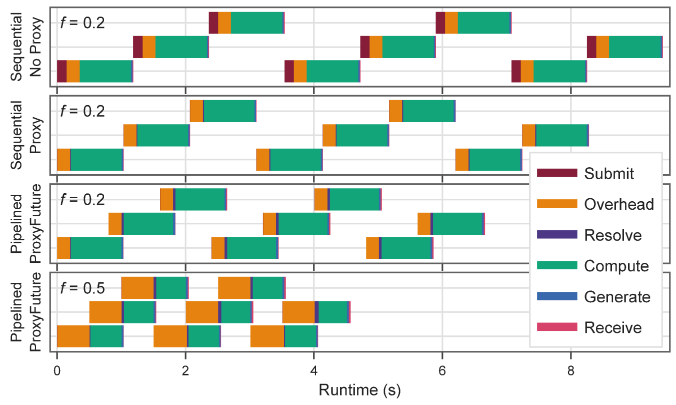
We provide the `Lifetime` construct, an alternative to proxy references, for managing object lifetimes in more complex scenarios. A lifetime, attached to one or more proxies upon proxy creation, will clean up associated objects once the lifetime has ended. We provide three `Lifetime` types and the API can be extended to implement new types. The context-manager lifetime enables mapping proxy lifetimes to discrete segments of code, the time-leased lifetime will clean up associated objects once the lease has expired and not been extended, and the static lifetime persists objects for the remainder of the program. Listing 4 provides a time-leased lifetime example.

V. Evaluation

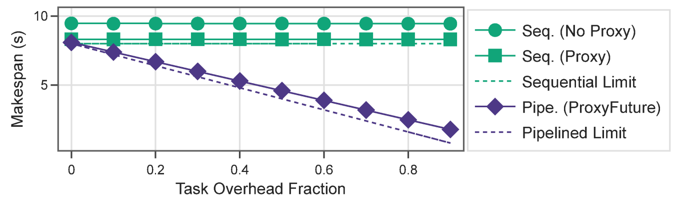
We conducted experiments on Polaris at the Argonne Leadership Computing Facility. Polaris has 560 nodes interconnected by an HPE Slingshot 11 network and a 100 PB Lustre file system. Each node contains one AMD EPYC Milan processor with 32 physical cores, 512 GB of DDR4 memory, and four 40 GB NVIDIA A100 GPUs.

A. Task Pipelining with ProxyFutures

We first evaluate the effectiveness of ProxyFutures for reducing workflow makespan via pipelining. We define a



(a) Task schedules for no-proxy; proxy; ProxyFutures $f = 0.2$, $f = 0.5$.



(b) Makespan vs. overhead fraction for no-proxy, proxy, ProxyFutures.

Fig. 5. Results for synthetic benchmark with 8 tasks, each sleeping for 1 s and communicating 10 MB to its successor, and with overhead fraction f determining how much of the 1 s can be overlapped with its predecessor task. (Top) Task execution schedules in four scenarios: *sequential no proxy*, with delays due to workflow engine submission costs; *sequential proxy*, with proxies enabling immediate task start after proxy is resolved; and two *pipelined ProxyFuture* cases ($f = 0.2$ and $f = 0.5$), in which distributed futures relax strict inter-task dependencies and enable pipelining to overlap initial task overheads. The *overhead* and *compute* sleeps dominate in all cases, while times to *resolve* task input data and *receive* task results increase, with overhead fraction, while makespan decreases due to pipelining overlap. (Bottom) Synthetic benchmark makespan vs. overhead fraction, for no proxy, proxy, and ProxyFuture scenarios. Each value is averaged over five runs; standard deviations are all less than 20 ms.

synthetic benchmark that submits n tasks in sequence, each sleeping for s seconds and then producing d bytes to be consumed by the next task. As in Fig 3, a fraction f of each task is treated as startup overhead (e.g., library loading, model initialization, state synchronization). Thus, each task sleeps for $f \times s$ seconds, resolves its input data, and then sleeps for the remaining $(1 - f) \times s$ seconds to simulate computation. We compare three deployments: sequential without proxies (*No Proxy*), sequential with proxies (*Proxy*), and pipelined with ProxyFutures (*ProxyFuture*). In the first two, task t_i is submitted once the result of task t_{i-1} is available, with in *No Proxy*, the workflow engine handling data transfer, and in *Proxy*, data transfer being offloaded from the workflow engine. In *ProxyFuture*, tasks t_{i-1} and t_i share a proxy and future pair and t_i is submitted before t_{i-1} is complete.

Setup: We run a Dask cluster on a single Polaris compute node. In the *Proxy* and *ProxyFuture* deployments, a Redis server running on the compute node is used as the mediated communication channel for the proxies. We run n = 8 tasks with intermediate data of d = 10 MB and task time of s = 1 s; the short task time is to focus on the time spent producing and waiting on data. We vary overhead fraction f from 0 to 0.9.

