Accelerating Communications in Federated Applications with Transparent Object Proxies

J. Gregory Pauloski
University of Chicago

Valerie Hayot-Sasson
University of Chicago

Logan Ward
Argonne National Laboratory

Nathaniel Hudson
University of Chicago

Charlie Sabino
University of Chicago

Matt Baughman
University of Chicago

Kyle Chard
University of Chicago
Argonne National Laboratory

Ian Foster
University of Chicago
Argonne National Laboratory

ABSTRACT

Advances in networks, accelerators, and cloud services encourage programmers to reconsider where to compute—such as when fast networks make it cost-effective to compute on remote accelerators despite added latency. Workflow and cloud-hosted serverless computing frameworks can manage multi-step computations spanning federated collections of cloud, high-performance computing (HPC), and edge systems, but passing data among computational steps via cloud storage can incur high costs. Here, we overcome this obstacle with a new programming paradigm that decouples control flow from data flow by extending the pass-by-reference model to distributed applications. We describe ProxyStore, a system that implements this paradigm by providing object *proxies* that act as wide-area object references with just-in-time resolution. This proxy model enables data producers to communicate data unilaterally, transparently, and efficiently to both local and remote consumers. We demonstrate the benefits of this model with synthetic benchmarks and real-world scientific applications, running across various computing platforms.

KEYWORDS

Data Communication and Storage, Distributed Computing, Federated Computing, Open-source Software, Python

1 INTRODUCTION

The function-as-a-service (FaaS) and workflow programming paradigms facilitate the development of scalable distributed applications. Programmers specify *what* task (e.g., function or workflow stage) to perform without regard to *where* they are executed; the FaaS or workflow system then handles the mechanics of routing each task to a suitable processor. FaaS systems often assume that tasks are independent, while in workflow systems tasks may be linked in a dependency graph (e.g., a directed acyclic graph). In both cases, it is common for all data movement to pass via a central location such as a FaaS service, workflow engine, shared file system, or task database, where task inputs and outputs can be stored persistently on stable storage. Such centralized approaches may lead to unnecessary communication [**? ?**] but facilitate the implementation of other useful features like re-execution of failed tasks or dynamic adjustments of task location.

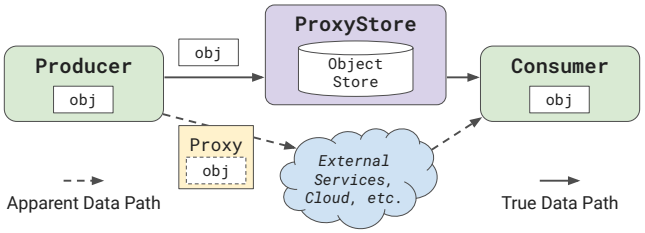

Figure 1: ProxyStore decouples the communication of object data from control flow transparently to the application. Data consumers receive lightweight proxies that act like the true object when used, while the heavy lifting of object communication is handled separately.

The passing of data among tasks via a central location become increasingly problematic when tasks are located on distinct computers. Consider a program that makes a function call `x=f()` to produce a value `x` that is to be consumed by a second function call `g(x)`. If `f()` and `g()` are dispatched to different computers C_a and C_b, respectively, then `x` must be transferred from C_a to C_b. Requiring that this transfer pass via a central location (e.g., FaaS service, workflow controller, shared file system) is inefficient, particularly if `x` is an intermediary value of no use to the client. Instead, it would be preferable to communicate `x` directly from `f()` to `g()`. To do this, we need methods for: (1) representing `x` such that `f()` and `g()` can produce and globally reference `x` and (2) communicating `x` from `f()` to `g()`, despite `f()` and `g()` running in different processes, compute nodes, or systems.

To address these challenges, we present ProxyStore, an abstraction for managing the routing of data between processes in distributed and federated Python applications. ProxyStore allows developers to focus, when writing and composing distributed applications, on logical data flow rather than physical details of where data reside and how data are communicated. This decoupling enables the dynamic selection of different data movement methods, depending on *what* data are moved, *where* data are moved, or *when* data are moved—a long-standing challenge in distributed application design [**? ? ? ?**]. The *proxy* programming model transparently provides pass-by-reference semantics and just-in-time object resolution to consumers. A proxy is lightweight and can be communicated efficiently via any means while its referenced object is communicated

transparently via optimal routes. By thus abstracting the use of specialized communication methods, the proxy paradigm improves code compatibility, performance, and productivity.

ProxyStore provides interfaces to common mediated communication channels (e.g., shared file systems, Globus [**?** **?** **?**], Redis [**?**]) and custom implementations that leverage the powerful communication technologies of high-performance computing (HPC) environments and enable direct communication between remote systems.

The contributions of this paper are:

- The design and implementation of the proxy model which is, to the best of our knowledge, the first system that transparently provides both pass-by-reference and pass-by-value semantics for distributed applications.
- Data transfer mechanisms that enable fast intra- and inter-site communication in various settings and an extensible framework for seamless integration of new technologies.
- Component level benchmarks of ProxyStore and comparisons to prior works.
- Experiments using ProxyStore to accelerate real-world federated science applications.

The ProxyStore framework is an open-source Python package available on GitHub and PyPI [**?** **?**].

The rest of this paper is as follows: **??** discusses related work in federated and distributed application design; **??** outlines ProxyStore design goals and introduces the core components; **??** describes the communication channels provided; **??** presents component-level benchmarks and real-world use cases; and **??** summarizes our contributions and future plans.

2 BACKGROUND AND RELATED WORK

Increasing hardware heterogeneity, faster and more reliable networks, and shifts in application requirements have motivated federated application design, i.e., applications that span several cloud, high-performance (HPC), edge, and personal systems. Here we discuss technologies that enable the management of computation across diverse systems.

Communication decoupling: The appropriate design for a distributed application's communication fabric depends on the decoupling needed among application processes. Eugster et al. [**?**] describe how decoupling can occur along space, time, and synchronization dimensions. Processes decoupled in *space* interact indirectly via a shared service (e.g., message queue or object store). A producer and consumer are decoupled in *time* if they need not be active at the same time. Decoupling in *synchronization* means that data production or consumption does not occur in the primary control flow, so that, for example, processes need not block on communication or can be notified asynchronously of events.

Prior work distinguishes *direct* communication channels from *mediated* channels where "the communication between participants is done over storage or other indirect means" [**?**]. Direct channels typically provide for rapid communication but prevent space and time decoupling among actors. Mediated channels necessarily provide space decoupling and can also provide time decoupling if the mediator (e.g., storage) persists for the entirety of the period over which any producers or consumers exist.

Data fabrics: Tuple spaces, such as in Linda [**?**], were early shared data fabrics. In the tuple-space model, producers post data as tuples in a shared distributed memory space, from which consumers can retrieve data that match a specified pattern. Tuple spaces have since been implemented in many languages including Python [**?**]. DataSpaces [**?**] provides a tuple-space-like interface to a virtual shared object space designed to support large-scale workflows composed of coupled applications. The shared space is implemented with the high-speed remote procedure call (RPC) and transfer provided by the Margo and Mercury RPC libraries [**?** **?** **?**]. WA-DataSpaces [**?**] extends the DataSpaces model to support data staging and predictive prefetching to improve data access times. The InterPlanetary File System (IPFS) is a decentralized, peer-to-peer file sharing network that provides content-addressing via a flat global namespace [**?**].

Network policies: Network access is a core problem in federated computing because policies vary between networks. Network address translation (NAT) and firewalls often prohibit outside access to local devices, thus preventing direct communication between hosts. These problems are particularly prevalent in scientific computing where experimental instruments are often in different locations from the data storage and analysis computers. At some sites, Science DMZs [**?**] permit bypassing firewalls under programmatic control [**?**]. Cross-site data transfer can be performed via cloud services (e.g., in Globus Compute [**?**]), but this adds latency and can be cost-prohibitive for data-intensive applications. SciStream [**?**] addresses these issues by using gateway nodes (e.g., data transfer nodes in a Science demilitarized zone (DMZ)) to facilitate fast memory-to-memory data transfers between remote hosts.