Results: We plot in Fig 5a the start and end times of each stage in each task's lifecycle for each deployment, for $f = 0.2$, and for *ProxyFuture*, also for $f = 0.5$. Each task incurs fixed *overhead* and *compute* costs, of f and $(1 - f)$ s, respectively. Other costs include: *submit*, the time to submit and begin execution; *generate*, the time to produce output data; and *receive*, the time to receive the result by the client. *Proxy* and *ProxyFuture* also incur *resolve* costs associated with the use of proxies. Fig 5b shows the implications of these differences by presenting average makespan as a function of task overhead fraction for the three deployments. The use of proxies in *Proxy* improves task submission time relative to *No Proxy*, reducing makespan by 12%. The pipeline overlapping in *ProxyFuture* enables close to the theoretical limit (dashed line) as determined by inter-task data dependencies. For example, the ideal makespan reduction of a pipeline execution is 20% when $f = 0.2$; we observe 19.6% in *ProxyFuture*. The increased divergence from the ideal reduction at larger overhead fractions occurs because task submission and data transfer costs become more significant as overlapping increases. Thus, a subsequent task begins waiting on its future slightly before the prior task has set the result of the future.

Outcomes: DAG-based workflow execution models limit optimization of task execution because a child task cannot start until its parents have finished, even if the programmer knows it may be beneficial to start it sooner. For example, module loading can account for a significant portion of overall task runtime. Loading TensorFlow on NERSC's Perlmutter takes 5 s in the best case but nearly a minute when many workers read files concurrently [34]. This is particularly noticeable with smaller models where inference time can be measured in fractions of a second. On Polaris, the machine used here, we found that five common libraries (NumPy, Scikit-learn, SciPy, PyTorch, TensorFlow) [50] require from 100 ms to 2 s to import even under ideal conditions with a single worker. Tasks must also often perform other work, such as file loading, initializing model weights, or state synchronization, before needing their input data. The ProxyFutures model provides for seamless encoding of data dependencies and optimistic task pipelining when tasks have nontrivial initial overheads. While we used Dask and Redis in this experiment, our approach will work with *any* task-based execution engine and mediated communication channel. This engine-agnostic approach will enable programmers to coordinate tasks across multiple execution engines concurrently.

B. *Scalable Stream Processing*

Here we evaluate scalable stream processing with ProxyStream. As in Fig 4, there is one data producer publishing data of size d to the stream with a rate r (items per second). A dispatch node consumes data from the stream and dispatches a compute task for each data item on to a cluster of n workers. Each compute task is simulated by a task which sleeps for s seconds. The dispatcher executes on a login node, and given n workers, one worker is allocated as the producer while the remaining $n - 1$ workers are used to execute compute tasks.

Setup: We compare three streaming configurations. In *Redis Pub/Sub*, data are published directly to a Redis pub/sub topic

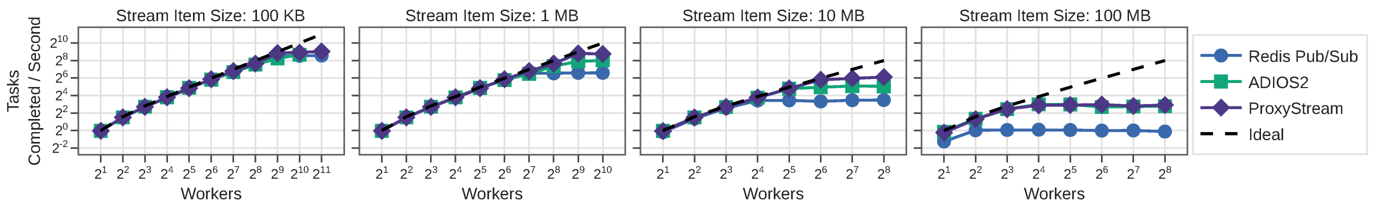


Fig. 6. Compute tasks completed per second as a function of stream item data size and number of workers. One worker generates data consumed by a central dispatcher that launches simulated compute tasks (one second sleep tasks) for each item across the remaining $n-1$ workers. At small data sizes ($\leq$ 100 KB), data transfer overheads are negligible and the dispatcher can keep up with incoming stream data; however, at large data sizes and worker counts, the dispatcher becomes overwhelmed by the size of data transfers required for each task in the Redis Pub/Sub configuration. ProxyStream transparently decouples data flow from control flow improving overall system performance as stream data sizes and the number of workers is increased.

that is consumed by the dispatcher before being sent to a worker to be computed on. In *ADIOS2*, data are written step-by-step to an ADIOS2 stream [51]. The dispatcher iterates on steps and launches worker tasks which will read the data from the ADIOS2 stream at a specified step. In *ProxyStream*, data are published to a `StreamProducer` which decouples metadata from bulk data, sending metadata to a Redis Pub/Sub topic and storing bulk data in a Redis Key/Value store. The dispatcher consumes proxies of stream data via the `StreamConsumer` and sends proxies to workers to be computed on. ADIOS2 and ProxyStream avoid data transfers through the dispatcher.

We use Parsl's `HighThroughputExecutor`, which can scale to thousands of tasks per second, to manage task execution. We set the producer's data publishing rate $r = (n-1)/s$ items per second, where $s = 1$ s for all tasks. Assuming no overheads in the system, this rate would keep each of the $n-1$ compute workers constantly fed with new data. A range of data sizes d and workers n are evaluated to understand stream scaling throughput limitations. We assign one worker per core so there are at most 32 workers per node. We run each configuration for between five and thirty minutes, depending on the scale, which is long enough for the processing throughput (i.e., tasks completed per second) to stabilize.