NAT traversal: A general solution for communication between two hosts behind separate NATs is via User Datagram Protocol (UDP) hole punching. In this model, a UDP connection is established between hosts by using a third-party, publicly accessible relay server that facilitates the connection [**?**]. For example, Globus Transfer uses such a mechanism for transfers between two Globus Connect Personal endpoints [**?**]. The FaaS Messaging Interface (FMI), modeled after the Message Passing Interface (MPI) [**?**], provides point-to-point and collective communication for serverless functions [**?**]. It supports both mediated channels, which use external storage accessible by all functions, and direct channels, which use Transmission Control Protocol (TCP) connections that, however, may not be accessible by all function invocations. When direct TCP communication is not possible, FMI uses a relay server and hole punching to establish a direct connection between function invocations. Libp2p defines a modular specification for developing peer-to-peer applications with support for NAT traversal [**?** **?**]. Implementations of the protocol are provided or planned for many popular languages.

Workflows: Scientific workflow systems (e.g., FireWorks [**?**], Parsl [**?**], Pegasus [**?**], Radical Pilot [**?**], Swift [**?**]), often include both intra- and inter-site data transfer functionality as a core feature, for movement both of input and output data between clients and execution environments and of intermediate data between tasks [**?** **?**]. Parsl, Pegasus, and Swift all enable transparent intra-site communication via shared file systems, and provide some support for inter-site communication via files. For example, Parsl supports movement of Python objects via ZeroMQ [**?**] sockets in a

hub-spoke architecture between the main Parsl process and workers; uni-directional file staging via Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP) (developers must specify URLs for downloading or uploading artifacts); and Globus-based data movement between sites based on user-supplied configuration information specifying the Globus endpoint for each site, in which case, Parsl inserts data transfer operations in the workflow graph and executes movement before/after task execution.

Function-as-a-Service: Cloud-hosted serverless frameworks (e.g., Amazon Web Services (AWS) Lambda [**?**], Azure Functions [**?**], Google Cloud Functions [**?**]) serialize input and output data along with a function request or result. Functions can also read and write from cloud object stores (e.g., AWS S3 [**?**]) and pass object IDs as function inputs or outputs. Globus Compute [**?**], formerly *func*X, is a cloud-managed serverless framework that supports remote execution across federated endpoints such as cloud machines, HPC clusters, edge nodes, and workstations. The Globus Compute cloud service routes each client task to a specified target endpoint and stores results until retrieved by the client. The cloud service is essential to providing the compute-anywhere features of Globus Compute but requires that all inputs and results be sent to, and stored in, the cloud (Redis servers hosted in AWS and S3), even if the Globus Compute client and endpoints are located in the same site, which introduces additional latency and costs. Globus Compute enforces a 5 MB task payload size limit to manage storage and egress costs.

3 DESIGN AND IMPLEMENTATION

Here we describe the goals and design of the framework, and detail the implementation choices necessary to enable the proxy model. ProxyStore provides four primary components: the `Proxy`, `Factory`, `Connector`, and `Store`. The ProxyStore design enables more features and greater flexibility compared to the de facto approaches for mediated communication in federated applications.

3.1 Assumptions

We make the following assumptions about usage model and target applications. (1) The application requires some combination of space, time, and synchronization decoupling (i.e., ProxyStore is not intended for highly synchronous applications). (2) The application can be described as a composition of dependent tasks that consume and produce Python objects. We target Python for its pervasiveness in the scientific and workflow systems communities and for the language features that make the proxy model possible. (3) Intermediate objects are written only once but may be read many times. Most task-based workflows fit this paradigm, especially those with pure functional tasks. (4) Objects need not be moved to a centralized store, but can stay where they are produced or be moved to where they are to be consumed. (5) Users may have their own object storage and communication backends that meet their performance and persistence requirements. Federated applications that employ FaaS and workflow systems fit these assumptions well.

3.2 Requirements

ProxyStore must support applications with any of the following attributes: (1) data can be produced in many places and must be globally accessible (including across NATs); (2) computation can be performed in many places, and regardless of location must be able to consume previously produced data and produce new objects that can then be accessed by others; (3) objects may be persistent (must be available for future unknown purposes) or ephemeral (e.g., an intermediate value that is produced by one function and consumed by another, and then never accessed again) and, thus, must exist as long as their associated proxies exist; (4) storage locations have varying reliability (e.g., persistent disk vs. in-memory) and performance; (5) multiple storage or communication methods may need to be employed within a single workload; and (6) data consumers need not know the communication method required to access data.

3.3 The `Proxy` and `Factory`

We meet these design requirements via the use of lazy, transparent object proxies that act as wide-area object references. The term *proxy* in computer programming refers to an object that acts as the interface for another object. Proxies are commonly used to add additional functionality to their *target* object or enforce assertions prior to forwarding operations to the target. For example, a proxy can wrap sensitive objects with access control or provide caching for expensive operations.

Two valuable properties that a proxy can provide are *transparency* and *lazy resolution*. A *transparent* proxy behaves identically to its target object by forwarding all operations on itself to the target. For example, given a proxy `p` of an object `x`, the types of `p` and `x` will be equivalent: i.e., `isinstance(p, type(x))` and any operation on `p` will invoke the corresponding operation on `x`.

A *lazy* or *virtual* proxy provides just-in-time resolution of its target object. The proxy is initialized with a *factory* rather than the target object. A factory is an object that is callable like a function and returns the target object. The proxy is *lazy* in that it does not call the factory to retrieve the target until it is first accessed—a process that is referred to as *resolving* the proxy. Functionally, proxies have both pass-by-reference and pass-by-value attributes. The eventual user of the proxied data gets a copy, but unnecessary copies are avoided when the proxy is passed among multiple functions.

This factory-proxy paradigm provides powerful capabilities. The proxy itself is a lightweight reference to the target that can be communicated cheaply between processes and systems. The proxy is self-contained because the proxy always contains its factory and the factory includes all logic for data retrieval and manipulation. That is, the proxy does not need any external information to function correctly. Proxies eliminate the need for shims or wrapper functions that convert objects into forms expected by downstream code. Rather, the proxy can be passed to any existing method or function and the conversion is handled internally by the factory. The consumer code is unaware that the resulting object is anything other than what it expected. Proxies also have other advantages. For example: lazy resolution can help amortize costs and avoids unnecessary computation/communication for objects that are never used; nested proxies can enable partial resolution of large objects; and proxies can be moved in place of confidential data (e.g., patient health information) while ensuring that the data can be resolved only where permitted.

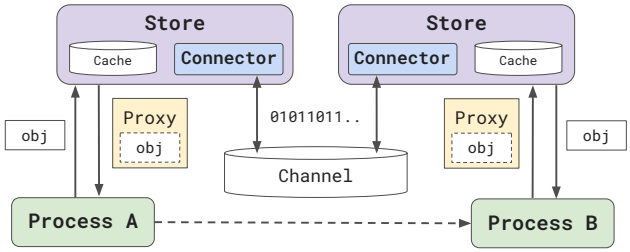

Figure 2: Processes interact with a `Store` to proxy objects, and proxy consuming processes will transparently interact with the local `Store` instance. The underlying communication is executed using the `Connector` interface.

```python
from proxystore.connectors.redis import RedisConnector
from proxystore.proxy import Proxy
from proxystore.store import Store

def my_function(x: MyDataType) -> ...:
    # x is resolved from "my-store" on first use
    assert isinstance(x, MyDataType)
    # More computation...

store = Store('my-store', RedisConnector(...))

# Store the object and get a proxy
my_object = MyDataType(...)
p = store.proxy(my_object)
assert isinstance(p, Proxy)

my_function(p)  # Succeeds
```

Listing 1: Example of ProxyStore usage.

ProxyStore implements lazy transparent object proxies. The `Proxy` class implementation is initialized with a factory and intercepts any access to a proxy instance attribute or method; calls the factory to resolve and cache the target object, if the target has not yet been resolved; and forwards the intercepted action to the cached target. The factory used to initialize a proxy can be any callable Python object (i.e., any object that implements `__call__`, such as lambdas, functions, and callable class instances). `Proxy` modifies its own pickling behavior to include only the factory, not the target, when serializing the proxy, so as to ensure that (1) proxies are small when communicated and (2) a proxy can still be resolved after being communicated to another process.

When a proxy is used, its factory must be able to resolve its target object efficiently. In a distributed application, this means a factory must be resolvable when the producer and consumer processes exist independently in space or time. Facilitating this property when processes can exist in the same network or across multiple requires careful consideration for the underlying mediated communication channels used. We discuss how we achieve this goal with the `Connector` in the following section.

3.4 The Connector

The `Connector` is a low level interface to a mediated communication channel. In order to support a wide range of application requirements, we have designed ProxyStore to be extensible to support various mediated channels that can support different space and time decoupling patterns. The `Connector` protocol defines how a client can connect to or operate on a mediated channel, and a `Connector` implementation must provide four primary operations: `evict`, `exist`, `get`, and `put`. The operations act on byte-string data and keys. E.g., `put` takes a byte-string to put in the mediated channel and returns a uniquely identifying key (a tuple of metadata); the byte-string is retrievable by calling `get` on the key. We chose this model so that third-party code can easily provide new `Connectors` that are plug-and-play with the rest of ProxyStore's features. A `Connector` implementation can be either an interface to an external mediated channel (e.g., a Redis server) or a mediated channel itself. ProxyStore provides many `Connector` implementations that fit both of these categories which we describe further in **??**.

3.5 The Store

The `Store` is the high-level interface used by applications to interact with ProxyStore as shown in **??**. A `Store` is initialized with a `Connector` instance (a dependency injection pattern) and provides additional functionality on top of the `Connector`. Similar to the `Connector`, the `Store` exposes `evict`, `exist`, `get`, and `put` operations; however, these operations act on Python objects rather than byte strings. The `Store` (de)serializes objects before invoking the corresponding operation on the `Connector`; custom (de)serialize functions can be registered with the `Store` if needed. The `Store` also provides caching of operations to reduce communication costs, with caching performed after deserialization to avoid duplicate deserializations.

However, rather than the application invoking the aforementioned operations directly, the `proxy` method, also provided by the `Store`, is used. Calling `Store.proxy` puts an object in the mediated channel via the `Connector` instance and returns a proxy (**??**). The object is serialized before being put in the mediated channel; a factory is generated, containing the key returned by the `Connector` and additional information necessary to retrieve the object from the mediated channel; and then a new proxy, internalized with the factory, is returned.

An evict flag can be passed when creating a proxy. If set, the proxy will evict the object from the mediated channel when first resolved. Subsequently, the `proxy` operation, alone, is a complete interface to an object store because the `proxy` method handles the `put` operation and the proxy resolution process handles `get`/`evict`.

The `Proxy` and `Factory` instances created by a `Store` provide functionality for asynchronously resolving the target object in a background thread using the `resolve_async` function. This is useful in code which expects a proxy and wants to overlap the communication of the proxy resolution with other computations.

`Store` instances are registered globally within a process by name so that initialization is performed only once, caches are shared, and stateful connection objects in the `Connector` are reused. Consider a `Connector` instance C and corresponding `Store` S. S has been registered in process P_a with name "my-store" and is used to create a proxy p. If p is resolved on a remote process P_b where a `Store` with name "my-store" has not yet been registered, p will initialize and register a new `Store` instance named "my-store" with the appropriate `Connector` when p is resolved. This is possible because

Table 1: Summary of provided `Connector` implementations.

Connector	Storage	Intra-Site	Inter-Site	Persistence
File	Disk	✓		✓
Redis	Hybrid	✓		✓
Margo	Memory	✓		
UCX	Memory	✓		
ZMQ	Memory	✓		
Globus	Disk		✓	✓
Endpoint	Hybrid	✓	✓	✓

p's factory, created in process P_a, includes the appropriate metadata necessary to recreate C and S in process P_b. Subsequent proxies created by any `Store` with the same name and resolved in P_b will then use the registered `Store` rather than initializing a new one.

4 SUPPORTED CONNECTORS

All `Connector` implementations are built on mediated, byte-level data storage. Data storage methods are broadly classified as in-memory or on-disk. Mediated channels use one or both methods, depending on performance and persistence aims. The proxy abstraction provided by the `Store` enables a producer to unilaterally (i.e., without the agreement of the receiver) choose the best mediated channel for object communication.

Data storage may be local to the process or machine, within the same network, or at a remote site. Here, we describe the various `Connector` implementations provided out-of-the-box that can be used with the `Store` that support in-memory and on-disk data storage within and between sites (summarized in **??**). We also describe an implementation provided `MultiConnector` abstraction which enables intelligent routing of objects across connectors.

4.1 Intra-Site Communication

Various technologies, such as shared file systems, TCP/UDP sockets, and remote distributed memory access (RDMA), enable data transfers between nodes on the same local area network: i.e., not located behind different NATs.

4.1.1 On-disk Storage. For large objects or data that needs to be persisted, ProxyStore provides the `FileConnector` for mediated communication via a shared file system. The `FileConnector` is initialized with a path to a data directory in which proxied objects can be serialized and written (and then read) as files.

4.1.2 Hybrid Storage. The `RedisConnector` uses an existing Redis [**?**] or KeyDB [**?**] server as the mediator. Redis provides a hybrid between in-memory and on-disk data storage with low-latency, easy configuration, persistence, and optional resilience via replication across nodes. The `RedisConnector` implementation is only 31 lines of Python code, exemplifying the ease with which the proxy model can be extended to other mediated communication methods via the `Connector` protocol.

4.1.3 Distributed In-memory Storage. Distributed memory backends for intra-site communication permit applications to benefit from increased memory capacity and scalability. Two implementations are provided, `MargoConnector` and `UCXConnector`, to leverage rapid communication on high-speed networks by using the Py-Mochi-Margo [**?**] and UCX-Py [**?**] libraries, respectively. A third

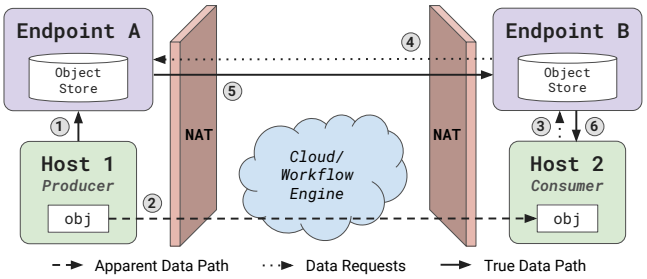

Figure 3: Data flow when transferring objects via proxies and PS-endpoints between sites. The proxy gives the appearance that data flows through the entire application, but the actual data transfer is performed via a peer connection between the PS-endpoints at the producing and consuming sites.

implementation, `ZMQConnector`, uses ZeroMQ for communication and is provided as a fallback for compatibility. When one of these connectors is initialized for the first time in a process, it spawns a process that acts as the storage server for that node. Thus, these connectors act as interfaces to these spawned servers which make up the actual distributed in-memory store. These distributed storage methods are elastic—expanding as proxies are propagated to new nodes—and enable the use of state-of-the-art direct communication methods in a mediated fashion.

4.2 Inter-Site Communication

ProxyStore enables data transfer between computers at different sites (and also between computers at the same site that are located behind different NATs) by using disk-to-disk solutions for bulk data and memory-to-memory solutions for low latency.

4.2.1 On-disk Storage. Bulk file transfers between sites are ubiquitous in scientific applications. To support such transfers, the `FileConnector` is extended as the `GlobusConnector` to use Globus to move object files between sites. Globus transfer supports efficient, secure, and reliable file movement and is widely adopted across computing centers with more than 20 000 active endpoints. Globus Connect software is easily deployed on computers without an existing endpoint.

The `GlobusConnector` is initialized with a mapping of hostname regular expressions to a tuple of `(Globus Endpoint UUID, Endpoint path)`. A proxy, while resolving itself, will match the hostname of the current system to the provided hostname regular expressions to determine the directory on the local endpoint with the transferred files. `GlobusConnector` keys are the tuple `(object_id, task_id)` where the `task_id` is the Globus transfer task ID. A proxy will wait for the transfer task to succeed before resolving itself or raise an error if there is a Globus transfer failure.

For efficient movement of many objects, the `Store` provides a `proxy_batch` method that will invoke a batch transfer of proxied objects as a single Globus transfer.