Results: Fig 6 shows the average compute tasks completed per second. At the smallest data size, $d = 100$ kB, performance is comparable between the three methods because data are not large enough to stress the system. For larger worker counts n and data sizes d, the default Redis Pub/Sub deployment slows because the dispatcher becomes a bottleneck, processing stream data at $\sim$100 MB/s. This rate is slower than the network connection between the Redis server and dispatcher because the dispatcher must, for each stream item, receive and deserialize the item from Redis; compose the task payload, serializing the item again; and communicate the task payload to a worker. Thus the dispatcher cannot process the incoming stream data fast enough to keep workers fed with new tasks when the number of workers or data size is sufficiently high.

ADIOS2 performs better than Redis Pub/Sub because we configured workers to read items from the stream directly based on a step index provided by the dispatcher, improving the latency between the dispatcher receiving stream data and launching a new task. However, ADIOS2 requires changes to the worker task code not needed by the other two methods.

ProxyStream also alleviates data transfer and serialization burdens from the dispatcher enabling performance on par

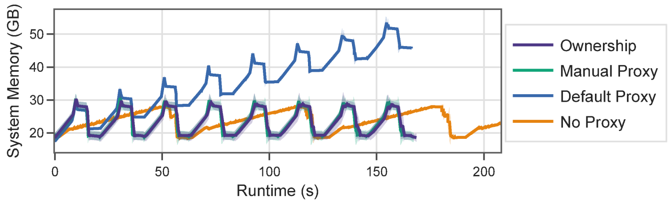


Fig. 7. Average system memory usage over three runs of a simulated MapReduce workflow. Shaded regions denote standard deviation in memory usage. Memory management limitations in ProxyStore cause baseline memory utilization to increase over time. Manual management can alleviate this problem, but requires careful implementation and prior knowledge. In contrast, our ownership model provides automated memory management equal to a hand-tuned implementation and enforces a set of rules at runtime.

with or better than ADIOS2 but does so transparently without needing changes to the worker task code. The peak processing throughput of ProxyStream is 1.7$\times$ and 2.0$\times$ faster than ADIOS2 for 1 MB and 10 MB item sizes, respectively. Compared to the Redis Pub/Sub baseline, ProxyStream is 4.6$\times$ and 6.2$\times$ faster for 1 MB and 10 MB item sizes, respectively. At $d = 100$ MB, the largest data size evaluated, and $n = 256$, ProxyStream is 7.3$\times$ faster than Redis Pub/Sub. ProxyStream and ADIOS2 perform similarly at this scale because other aspects of the experimental configuration become bottlenecks. Namely, task execution overheads and storing the data produced by the generator limit peak throughput. A faster data storage system or multiple data generators would be needed to achieve scaling beyond this point, and ProxyStream does support modular data storage and multi-producer configurations.

Outcomes: Streaming proxies, rather than data directly, ensures that objects in the stream are only resolved once needed, thus avoiding overheads due to objects passing via intermediate processes. The `StreamProducer` and `StreamConsumer` interfaces provide a mechanism for composing arbitrary message brokers and mediated communication methods, permitting developers to optimize application deployments without altering task code. The resulting distributed applications are more portable and generalizable to new hardware systems.

C. Memory Management

We evaluate the automatic memory management of the proxy ownership model by comparing system memory usage over a simulated workflow to ProxyStore's default memory management and a manual memory management approach which relies on the a priori knowledge of the programmer to free shared objects. We also compare to a baseline without any proxies where data are sent directly along with task requests.

Setup: We execute a simulated workflow that imitates a series of map-reduces across a local Dask cluster on a single compute node of Polaris. We run the workflow using each of the proxy memory management models, default, manual, and ownership, and a baseline without proxies using Dask for all data management. We record average memory usage across three workflow executions for each configuration. Eight consecutive map-reduces are performed where each of 32 mappers receives 100 MB and produces 10 MB. We choose 100 MB because the value is large enough to be observable in the memory trace (i.e., larger than the baseline memory usage fluctuations) but is also below the Redis default maximum value size of 512 MB. A single reducer consumes data produced by all mappers. In addition to consuming and producing data, each tasks sleeps for 5 s.

Results: Fig 7 presents the system memory usage traces for each memory management model. The limited default memory management of ProxyStore results in memory usage slowly increasing throughout execution as shared objects are created but never freed. The automated management of our ownership model performs identically to manual management and appropriately evicts objects as references go out of scope.

The "no proxy" baseline passes data directly to Dask and utilizes Dask's built-in distributed memory management. We observe that Dask appropriately frees all task data; however, the overall runtime is three times slower. The severe slow down is because Dask's graph serialization performs poorly with large (>1 MB in our experience) arbitrary Python objects. Dask is optimized for transferring arrays and dataframes, and we found Dask's performance to be similar to the proxy cases when data were formatted as NumPy arrays.

Outcomes: Our ownership model presents a marked improvement in using proxies in distributed workflows. Enforcing ownership rules at runtime makes it easy to reason about what operations on shared objects are safe and prevents programming mistakes which may lead to memory leaks. Our reference implementation is designed to be agnostic to the underlying task execution engine, but we believe that incorporating this model directly into execution engines can enable more powerful features.

VI. Applications

1000 Genomes: We use the 1000 Genomes workflow to investigate ProxyFutures as a mechanism for reducing task overheads and extending data flow dependencies to FaaS systems. Tasks in the original 1000 Genomes workflow were implemented as Bash scripts. We use the Python implementation of 1000 Genomes, where tasks are implemented as functions, to execute the workflow using a FaaS execution engine such as Globus Compute (which we use in the experiments reported here, due to its integration with HPC systems).