4.2.2 In-memory Storage. A common pattern in inter-site applications is the use of a centralized orchestrator that can communicate with all sites and mediates the control flow between actors across the sites. A simple example is a cloud-hosted queue of tasks, which

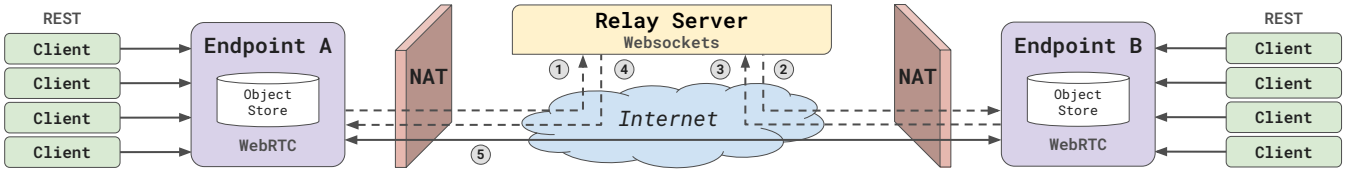

Figure 4: Client requests directed to any PS-endpoint are forwarded to the correct PS-endpoint via a peer connection. The peer connections are opened by using UDP hole-punching and a publicly accessible relay server. When PS-endpoint *A* wants to connect to PS-endpoint *B*, *A* asks the relay server *R* to forward a session description protocol (SDP) [?] to *B* (1 and 2). This description contains information about how the two peers can connect, such as what protocols they support. *B* receives *A*'s session description from *R* and replies with *B*'s session description (3 and 4). *A* and *B* then generate interactive connectivity establishment (ICE) candidates [?] (i.e., public IPs and ports to try for the connection) which they exchange via *R*. Once *A* and *B* have exchanged ICE candidates, they can connect by completing the hole punching process (5).

actors at each site poll to obtain tasks to execute or to place new tasks on the queue. In this model, data producers may not always know where data are eventually needed, but it can also be prohibitively expensive (monetary or overhead) to store data in the cloud or in some other central service. The proxy model allows applications to pass data by reference across sites and perform the underlying communication more directly, avoiding additional overheads of unnecessary data movements. ProxyStore includes a ProxyStore endpoint (PS-endpoints) model that facilitates direct data transfer between sites as shown in **??**.

PS-endpoints are in-memory object stores, with optional on-disk storage if host memory is insufficient or data persistence is required. PS-endpoints are managed with the `proxystore-endpoint` command-line interface. Clients use the `EndpointConnector` to interact with PS-endpoints, and object keys are the tuple `(object_id, endpoint_id)`. If a PS-endpoint receives an operation request on a key with an `endpoint_id` that is not its own, the PS-endpoint establishes a peer connection to the target PS-endpoint and forwards the request.

Peer-to-peer communication between PS-endpoints is achieved via the Web Real-Time Communication (WebRTC) standard [**? ?**]—specifically, by using the *RTCPeerConnection* and *RTCDataChannel* components of the `aiortc` open-source WebRTC implementation [**?**]. The RTCPeerConnection handles the establishment of peer connections across firewalls using NAT traversal and hole punching, as described in **??**; security; and connection management. The RTCDataChannels are associated with an RTCPeerConnection and enable bidirectional transfer between peers; data are transported over the channel via SCTP (Stream Control Transmission Protocol) over DTLS (Datagram Transport Layer Security).

PS-endpoints use a publicly accessible relay server or signaling server to facilitate the creation of RTCPeerConnections. The process of establishing the connection via the relay server is illustrated in **??**. Once a peer connection is established, the PS-endpoints maintain the connection until one of the PS-endpoints is stopped; the connection is re-established if lost for any reason, e.g., due to a PS-endpoint going offline temporarily. The hosting requirements for the relay server are minimal because establishing a peer connection only requires the relay server to exchange a few small (O(KB)) messages between the peers. We provide a WebSocket-based [**?**] relay server implementation that can be self hosted.

PS-endpoints are single-threaded, asyncio applications. When started, they connect and register with the relay server, and the relay server assigns a unique UUID if not already assigned. An asyncio task is created which listens on the WebSocket connection with the relay server for incoming peering requests and responds appropriately. Once a peer connection is established, the PS-endpoint also listens for and responds to incoming requests from its peers.

4.3 The `MultiConnector` Abstraction

Sophisticated applications that employ multiple data communication patterns can benefit from using multiple types of mediated communication (i.e., `Connector` implementations). Rather than creating multiple `Store` instances and a policy directing when to use each instance, ProxyStore provides the `MultiConnector` abstraction, which is initialized with a mapping of `Connector` instances to policies, to indicate how each `Connector` should be used. Thus, an application can use a single `Store` instance, and operations will be routed transparently and automatically to the appropriate `Connector`. Policy definitions are flexible and can be extended by developers. An example policy may include minimum and maximum object sizes, representing the ideal operating range for that `Connector`; tags denoting the sites at which the `Connector` is accessible (e.g., a `MargoConnector` is available within a single cluster, while an `EndpointConnector` is available at multiple sites); and a prioritization function for breaking ties when multiple `Connector` instances are otherwise suitable for a given object.

`Store` operations accept additional keyword arguments that are passed to the corresponding `Connector` method. The `put` and `proxy` methods of `MultiConnector` take a set of optional constraints on the data being stored. These constraints, as well as other metadata (object type or size, location, etc.), are matched against each policy of each `Connector` managed by the `MultiConnector`. If no match is found then an error is raised, although deployments may often prefer to provide a low priority fallback with no constraints.

5 EVALUATION

We evaluate the component-level performance of ProxyStore, quantify overhead reductions in compute frameworks when using ProxyStore, and demonstrate the use of ProxyStore in three real-world scientific applications. For brevity, we use the term `XStore` to mean we are using a ProxyStore `Store` initialized with an `XConnector`

```python
from globus_compute_sdk import Executor
from proxystore.connectors.redis import RedisConnector
from proxystore.store import Store

def my_function(x: MyDataType): ...

store = Store("store-name", RedisConnector(...))
data = store.proxy(...)

with Executor(...) as gce:
    fut = gce.submit(my_function, data, ...)
    fut.result()
```

Listing 2: Example ProxyStore usage with Globus Compute.

for communication. E.g., `RedisStore` is a `Store` initialized with a `RedisConnector`.

We performed experiments using six machines: Theta, Polaris, Perlmutter, Frontera, Midway2, and Chameleon Cloud. Theta and Polaris are at Argonne National Laboratory. Theta is a 4392-node Intel Knights Landing (KNL) cluster. The 560-node Polaris has four NVIDIA A100 GPUs per node. NERSC's Perlmutter cluster has 1536 NVIDIA A100 GPU nodes and 3072 AMD EPYC CPU nodes. We use the login nodes of Midway2 at the University of Chicago and the Texas Advanced Computing Center's (TACC) Frontera cluster as clients to distributed applications running on the aforementioned systems. Chameleon Cloud [**?**] provides bare-metal compute nodes.

5.1 ProxyStore with FaaS

We first evaluate ProxyStore with the federated FaaS platform Globus Compute [**?**], with the goal of quantifying the performance gains that may be achieved with minimal code changes to the producer and no changes to the compute framework.

To quantify the extent to which passing task inputs with proxies can reduce data transfer overheads, we perform experiments with Globus Compute where we execute no-op and 1 s sleep tasks with payload sizes from 10 bytes to 100 MB (**??**). We use four different configurations of Globus Compute clients and endpoints. (1) Theta → Theta: Client and tasks all run on the same Theta node. (2) Perlmutter Login → Perlmutter Compute: Client runs on a Perlmutter login node and tasks on a Perlmutter compute node. (3) Midway2 → Theta: Client runs on a Midway2 login node and tasks on a Theta compute node. (4) Frontera → Theta: Client runs on a Frontera login node and tasks on a Theta compute node. In the first two scenarios, the client and task execute in the same site and thus we compare the round-trip time when data are moved via the Globus Compute cloud service to data movement via ProxyStore's `FileStore`, `RedisStore`, and `EndpointStore`. In the latter two scenarios, the client and task execute in different sites, so we compare the baseline to ProxyStore's `EndpointStore` and `GlobusStore`. We also compare with a configuration in which data are moved to the Globus Compute endpoint by using the InterPlanetary File System (IPFS) [**?**]. IPFS is a peer-to-peer distributed file system, so we treat the Globus Compute client and Globus Compute endpoint as two nodes of the distributed file system. In no-op tasks, we ensure that the proxy is resolved even though no computation is performed, and in the sleep tasks, we begin asynchronously resolving the proxy before sleeping and then wait on the asynchronous resolve after the sleep to simulate overlapping proxy resolution with compute.

The baseline round-trip time, where data are transferred along with the task request to the Globus Compute cloud service, increases with data size up to the Globus Compute limit. In the first two scenarios where the client and task execution occur at the same site, all three ProxyStore options eliminate the Globus Compute data transfer overhead. This was achieved with only two client-side lines of code: one to initialize the `Store` and one to proxy task inputs before submitting to Globus Compute (**??** lines **??**–**??**). The asynchronous resolve in the sleep task requires one additional line of code in the task itself, but the overlap of communication and compute can yield benefits.

In the inter-site cases where the clients run on Midway2 or Frontera login nodes and execute tasks on Theta, we use the `GlobusStore` and `EndpointStore`. `GlobusStore` performance is not competitive with the Globus Compute baseline up to Globus Compute's payload limit. The performance is a consequence of Globus transfer's hybrid software-as-a-service model, which results in high bandwidth for larger transfers but not low latency for small transfers. However, the benefits of Globus transfer become substantial as data sizes grow beyond those used in this experiment. The `EndpointStore` outperforms the baseline, except for no-op tasks between Frontera and Theta where the performance is comparable. For the largest (100 MB) payloads, `EndpointStore` performance is less than the theoretical peak of the connection. We investigate this discrepancy further in **??**.

We also compare to IPFS for inter-site data transfer. Task data are written to disk, the file is added to IPFS, and the content ID of the IPFS file is passed as input to the Globus Compute task. When the Globus Compute task is invoked, IPFS is used to retrieve the file, and the data are read back into memory. Whereas ProxyStore required two extra client side lines of code, IPFS support required 13 extra lines of code on both the client and task. The performance of PS-endpoints and IPFS for no-op tasks between Midway2 and Theta are within run-to-run variances of each other. PS-endpoints are faster with the one second sleep tasks because of the asynchronous resolution of proxies. PS-endpoints outperform IPFS for Frontera to Theta transfers due to Frontera having a slower file system and slower transfers between the IPFS peers compared to the Midway2 → Theta scenario. IPFS and PS-endpoints address a different set of problems—IPFS is designed for decentralized and persistent sharing of content-addressable files; however, IPFS has a mature peer-to-peer transfer protocol which we can use as a point of comparison to show that PS-endpoints can outperform IPFS.

We repeat these experiments with the distributed in-memory connectors described in **??** and compare performance to DataSpaces, a shared-space abstraction designed for large-scale scientific applications. The experiments were executed on Polaris, which has a high-performance HPE Slingshot 11 network, and on two Chameleon Cloud nodes with a Mellanox Connect-X3 40GbE InfiniBand interconnect. **??** shows that the baseline cloud transfer and ProxyStore alternatives all exhibit similar performance at data sizes <1 GB, after which bandwidth dominates performance.

`MargoStore` and `UCXStore`, which both leverage RDMA, achieve the best overall performance on Polaris. However, `UCXStore` performs measurably worse than `MargoStore` and `RedisStore` for larger data sizes on Chameleon. We suspect the disparity is a result

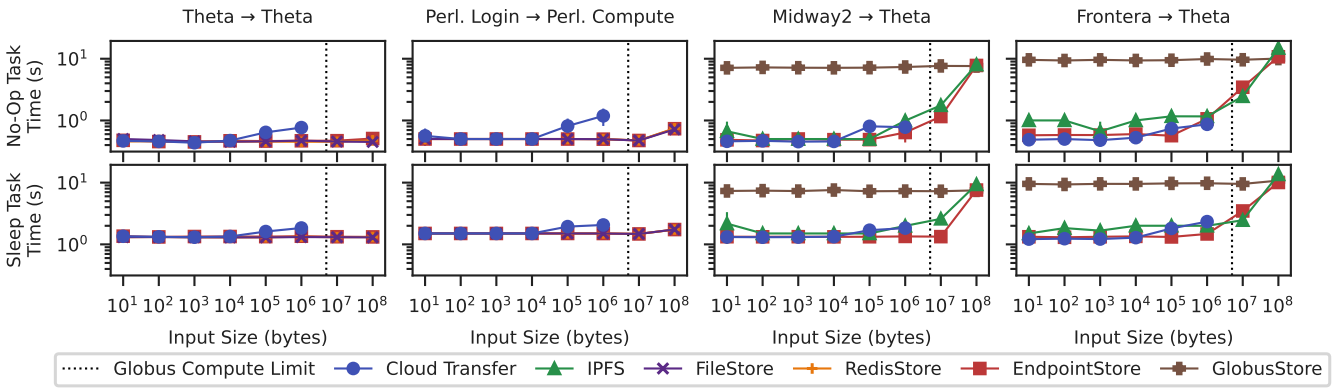

Figure 5: Average performance for round-trip Globus Compute no-op (top) and 1 s sleep tasks (bottom), for intra-site (two left columns) and inter-site (two right columns) configurations. In intra-site configurations, we compare baseline input data transfer via cloud to ProxyStore's `FileStore`, `RedisStore`, and `EndpointStore`. For inter-site, we compare to IPFS and ProxyStore's `EndpointStore` and `GlobusStore`. Dashed lines denote the 5 MB Globus Compute payload size limit for transfer via the cloud; ProxyStore can handle >5 MB task payloads without modifying task code to communicate via alternate means. Error bars denote standard deviation.

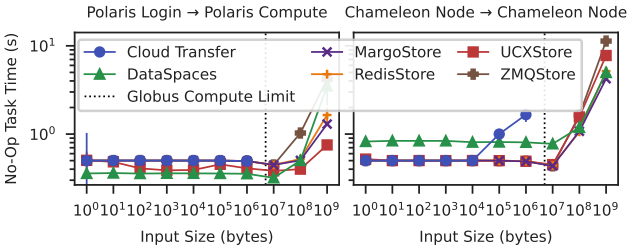

Figure 6: Average round-trip performance of no-op Globus Compute tasks on Polaris and Chameleon Cloud for the baseline cloud transfer via the Globus Compute service, ProxyStore centralized stores (`RedisStore`), ProxyStore distributed in-memory stores (`MargoStore`, `UCXStore`, `ZMQStore`) and DataSpaces. Error bars denote standard deviation.

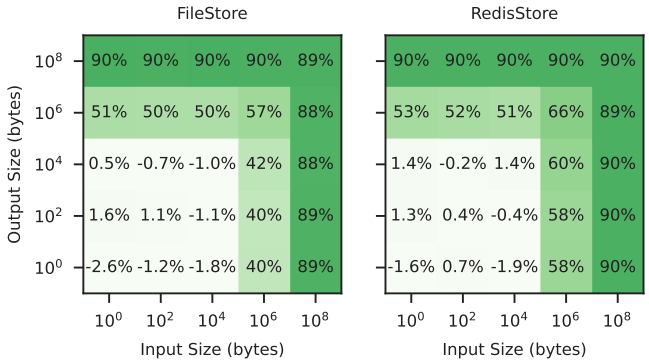

Figure 7: Percent improvements in task round-trip time when using ProxyStore to move data vs. Colmena's default method with Parsl. Each task configuration is repeated 100 times, and the median time is used to compute the improvement.

of the network differences between the two systems. While we expect DataSpaces and `MargoStore` to perform similarly because both use Margo for the transport layer, `MargoStore` outperforms DataSpaces on both systems. We observed prominent startup overheads, particularly for smaller transfers, with DataSpaces on Chameleon.

We focus on FaaS for HPC and choose FuncX because it is designed to coordinate computation across federated resources (e.g., cloud, HPC, and edge devices). However, ProxyStore is agnostic to the compute framework and will work with other FaaS systems. We expect comparable performance characteristics since Globus Compute's data storage and communication mechanisms are similar to cloud-specific FaaS systems.

5.2 ProxyStore with Workflow Systems

Colmena is a Python library for steering large ensembles of simulations [**?**]. Colmena applications contain three components: (1) a "Thinker," one or more agents that create tasks and consume results; (2) a "Task Server," which coordinates tasks to be executed by using a workflow engine (here, Parsl); and (3) workers which execute the tasks and return results to the Task Server. We integrate ProxyStore into Colmena at the library level. Users can register a `Store` and associated threshold for each task type. Task inputs or results greater than the threshold will be proxied before the task is sent to the Task Server. Passing proxies with the task can alleviate overheads in the Task Server and underlying workflow system.