We evaluate the makespan of the resulting workflow, using 5% of the 1000 Genomes dataset, on a single `compute-zen-3` node, with two 64-core CPUs and 256 GB memory, on

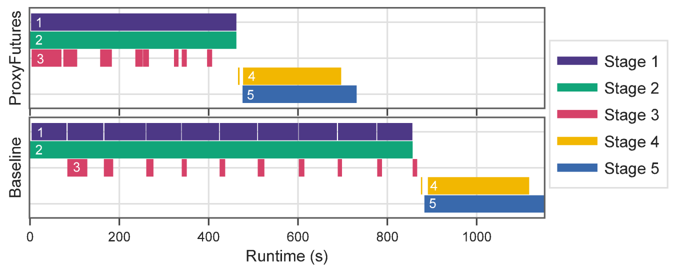


Fig. 8. 1000 Genomes Workflow stage start and ends times. ProxyFutures reduces workflow makespan by starting computations when data are available rather than when prior tasks complete.

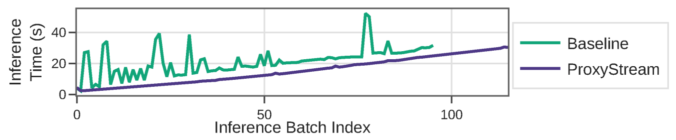


Fig. 9. Comparison of inference round-trip time between two DeepDriveMD implementations: baseline and ProxyStream. The size of each batch increases over time as the application accumulates more data points.

Chameleon Cloud's CHI@TACC cluster [52]. Fig 8 shows workflow stage start and end times for a baseline implementation, which uses Globus Compute's native futures for data synchronization between tasks, and a ProxyFutures implementation. As each stage can contain up to thousands of tasks, we consolidate the tasks within stages for clarity. ProxyFutures reduce workflow makespan by 36%, by better overlapping task execution and communication costs across stages. More specifically: (1) tasks within stages 1, 2, and 3 are better overlapped, reducing the stage makespans by 47–48%; (2) response time, the time between receiving a task result and submitting another task, is improved (for example, by 54% when starting stage 4); and (3) stages 4 and 5 are 5% faster due to reduced data transfer overheads. We also note there are no dependencies between tasks within stages 4 or 5, so these stages do not benefit to the same degree as the earlier stages.

DeepDriveMD: We modify the Parsl implementation of DeepDriveMD [53] to stream inference batches and results to and from a single, persistent inference task with ProxyStream. A persistent inference task eliminates task overheads and enables reuse of models and caches. Streaming with proxies reduces overheads in the DeepDriveMD client because received inference results are immediately added to a queue of simulation task inputs. In addition to ProxyStream, ProxyFutures are used to indicate availability of a new ML model to the inference task and proxy references for management of intermediate task data.

We compare the performance of DeepDriveMD to a version that uses proxy patterns. We run each version for three hours using 40 GPUs on Polaris, dedicating one GPU for inference, one for training, and the remainder for simulations. Round-trip inference time, shown in Fig 9, is reduced from an average of 21.9±8.8 s to 15.0±8.4 s, a 32% improvement, and 21% more inference batches were processed in the same wall time. Reducing inference time is key to enabling greater simulation throughput, such as when the number of simulation workers

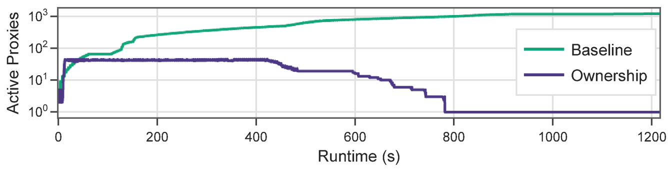


Fig. 10. Number of active proxies (i.e., proxies that still have a stored target object) during the runtime of the MOF Generation application. Our ownership model for proxies appropriately cleans up proxies when no longer needed while maintaining the benefits of the pass-by-reference model.

is increased or simulation time is reduced.

MOF Generation: We modify the MOF Generation application to communicate all task input and output data larger than 1 kB via proxies. (The overhead of proxying simple data types such as boolean flags or configuration strings is greater than sending those objects directly.) We deploy the application with default settings on ten Polaris nodes. We run the application twice: with the standard proxy implementation of ProxyStore and with our proxy ownership model. Here, ownership was sufficient; we did not use the lifetimes model. We record the number of actively proxied objects during the application's runtime. As shown in Fig 10, the ownership model appropriately evicts proxied data when the lifetime of the associated proxy ends without altering the runtime behaviour of the application. Manual memory management is possible, as discussed in Sec V-C, but automated management is safer and makes adoption of advanced programming practices, such as those we present here, easier and more appealing.

VII. Related Work

Futures are a pervasive programming abstraction for asynchronous and concurrent programming [54], [55]. Implicit futures act as references; any dereference blocks automatically until the value is resolved [56]; thus, they typically require language-level support [57]–[61]. Explicit futures provide a public interface, such as a `get` method, that must be invoked to block and retrieve the value; consequently, they can be provided by languages and third-party libraries.