We investigate overhead improvements in **??**, where we report the percent improvement over the baseline of median round-trip task times. We execute a series of no-op tasks using Colmena and Parsl with varied input and output sizes. The Thinker, Task Server, and worker are co-located on a single Theta node to isolate effects of the network. Neither ProxyStore's caching capabilities nor asynchronous resolving of proxies are used. For small data sizes (<100 KB), any improvements in overhead in Colmena are largely

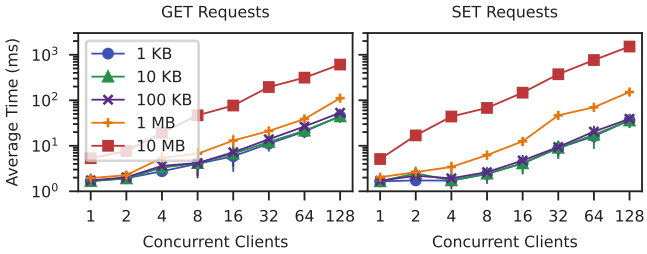

Figure 8: Average client `get` and `set` request times to a single PS-endpoint with respect to payload size and concurrent clients issuing the same request. Error bars denote standard deviation.

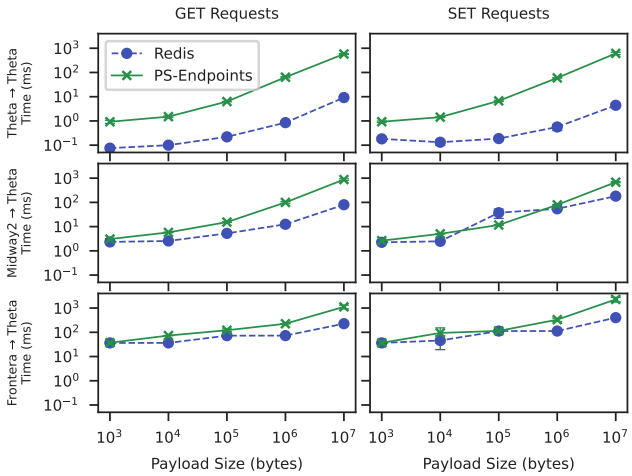

Figure 9: Average `get` and `set` times, over 1000 requests, between two PS-endpoints, with error bars showing the standard deviation. Comparisons are made to hosting a Redis server on the target site and opening an SSH tunnel when the two sites are different. The PS-endpoint configuration has one more hop (client—local endpoint—remote endpoint) than Redis (client—remote Redis).

negated by the additional overhead of proxying the data (i.e., I/O with storage). However, ProxyStore yields 40–60% improvements in overhead for 1 MB data sizes and 88–89% for 100 MB data sizes. This exemplifies why passing by proxy can be invaluable in distributed systems with many interconnected components. Proxies can be passed around cheaply while ensuring that data are only communicated between producer and consumer.

5.3 ProxyStore Endpoint Performance

To better understand the characteristics of PS-endpoints, we next study the times taken for client-to-PS-endpoint requests and PS-endpoint-to-PS-endpoint requests.

5.3.1 Client Access. In **??**, we show average per-request times for `get` and `set` operations versus the number of concurrent clients making the same request and for varied payload sizes. Each client makes 1000 requests, and the experiment was performed with Python 3.11 on a Perlmutter CPU node. Response times scale linearly with number of clients for more than two concurrent clients, and also scales with payload size. This is reasonable given that the proof-of-concept PS-endpoint implementation is single-threaded. Handling many concurrent workers with low latency is better suited for another mediated communication channel such as Redis.

5.3.2 Endpoint Peering. The primary use case for PS-endpoints is transfers between different sites. Thus, we measure request times between PS-endpoints as a function of payload size (see **??**). We consider three scenarios: requests between two PS-endpoints on different Theta nodes, which serves as a baseline; requests between PS-endpoints on Midway2 and Theta; and requests between PS-endpoints on Frontera and Theta. These scenarios differ in latency—packets need only travel tens of meters in the first scenario but 1500 kilometers in the third—and bandwidth—the first scenario can utilize the high bandwidth Aries Dragonfly network of Theta while the latter must cross multiple network boundaries. While no system provides equivalent features to PS-endpoints, we compare its performance to that of a Redis server hosted on the target site with a (manually created) secure shell (SSH) tunnel between the two sites. While in practice SSH tunnels can be fragile and difficult to configure (e.g., to authenticate automatically), they are commonly used by workflow systems [**? ?**] and the comparison can help highlight strengths and weaknesses of the PS-endpoint implementation.

We observe that Redis with SSH is generally faster than PS-endpoints, a result for which we identify two primary reasons. First, the PS-endpoint configuration has one more hop than the Redis configuration because two endpoints must be used in contrast to a single Redis server and SSH tunnel. This factor is most prevalent in the Theta-to-Theta scenario where network latency is minimal so the overhead of the extra hop dominates. Second, we discovered that the `aiortc` RTCDataChannel cannot fully utilize the available bandwidth between sites. This is why the difference in performance between PS-endpoints and Redis increases at larger data sizes. A simple test where we established an RTCDataChannel between a process on Frontera and another on Theta achieved a maximum bandwidth of 80 Mbps, a fraction of the full bandwidth available. This is because computing centers throttle UDP connections to avoid congestion, and `aiortc` congestion control is slower than other congestion control algorithms like Google's BBR [**?**]. We support multiplexing data transfer over multiple RTCDataChannels; however, the single-threaded asyncio model is unable to benefit from multiplexing over more than a couple RTCDataChannels. Despite these networking limitations, the performance of PS-endpoints is still competitive with Redis for long distance transfers while not requiring SSH tunnels or open ports.

5.4 Application: Real Time Defect Analysis

A common pattern in scientific applications is to transfer data produced by an experiment to a compute facility for analysis. For example, Argonne National Laboratory's transmission electron microscopy facility uses Globus Compute to invoke a machine-learned segmentation model to quantify radiation damage in acquired images, dispatching this computation to an HPC facility for fast GPU

Table 2: Round-trip task times for the real-time defect analysis application. The Globus Compute endpoint is hosted on a Polaris login node and the tasks are executed on a Polaris compute node. In the Globus Compute baseline and `FileStore` configurations, the client (simulating an experimental setup) is hosted on Theta, and the client is hosted on Midway2 in the `EndpointStore` configuration. Transferring task inputs and outputs via ProxyStore yields >30% performance improvements in intra- and inter-site task execution.

Configuration	Proxied	Time (ms)	Improvement
Globus Compute baseline	—	3411 ± 389	—
`FileStore`	Inputs	2318 ± 130	32.1%
	Inputs/Outputs	2160 ± 46	36.6%
`EndpointStore`	Inputs	2375 ± 98	30.4%
	Inputs/Outputs	2280 ± 107	33.2%

inference. We modify an open-source real time defect analysis application [**?**] to create and send proxies of images, rather than the actual images. We create a test deployment to mirror the production environment with remotely located instruments and compute.

We measure the baseline round-trip task time for inference on 1 MB images and compare to `FileStore` and `EndpointStore` (**??**). In all cases, we use a Globus Compute endpoint on a Polaris login node that executes tasks on a Polaris compute node. In the Globus Compute baseline and `FileStore` cases, our client (i.e., simulated beam facility) is hosted on a Theta login node, and in the `EndpointStore` case, when the client is on Midway2, with PS-endpoints on both Midway2 and a Polaris login node. We test with only the input images being proxied and with both the input images and inference outputs being proxied. Note that in the former, the code executed on the Globus Compute endpoint is unchanged, while the latter required two additional lines of task code to proxy the output by using the same `Store` that was used to resolve the input proxy.

We see in **??** that ProxyStore improves round-trip task times by 32.1% and 30.4% with `FileStore` and `EndpointStore`, respectively, when only the inputs are proxied. Further improvements of a few percentage points can be gained if the downstream code also returns proxies. We note that ProxyStore enabled greater flexibility in terms of how clients interact with tasks executed on the Globus Compute endpoint. Each client can choose its preferred communication method, depending on the mediated communication channels available from itself to the Globus Compute endpoint.

5.5 Application: Federated Learning

Federated learning (FL) [**?**] is an increasingly popular approach to distribute machine learning (ML) training across, often edge [**? ?**], devices. In FL, an aggregator node initializes an ML model and shares it with edge devices to train the model on their own private data in small batches. Once the edge training is complete, the locally-trained models are returned to the aggregator node to "average" the model to create a new global model. This new global model is then shared again with the edge devices for further training. In FL only the model is transferred across the network; the distributed edge devices' data are never shared.

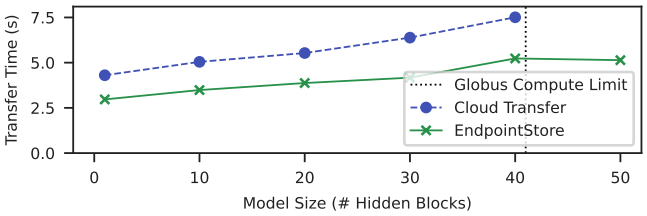

Figure 10: Average transfer times for the federated learning use case. PS-endpoints greatly reduce transfer times between nodes compared to cloud transfer. In addition, without ProxyStore, we are unable to transfer models larger than ~40 hidden blocks due to cloud transfer limits.

Here, we demonstrate the applicability and benefit of ProxyStore not only for FL use cases but edge computing workflows in general. Due to constrained capacity of edge devices, limited connectivity, and application requirements, making effective use of networks and providing low latency is often crucially important [**?**]. ProxyStore allows for FL control to be separated from data movement, enabling aggregation to occur *anywhere*, and for models to be transferred directly between edge and aggregation nodes when needed.

Our application is implemented using FLoX [**?**], a FL framework which uses Globus Compute to orchestrate training of TensorFlow [**?**] models. Our application trains a convolutional neural network for image classification with the Fashion-MNIST benchmark dataset [**?**]. We increase the number of hidden layers of the neural network to show ProxyStore's ability to support larger models compared to a purely FaaS-based approach. We use the same test bed as used in [**?**] to deploy our application across four edge devices. **??** shows the transfer time as we increase the number of model parameters when using Globus Compute or using Globus Compute and ProxyStore. We see that ProxyStore both reduces transfer time and also enables use of larger models. In the cases where Globus Compute is able to complete the model transfer, ProxyStore is able to reduce transfer time by ~68% on average. Further, ProxyStore can be used to implement hierarchical model aggregation, where sets of edge-trained models are aggregated in a distributed fashion.

5.6 Application: Molecular Design

We adapt an open source molecular design workflow to use the `MultiConnector` for communication between tasks. The workflow uses a mix of quantum chemistry simulations and surrogate machine learning models [**?**] to identify electrolytes with high ionization potentials (IP) in a candidate set.

The workflow comprises: (1) *simulation* tasks that compute IPs on CPUs, (2) *training* tasks that train surrogate models to predict IPs, and (3) *inference* tasks that use trained surrogate models to predict IPs, which are then ranked by confidence and used to guide future simulation tasks. The simulation tasks run on Theta compute nodes, and the training and inference tasks run on a remote GPU node (located behind a different NAT and using a different authentication procedure than Theta). Tasks are orchestrated with a

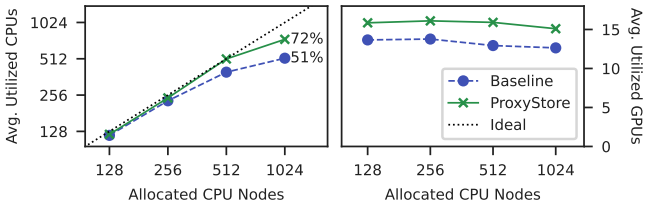

Figure 11: Average node utilization of the molecular design application, with and without ProxyStore. The number of GPUs used for training and inference tasks is constant while the number of CPU nodes for simulations is increased. Without ProxyStore, the application struggles to keep all CPU nodes and GPUs fed with tasks. ProxyStore reduces the amount of data flowing through the workflow system, thus reducing the latency between task results being received and new tasks dispatched.

Colmena Thinker running on a Theta login node and task execution is managed with Parsl.

To optimize communication of task data, we use the `MultiConnector` configured to use `RedisConnector` for simulation tasks and `EndpointConnector` for training and inference tasks. `RedisConnector` is suitable for low-latency communication between Theta login and compute nodes and provides persistence when an application spans multiple batch jobs; PS-endpoints enable peer-to-peer transfer of model weights (10 MB in this case) to and from a remote GPU node. Inputs to inference tasks also require peer-to-peer data transfer to remote GPU nodes. The inference dataset is static, so while the first round of proxies result in data being moved to the GPU node; proxies for later inference rounds benefit from cached data. We also investigated using `GlobusConnector` for data movement. While in this case the dataset was not large enough to benefit from Globus transfers, this would be a good option if a larger dataset were used. This workflow exemplifies how ProxyStore can coordinate optimal communication in complex workflows. We note that no task code needed to be modified to work with the diverse communication methods employed.

In this application, we want to use proxies to reduce overheads in the workflow system. We evaluate their effectiveness for this purpose by measuring average node utilization during application execution as a function of the number of Theta KNL nodes used for simulations. We see in **??** that the workflow system struggles to keep nodes fed with new tasks as scale increases. However, use of ProxyStore removes data movement burdens from the workflow system and improves scaling, improving utilization by 29% and 43% at 512 and 1024 nodes, respectively. We also observe ProxyStore improves utilization of the remote GPUs by speeding up data transfer. At 1024 nodes with ProxyStore, computation, rather than communication, becomes the bottleneck because simulation results must be processed serially prior to dispatching new simulations. Processing a simulation result takes 267 ± 518 ms on average in the baseline 1024 node run, but ProxyStore improves this time by 25% to 201 ± 140 ms.

6 CONCLUSION

ProxyStore is a novel framework for facilitating wide-area data management in distributed applications. The proxy model provides a pass-by-reference-like interface that can work across processes, machines, and sites, and enables data producers to change communication methods dynamically without altering application behavior. ProxyStore provides a suite of communication channel implementations intended to meet most requirements and can be extended to other communication methods. We demonstrated the use of ProxyStore with FaaS and workflow systems, synthetic benchmarks, and real-world scientific applications. We showed that ProxyStore can accelerate a diverse range of distributed applications and enables comparable performance to alternative approaches while avoiding the cumbersome code changes and/or manual deployment and configurations required by alternatives.

In future work, we will investigate support for more communication methods, advanced data management policies for persistence and replication, and wide-area reference counting for object eviction. It may be useful to allow for data flow semantics on proxies, so that readers of an object block until the object is written, as in Id [**?**]. We will investigate areas for optimization such as intelligent prefetching and faster peer-to-peer networking protocols. We plan to explore extension of the proxy model to other languages and other problems in distributed computing (e.g., lazy library loading). We hope thus to encourage further research in data fabrics for federated applications and enable scientists and engineers to more easily design sophisticated distributed applications.

ACKNOWLEDGMENTS

This research was supported in part by the Department of Energy (DOE) under Contract DE-AC02-06CH11357, the ExaWorks project and ExaLearn Co-design Center of the Exascale Computing Project (17-SC-20-SC), and the NSF under Grant 2004894. This research used resources provided by the Argonne Leadership Computing Facility (ALCF), a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357; the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231; the Texas Advanced Computing Center (TACC) at the University of Texas at Austin; the University of Chicago's Research Computing Center; and the Chameleon testbed supported by the National Science Foundation.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283. USENIX Association, 2016.

[2] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *Computer*, 19(08):26–34, 1986.

[3] aiortc. https://github.com/aiortc/aiortc. Accessed Mar 2023.

[4] Mehmet Fatih Aktas, Javier Diaz-Montes, Ivan Rodero, and Manish Parashar. WA-DataSpaces: Exploring the data staging abstractions for wide-area distributed scientific workflows. In *46th International Conference on Parallel Processing*, pages 251–260. IEEE, 2017.

[5] Aymen Al-Saadi, Dong H Ahn, Yadu Babuji, Kyle Chard, James Corbett, Mihael Hategan, Stephen Herbein, Shantenu Jha, Daniel Laney, Andre Merzky, Todd Munson, Michael Salim, Mikhail Titov, Matteo Turilli, and Justin M. Wozniak.