Explicit futures require control flow synchronization code, which reduces code flexibility and complicates functions that want to operate on a future or a value directly. Either two implementations or multiple execution paths must be present to support each case. Implicit futures are also inflexible because they require that the language's type system handle the mechanics of lifting the value out of the future transparently. Thus few languages support implicit futures, and programmers have limited ability to modify the resolution and lifting processes. ProxyFutures address these key limitations by providing both an explicit mechanism, the `Future`, and an implicit mechanism, the `Proxy`, for Python applications.

Distributed futures represent values that, when available, may be located in remote process memory. Distributed futures are often underpinned by a remote procedure call (RPC) system, such as in Dask [1], PyTorch [62], and Ray [45], [63]. Because these futures are implemented by the RPC framework, rather than the language, all are necessarily explicit futures, and their use is limited to the confines of the framework. Thus, for example, one cannot create a distributed future in Dask or Ray and then invoke a serverless function with Globus Compute [6] on that future. In contrast, ProxyFutures works across frameworks and supports many mediated communication methods via a robust and extensible plugin system.

Streaming applications in which producers and consumers generate and process data continuously are commonly executed at scale on high-performance and cloud computing systems. Their persistence and resilience needs may be met by message queuing systems such as Apache Kafka [46], Redis [36], and RabbitMQ [64]. However, these systems typically optimize for high-throughput, low-latency transmission of small, structured events, in order that these events can be aggregated, filtered, or transformed, as in Kafka.

In contrast, high-performance science applications often produce large raw or unstructured data accompanied by structured metadata [65]. File-oriented distributed applications often use GridFTP [39], [66]. Dispel4py [67], [68] maps abstract definitions of streaming workflows onto concrete distributed execution frameworks, such as Python multiprocessing or MPI [69]. Streamflow [70] extends the DAG-based workflow model to integrate continuous event processing. ADIOS WASP [71], a data staging platform for scientific stream processing, uses a self-describing file format and supports advanced networking technologies such as RDMA. The SciStream middleware [72] enables fast, secure memory-to-memory streaming between nodes that lack direct network connectivity. CAPIO [73] provides a middleware layer for injecting I/O streaming capabilities into file-based workflows.

Consuming an entire stream item (data and metadata) is expensive when only metadata are needed for decision making or data is to be forwarded to another application component. ProxyStream decouples event metadata notification from bulk data transfer. Streaming proxies allows data transfers to occur when and where needed, with specifics of the message broker and data storage abstracted from the program.

Garbage collection in distributed environments is challenging. Automatic techniques such as reference counting and tracing garbage collectors exist, but often requires a priori knowledge by the application programmer to add custom logic for shared object management, and can be inefficient in distributed environments [74]–[77]. Maintaining global reference counts or traces adds network overheads, single sources of failure (if reference counting is centralized), or atomicity/consistency challenges (if reference counting is distributed).

Leases, a decentralized, time-based mechanism, can be used to avoid maintaining a shared state across processes [78]. Task-based execution engines can avoid shared state problems and the complexities of reference count message passing because the central client or scheduler can act as a single source of truth [1], [63]. The notion of ownership uses a program's inherent structure to decentralize state management. In PyTorch RPC, each object has a single owner that maintains the global reference count as remote processes need to access the data [62]. Related work extends this concept to implement distributed futures and task recovery in Ray [45].

Our proxy-based approach avoids the complexities of global reference counting by associating object lifetimes with tasks, and our framework-agnostic approach means that object scopes can be appropriately managed across complex, distributed applications.

VIII. Conclusion

The lazy object proxy is a powerful construct for building distributed applications, providing benefits of both pass-by-reference and pass-by-value while abstracting low-level communication details from consumers. Here, we have applied this construct to realize three powerful parallel programming patterns: a compute framework agnostic distributed futures system, a composable streaming interface for data-intensive workloads, and an ownership model for object management in distributed, task-based applications. We evaluated these patterns through synthetic benchmarks and showcased three classes of scientific applications that can benefit from the proxy paradigm powered patterns. Specifically, we reduced the 1000 Genomes workflow makespan by 36%, reduced DeepDriveMD inference latency by 32%, and optimized memory usage during MOF generation.

These patterns enable the development of robust, scalable, and portable applications. For example, ProxyFutures empowers data flow dependencies between tasks executed across different execution engines, such as when one engine is used for local execution on a cluster and another for remote execution on cloud resources. ProxyStream can support long-running scientific campaigns by using cloud-hosted message brokers for reliable metadata streaming and Globus Transfer for federated, persistent bulk storage and efficient transfer. The proxy ownership model provides automated wide-area memory management for distributed and cross-site workflows. In the future, we will investigate further programming patterns that can be enhanced with the proxy paradigm. Our work here serves as a reference for integrating these design patterns into execution frameworks, such as Dask, Globus Compute, or Parsl, and other high-performance computing toolkits. By providing first-class support for these patterns directly within commonly used frameworks, we expect to enable speedups in many scientific applications.

Acknowledgments

This research was supported in part by the National Science Foundation under Grant 2004894 and the ExaWorks Project and ExaLearn Co-design Center of the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration. We used resources provided by the Argonne Leadership Computing Facility (ALCF), a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357, and the Chameleon testbed supported by the National Science Foundation.

References

[1] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *14th Python in Science Conference*, vol. 130, 2015, p. 136.

[2] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/. Accessed Jan 2023.