ExaWorks: Workflows for exascale. In *IEEE Workshop on Workflows in Support of Large-Scale Science*, pages 50–57. IEEE, 2021.
[6] Bryce Allen, John Bresnahan, Lisa Childers, Ian Foster, Gopi Kandaswamy, Raj Kettimuthu, Jack Kordas, Mike Link, Stuart Martin, Karl Pickett, and Steven Tuecke. Software as a service for data scientists. *Communications of the ACM*, 55(2):81–88, feb 2012.
[7] Aymen Alsaadi, Logan Ward, Andre Merzky, Kyle Chard, Ian Foster, Shantenu Jha, and Matteo Turilli. Radical-Pilot and Parsl: Executing heterogeneous workflows on HPC platforms. *arXiv preprint arXiv:2105.13185*, 2021.
[8] Amazon Lambda. https://aws.amazon.com/lambda. Accessed Jan 2023.
[9] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Luksaz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. Parsl: Pervasive parallel programming in Python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2019.
[10] Georg Bauer. Linda tuple spaces for Python. https://pypi.org/project/lindypy/. Accessed Mar 2023.
[11] Juan Benet. IPFS - Content addressed, versioned, P2P file system, 2014. https://arxiv.org/abs/1407.3561.
[12] Niklas Blum, Serge Lachapelle, and Harald Alvestrand. WebRTC: Real-time communication for the Open Web Platform. *Communications of the ACM*, 64(8):50–54, jul 2021.
[13] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *ACM SIGOPS 28th Symposium on Operating Systems Principles*, page 836–850. ACM, 2021.
[14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-based congestion control. *ACM Queue*, 14, September-October:20 – 53, 2016.
[15] Kyle Chard, Eli Dart, Ian Foster, David Shifflett, Steven Tuecke, and Jason Williams. The Modern Research Data Portal: A design pattern for networked, data-intensive science. *PeerJ Computer Science*, 4:e144, 2018.
[16] Kyle Chard, Steven Tuecke, and Ian Foster. Efficient and secure transfer, synchronization, and sharing of big data. *IEEE Cloud Computing*, 1(3):46–55, 2014.
[17] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. funcX: A federated function serving fabric for science. In *29th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2020.
[18] Kenneth Chiu, Madhusudhan Govindaraju, and Dennis Gannon. The Proteus multiprotocol message library. In *ACM/IEEE Conference on Supercomputing*, pages 30–30. IEEE, 2002.
[19] Joaquin Chung, Wojciech Zacherek, AJ Wisniewski, Zhengchun Liu, Tekin Bicer, Rajkumar Kettimuthu, and Ian Foster. SciStream: Architecture and toolkit for data streaming between federated science instruments. In *31st International Symposium on High-Performance Parallel and Distributed Computing*, page 185–198. ACM, 2022.
[20] Marcin Copik, Roman Böhringer, Alexandru Calotoiu, and Torsten Hoefler. FMI: Fast and cheap message passing for serverless functions. Technical report, Scalable Parallel Computing Laboratory, ETH Zurich, 2022.
[21] Eli Dart, Lauren Rotman, Brian Tierney, Mary Hester, and Jason Zurawski. The Science DMZ: A network design pattern for data-intensive science. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13. ACM, 2013.
[22] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.
[23] Ewa Deelman, Karan Vahi, Gideon Juve, Mats Rynge, Scott Callaghan, Philip J. Maechling, Rajiv Mayani, Weiwei Chen, Rafael Ferreira da Silva, Miron Livny, and Kent Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.
[24] Ciprian Docan, Manish Parashar, and Scott Klasky. DataSpaces: An interaction and coordination framework for coupled simulation workflows. In *19th ACM International Symposium on High Performance Distributed Computing*, page 25–36. ACM, 2010.
[25] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.
[26] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. http://www.rfc-editor.org/rfc/rfc6455.txt.
[27] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference*, page 13, USA, 2005. USENIX Association. https://arxiv.org/abs/cs/0603074.
[28] Ian Foster. Globus Online: Accelerating and Democratizing Science through Cloud-Based Services. *IEEE Internet Computing*, 15(3):70–73, 2011.
[29] Ian Foster, Jonathan Geisler, Carl Kesselman, and Steven Tuecke. Managing multiple communication methods in high-performance networked computing systems. *Journal of Parallel and Distributed Computing*, 40(1):35–48, 1997.
[30] Google Cloud Functions. https://cloud.google.com/functions/. Accessed Jan 2023.
[31] Mark Handley, Van Jacobson, and Colin Perkins. SDP: Session description protocol. RFC 4566, IETF, 2006. https://www.rfc-editor.org/rfc/rfc4566.html.
[32] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.
[33] Anubhav Jain, Shyue Ping Ong, Wei Chen, Bharat Medasani, Xiaohui Qu, Michael Kocher, Miriam Brafman, Guido Petretto, Gian-Marco Rignanese, Geoffroy Hautier, Daniel Gunter, and Kristin A. Persson. FireWorks: A dynamic workflow system designed for high-throughput applications. *Concurrency and Computation: Practice and Experience*, 27(17):5037–5059, 2015.
[34] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the Chameleon testbed. In *USENIX Annual Technical Conference*. USENIX Association, July 2020.
[35] Ari Keranen, Christer Holmberg, and Jonathan Rosenberg. Interactive connectivity establishment (ICE): A protocol for network address translator (NAT) traversal. RFC 8445, IETF, 2018.
[36] Nikita Kotsehub, Matt Baughman, Ryan Chard, Nathaniel Hudson, Panos Patros, Omer Rana, Ian Foster, and Kyle Chard. Flox: Federated learning with faas at the edge. In *2022 IEEE 18th International Conference on e-Science (e-Science)*, pages 11–20, 2022.
[37] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.
[38] libp2p. https://libp2p.io/. Accessed March 2023.
[39] Steven S Lumetta, Alan M Mainwaring, and David E Culler. Multi-protocol active messages on a cluster of SMP's. In *ACM/IEEE Conference on Supercomputing*, pages 1–22, 1997.
[40] Pavel Mach and Zdenek Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.
[41] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B Letaief. A survey on mobile edge computing: The communication perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.
[42] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, pages 1273–1282. PMLR, 2017.
[43] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, USA, 1994.
[44] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed Jan 2023.
[45] Rishiyur S Nikhil and Keshav K Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, 1989.
[46] ProxyStore GitHub. https://github.com/proxystore. Accessed May 2023.
[47] ProxyStore PyPi. https://pypi.org/project/proxystore/. Accessed May 2023.
[48] Py-Margo. https://github.com/mochi-hpc/py-mochi-margo. Accessed Mar 2023.
[49] Redis, 2023. https://redis.io/. Accessed Mar 2023.
[50] Robert Ross, George Amvrosiadis, Philip Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Greg Ganger, Garth Gibson, Samuel Gutierrez, Robert Latham, Bob Robey, Dana Robinson, Bradley Settlemyer, Galen Shipman, Shane Snyder, Jerome Soumagne, and Zheng Qing. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35(1):121 – 144,, Jan 2020. 10.1007/s11390-020-9802-0.
[51] Real-Time Defect Identification. https://github.com/ivem-argonne/real-time-defect-analysis. Accessed Mar 2023.
[52] Michael Salim, Thomas Uram, J. Taylor Childers, Venkatram Vishwanath, and Michael Papka. Balsam: Near real-time experimental data analysis on supercomputers. In *1st Annual Workshop on Large-scale Experiment-in-the-Loop Computing*. IEEE, November 2019.
[53] Marten Seemann, Max Inden, and Dimitris Vyzovitis. Decentralized hole punching. In *IEEE 42nd International Conference on Distributed Computing Systems Workshops*, pages 96–98. IEEE, 2022.
[54] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018.
[55] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.
[56] Snap Inc. KeyDB: A database built for scale. https://github.com/Snapchat/KeyDB. Accessed Mar 2023.
[57] Jerome Soumagne, Dries Kimpe, Judicael Zounmevo, Mohamad Chaarawi, Quincey Koziol, Ahmad Afsahi, and Robert Ross. Mercury: Enabling remote

procedure call for high-performance computing. In *IEEE International Conference on Cluster Computing*, pages 1–8, 2013.
[58] UCX-Py. https://ucx-py.readthedocs.io/en/latest/. Accessed Mar 2023.
[59] Logan Ward, J. Gregory Pauloski, Valerie Hayot-Sasson, Ryan Chard, Yadu Babuji, Ganesh Sivaraman, Sutanay Choudhury, Kyle Chard, Rajeev Thakur, and Ian 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.
[60] Logan Ward, Ganesh Sivaraman, J. Gregory Pauloski, Yadu Babuji, Ryan Chard, Naveen Dandu, Paul C. Redfern, Rajeev S. Assary, Kyle Chard, Larry A. Curtiss, Rajeev Thakur, and Ian 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.
[61] WebRTC. https://www.w3.org/TR/webrtc/. Accessed Mar 2023.
[62] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.
[63] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms, 2017. https://doi.org/10.48550/arXiv.1708.07747.