[3] "Google Cloud Functions," https://cloud.google.com/functions/. Accessed Jan 2023.

[4] "AWS Lambda," https://aws.amazon.com/lambda. Accessed Jan 2023.

[5] Y. Babuji, A. Woodard, Z. Li, D. S. Katz, B. Clifford, R. Kumar, L. Lacinski, R. Chard, J. M. Wozniak, I. Foster, M. Wilde, and K. Chard, "Parsl: Pervasive parallel programming in Python," in *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2019. [Online]. Available: https://doi.org/10.1145/3307681.3325400

[6] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "funcX: A Federated Function Serving Fabric for Science," in *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020. [Online]. Available: http://dx.doi.org/10.1145/3369583.3392683

[7] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus, a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X14002015

[8] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633–652, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819111000524

[9] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. ul Haq, M. I. ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows Azure Storage: A highly available cloud storage service with strong consistency," in *23rd ACM Symposium on Operating Systems Principles*. ACM, 2011, p. 143–157. [Online]. Available: https://doi.org/10.1145/2043556.2043571

[10] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and friends," *Computer*, vol. 19, no. 08, pp. 26–34, 1986.

[11] C. Docan, M. Parashar, and S. Klasky, "DataSpaces: An interaction and coordination framework for coupled simulation workflows," in *19th ACM International Symposium on High Performance Distributed Computing*. ACM, 2010, p. 25–36. [Online]. Available: https://doi.org/10.1145/1851476.1851481

[12] M. F. Aktas, J. Diaz-Montes, I. Rodero, and M. Parashar, "WA-Dataspaces: Exploring the data staging abstractions for wide-area distributed scientific workflows," in *46th International Conference on Parallel Processing*, 2017, pp. 251–260.

[13] J. Benet, "IPFS - Content addressed, versioned, P2P file system," 2014, https://arxiv.org/abs/1407.3561.

[14] G. Biegel, V. Cahill, and M. Haahr, "A dynamic proxy based architecture to support distributed Java objects in a mobile environment," in *OTM Conferences / Workshops*, vol. 2519, 11 2002, pp. 809–826.

[15] J. G. Pauloski, V. Hayot-Sasson, L. Ward, N. Hudson, C. Sabino, M. Baughman, K. Chard, and I. Foster, "Accelerating Communications in Federated Applications with Transparent Object Proxies," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23, New York, NY, USA, 2023. [Online]. Available: https://doi.org/10.1145/3581784.3607047

[16] L. Ward, G. Sivaraman, J. G. Pauloski, Y. Babuji, R. Chard, N. Dandu, P. C. Redfern, R. S. Assary, K. Chard, L. A. Curtiss, R. Thakur, and I. Foster, "Colmena: Scalable machine-learning-based steering of ensemble simulations for high performance computing," in *IEEE/ACM Workshop on Machine Learning in High Performance Computing Environments*. IEEE, 2021. [Online]. Available: http://dx.doi.org/10.1109/mlhpc54614.2021.00007

[17] H. Harb, S. N. Elliott, L. Ward, I. T. Foster, S. J. Klippenstein, L. A. Curtiss, and R. S. Assary, "Uncovering novel liquid organic hydrogen carriers: a systematic exploration of chemical compound space using cheminformatics and quantum chemical methods," *Digital Discovery*, vol. 2, pp. 1813–1830, 2023. [Online]. Available: http://dx.doi.org/10.1039/D3DD00123G

[18] N. Collier, J. M. Wozniak, A. Stevens, Y. Babuji, M. Binois, A. Fadikar, A. Würth, K. Chard, and J. Ozik, "Developing distributed high-performance computing capabilities of an open science platform for robust epidemic analysis," 2023.

[19] G. Dharuman, L. Ward, H. Ma, P. V. Setty, O. Gokdemir, S. Foreman, M. Emani, K. Hippe, A. Brace, K. Keipert, T. Gibbs, I. Foster, A. Anandkumar, V. Vishwanath, and A. Ramanathan, "Protein generation via genome-scale language models with bio-physical scoring," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 95–101. [Online]. Available: https://doi.org/10.1145/3624062.3626087

[20] J. G. Pauloski, V. Hayot-Sasson, M. Gonthier, N. Hudson, H. Pan, S. Zhou, I. Foster, and K. Chard, "TaPS: A Performance Evaluation Suite for Task-based Execution Frameworks," in *IEEE 20th International Conference on e-Science*. New York, NY, USA: IEEE, 2024, pp. 1–10.

[21] "ProxyStore GitHub," https://github.com/proxystore. Accessed July 2024.

[22] "ProxyStore PyPI," https://pypi.org/project/proxystore/. Accessed July 2024.

[23] "1000 Genomes Workflow," https://github.com/pegasus-isi/1000genome-workflow/. Accessed Mar. 2024.

[24] "1000 Genomes Project," https://www.internationalgenome.org/1000-genomes-summary/. Accessed Mar. 2024.

[25] R. O. Dror, R. M. Dirks, J. Grossman, H. Xu, and D. E. Shaw, "Biomolecular simulation: A computational microscope for molecular biology," *Annual review of biophysics*, vol. 41, pp. 429–452, 2012.

[26] A. Hospital, J. R. Goñi, M. Orozco, and J. L. Gelpí, "Molecular dynamics simulations: Advances and applications," *Advances and Applications in Bioinformatics and Chemistry*, pp. 37–47, 2015.

[27] H. Lee, M. Turilli, S. Jha, D. Bhowmik, H. Ma, and A. Ramanathan, "DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding," in *IEEE/ACM Third Workshop on Deep Learning on Supercomputers*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2019, pp. 12–19. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/DLS49591.2019.00007

[28] A. Brace, I. Yakushin, H. Ma, A. Trifan, T. Munson, I. Foster, A. Ramanathan, H. Lee, M. Turilli, and S. Jha, "Coupling streaming AI and HPC ensembles to achieve 100–1000× faster biomolecular simulations," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 806–816. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/IPDPS53621.2022.00083

[29] "MOF Generation on HPC," https://github.com/globus-labs/mof-generation-at-scale/. Accessed Mar. 2024.

[30] H. Park, X. Yan, R. Zhu, E. A. Huerta, S. Chaudhuri, D. Cooper, I. Foster, and E. Tajkhorshid, "A generative artificial intelligence framework based on a molecular diffusion model for the design of metal-organic frameworks for carbon capture," *Communications Chemistry*, vol. 7, no. 1, Feb. 2024. [Online]. Available: http://dx.doi.org/10.1038/s42004-023-01090-2

[31] L. Ward, J. G. Pauloski, V. Hayot-Sasson, R. Chard, Y. Babuji, G. Sivaraman, S. Choudhury, K. Chard, R. Thakur, and I. Foster, "Cloud services enable efficient AI-guided simulation workflows across heterogeneous resources," in *Heterogeneity in Computing Workshop*. IEEE Computer Society, 2023, https://arxiv.org/abs/2303.08803.

[32] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994. [Online]. Available: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1

[33] M. Zvyagin, A. Brace, K. Hippe, Y. Deng, B. Zhang, C. O. Bohorquez, A. Clyde, B. Kale, D. Perez-Rivera, H. Ma *et al.*, "GenSLMs: Genome-scale language models reveal SARS-CoV-2 evolutionary dynamics," *The International Journal of High Performance Computing Applications*, vol. 37, no. 6, pp. 683–705, 2023.

[34] A. Kamatar, M. Sakarvadia, V. Hayot-Sasson, K. Chard, and I. Foster, "Lazy Python dependency management in large-scale systems," in *IEEE 19th International Conference on e-Science*. IEEE, 2023, pp. 1–10.

[35] M. Copik, R. Böhringer, A. Calotoiu, and T. Hoefler, "FMI: Fast and Cheap Message Passing for Serverless Functions," Scalable Parallel Computing Laboratory, ETH Zurich, Tech. Rep., 2022.

[36] "Redis," 2023, https://redis.io/. Accessed Mar 2023.

[37] Snap Inc., "KeyDB: A database built for scale," https://github.com/Snapchat/KeyDB. Accessed Mar 2023.

[38] I. Foster, "Globus Online: Accelerating and democratizing science through cloud-based services," *IEEE Internet Computing*, vol. 15, no. 3, pp. 70–73, 2011.

[39] K. Chard, S. Tuecke, and I. Foster, "Efficient and secure transfer, synchronization, and sharing of big data," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 46–55, 2014.

[40] "UCX-Py," https://ucx-py.readthedocs.io/en/latest/. Accessed Mar 2023.

[41] "Py-Margo," https://github.com/mochi-hpc/py-mochi-margo. Accessed Mar 2023.

[42] J. G. Pauloski, K. Rydzy, V. Hayot-Sasson, I. Foster, and K. Chard, "Accelerating Python Applications with Dask and ProxyStore," 2024. [Online]. Available: https://arxiv.org/abs/2410.12092

[43] "C++ std::futures," https://en.cppreference.com/w/cpp/thread/future. Accessed Oct 2024.

[44] "Python Concurrent Futures," https://docs.python.org/3/library/concurrent.futures.html. Accessed Oct 2024.

[45] S. Wang, E. Liang, E. Oakes, B. Hindman, F. S. Luan, A. Cheng, and I. Stoica, "Ownership: A distributed futures system for Fine-Grained tasks," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 671–686. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/cheng

[46] "Apache Kafka," 2024, https://kafka.apache.org/. Accessed Feb 2024.

[47] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[48] "The Rust Language," https://rust-lang.org. Accessed Feb 2024.

[49] "Rust Documentation: The Dot Operator," https://doc.rust-lang.org/nomicon/dot-operator.html. Accessed Feb 2024.

[50] A. Bauer, H. Pan, R. Chard, Y. Babuji, J. Bryan, D. Tiwari, I. Foster, and K. Chard, "The Globus Compute Dataset: An open function-as-a-service dataset from the edge to the cloud," *Future Generation Computer Systems*, vol. 153, pp. 558–574, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X23004703

[51] W. F. Godoy, N. Podhorszki, R. Wang, C. Atkins, G. Eisenhauer, J. Gu, P. Davis, J. Choi, K. Germaschewski, K. Huck, A. Huebl, M. Kim, J. Kress, T. Kurc, Q. Liu, J. Logan, K. Mehta, G. Ostrouchov, M. Parashar, F. Poeschel, D. Pugmire, E. Suchyta, K. Takahashi, N. Thompson, S. Tsutsumi, L. Wan, M. Wolf, K. Wu, and S. Klasky, "ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management," *SoftwareX*, vol. 12, p. 100561, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2352711019302560

[52] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the Chameleon testbed," in *USENIX Annual Technical Conference*. USENIX Association, July 2020.

[53] A. Brace, L. Ward, H. Ma, and A. Ramanathan, "DeepDriveMD," Accessed Mar. 2024. [Online]. Available: https://github.com/ramanathanlab/deepdrivemd

[54] H. C. Baker and C. Hewitt, "The incremental garbage collection of processes," in *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1977, p. 55–59. [Online]. Available: https://doi.org/10.1145/800228.806932

[55] D. P. Friedman and D. Wise, "Aspects of applicative programming for parallel processing," *IEEE Transactions on Computers*, vol. C-27, no. 4, pp. 289–296, 1978.

[56] K. Fernandez-Reyes, D. Clarke, L. Henrio, E. B. Johnsen, and T. Wrigstad, "Godot: All the benefits of implicit and explicit futures," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), A. F. Donaldson, Ed., vol. 134. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 2:1–2:28. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2019.2

[57] K. L. Clark and S. Gregory, "A relational language for parallel programming," in *Conference on Functional Programming Languages and Computer Architecture*, 1981, pp. 171–178.

[58] K. Clark and S. Gregory, "Parlog: Parallel programming in logic," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 1, pp. 1–49, 1986.

[59] I. Foster and S. Taylor, *Strand: New concepts in parallel programming*. Prentice-Hall, Inc., 1989.

[60] I. Foster, R. Olson, and S. Tuecke, "Productive parallel programming: The PCN approach," *Scientific Programming*, vol. 1, no. 1, pp. 51–66, 1992.

[61] K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 1992, pp. 124–144.

[62] "PyTorch RPC," https://pytorch.org/docs/2.2/rpc.html#torch.distributed.rpc.rpc_async. Accessed Feb 2024.

[63] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: a distributed framework for emerging AI applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 561–577.

[64] "RabbitMQ," 2024, https://rabbitmq.com/. Accessed Feb 2024.

[65] T. Bicer, D. Gursoy, R. Kettimuthu, I. T. Foster, B. Ren, V. De Andrede, and F. De Carlo, "Real-time data analysis and autonomous steering of synchrotron light source experiments," in *2017 IEEE 13th International Conference on e-Science (e-Science)*, 2017, pp. 59–68.

[66] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke, "Software as a service for data scientists," *Communications of the ACM*, vol. 55, no. 2, p. 81–88, feb 2012. [Online]. Available: https://doi.org/10.1145/2076450.2076468

[67] R. Filguiera, I. Klampanos, A. Krause, M. David, A. Moreno, and M. Atkinson, "dispel4py: A Python framework for data-intensive scientific computing," in *2014 International Workshop on Data Intensive Scalable Computing Systems*, 2014, pp. 9–16.

[68] L. Liang, R. Filgueira, Y. Yan, and T. Heinis, "Scalable adaptive optimizations for stream-based workflows in multi-HPC-clusters and cloud infrastructures," *Future Generation Computer Systems*, vol. 128, pp. 102–116, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X21003824

[69] Message Passing Forum, "MPI: A message-passing interface standard," University of Tennessee, USA, Tech. Rep., 1994.

[70] C. Herath and B. Plale, "Streamflow programming model for data streaming in scientific workflows," in *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 302–311.

[71] J. Y. Choi, T. Kurc, J. Logan, M. Wolf, E. Suchyta, J. Kress, D. Pugmire, N. Podhorszki, E.-K. Byun, M. Ainsworth, M. Parashar, and S. Klasky, "Stream processing for near real-time scientific data analysis," in *2016 New York Scientific Data Summit (NYSDS)*, 2016, pp. 1–8.

[72] J. Chung, W. Zacherek, A. Wisniewski, Z. Liu, T. Bicer, R. Kettimuthu, and I. Foster, "SciStream: Architecture and toolkit for data streaming between federated science instruments," in *31st International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2022, p. 185–198. [Online]. Available: https://doi.org/10.1145/3502181.3531475

[73] A. R. Martinelli, M. Torquati, M. Aldinucci, I. Colonnelli, and B. Cantalupo, "CAPIO: a Middleware for Transparent I/O Streaming in Data-Intensive Workflows," in *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2023.

[74] C.-W. Lermen and D. Maurer, "A protocol for distributed reference counting," in *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, ser. LFP '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 343–350. [Online]. Available: https://doi.org/10.1145/319838.319875

[75] D. I. Bevan, "Distributed garbage collection using reference counting," in *PARLE Parallel Architectures and Languages Europe: Volume II: Parallel Languages Eindhoven, The Netherlands, June 15–19, 1987 Proceedings 1*. Springer, 1987, pp. 176–187.

[76] J. M. Piquer, "Indirect reference counting: A distributed garbage collection algorithm," in *Parallel Architectures and Languages Europe: Volume I: Parallel Architectures and Algorithms Eindhoven, The Netherlands, June 10–13, 1991 Proceedings*. Springer, 1991, pp. 150–165.

[77] L. Moreau and J. Duprat, "A construction of distributed reference counting," *Acta Informatica*, vol. 37, pp. 563–595, 2001.

[78] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *12th ACM Symposium on Operating Systems Principles*, ser. SOSP '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 202–210. [Online]. Available: https://doi.org/10.1145/74850.74870