Ownership: A Distributed Futures System for Fine-Grained Tasks

Stephanie Wang, Eric Liang, and Edward Oakes, *UC Berkeley and Anyscale;*
Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica, *UC Berkeley*

https://www.usenix.org/conference/nsdi21/presentation/cheng

This paper is included in the
Proceedings of the 18th USENIX Symposium on
Networked Systems Design and Implementation.

April 12–14, 2021

978-1-939133-21-2

Open access to the Proceedings of the
18th USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

Ownership: A Distributed Futures System for Fine-Grained Tasks

Stephanie Wang*, Eric Liang*, Edward Oakes*, Ben Hindman, Frank Luan, Audrey Cheng, Ion Stoica
*UC Berkeley, *and Anyscale*

Abstract

The distributed futures interface is an increasingly popular choice for building distributed applications that manipulate large amounts of data. Distributed futures are an extension of RPC that combines futures and distributed memory: a distributed future is a reference whose eventual value may be stored on a remote node. An application can then express distributed computation without having to specify when or where execution should occur and data should be moved.

Recent distributed futures applications require the ability to execute *fine-grained computations*, i.e., tasks that run on the order of milliseconds. Compared to coarse-grained tasks, fine-grained tasks are difficult to execute with acceptable system overheads. In this paper, we present a distributed futures system for fine-grained tasks that provides fault tolerance without sacrificing performance. Our solution is based on a novel concept called *ownership*, which assigns each object a leader for system operations. We show that this decentralized architecture can achieve horizontal scaling, 1ms latency per task, and fast failure handling.

1 Introduction

RPC is a standard for building distributed applications because of its generality and because its simple semantics yield high-performance implementations. The original proposal uses *synchronous* calls that *copy* return values back to the caller (Figure 2a). Several recent systems [4, 34, 37, 45] have extended RPC so that, in addition to distributed communication, the system may also manage *data movement* and *parallelism* on behalf of the application.

Data movement. Pass-by-value semantics require all RPC arguments to be sent to the executor by copying them directly into the request body. Thus, performance degrades with *large* data. Data copying is both expensive and unnecessary in cases like Figure 2a, where a process executes an RPC over data that it previously returned to the same caller.

To reduce data copies, some RPC systems use *distributed memory* [16, 27, 37, 40, 41]. This allows large arguments to be passed by *reference* (Figure 2b), while small arguments can still be passed by value. In the best case, arguments passed by reference to an RPC do not need to be copied if they are already on the same node as the executor (Figure 2b). Note that, like traditional RPC, we make all values *immutable* to simplify the consistency model and implementation.

Parallelism. RPCs are traditionally *blocking*, so control is only returned to the caller once the reply is received (Figure 2a). *Futures* are a popular method for extending RPC with asynchrony [8, 29], allowing the system to execute functions in parallel with each other and the caller. With *composition* [29, 37], i.e., passing a future as an argument to another RPC, the application can also express the parallelism and dependencies of future RPCs. For example, in Figure 2c, `add` is invoked at the beginning of the program but only executed by the system once `a` and `b` are computed.

```
a_future = compute()
b_future = compute()
c_future = add(a_future, b_future)
c = system.get(c_future)
```

Figure 1: A distributed futures program. `compute` and `add` are stateless. `a_future`, `b_future`, and `c_future` are distributed futures.

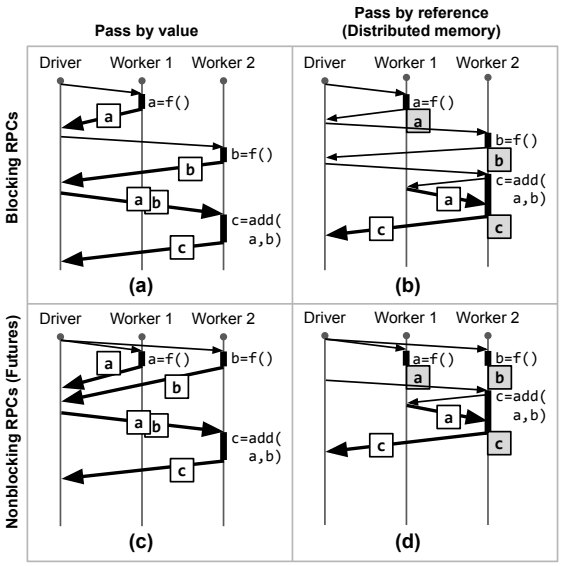


Figure 2: Example executions of the program from Figure 1. **(a)** With RPC. **(b)** With RPC and distributed memory, allowing the system to reduce data copies. **(c)** With RPC and futures, allowing the system to manage parallel execution. **(d)** With distributed futures.

Distributed futures are an extension of RPC that combines futures with distributed memory: a *distributed future* is a reference whose eventual value may be stored on a remote node (Figure 2d). An application can then express distributed computation without having to specify when or where execution should occur and data should be moved. This is an increasingly popular interface for developing distributed applications that manipulate large amounts of data [4, 34, 37, 45].

As with traditional RPC, a key goal is generality. To achieve this, the system must minimize the overhead of each function call [13]. For example, the widely used gRPC provides horizontal scalability and sub-millisecond RPC latency, making

it practical to execute millions of *fine-grained* functions, i.e. millisecond-level "tasks", per second [2].

Similarly, there are emerging examples of large-scale, fine-grained applications of distributed futures, including reinforcement learning [34], video processing [22,43], and model serving [49]. These applications must optimize parallelism and data movement for performance [39,43,49], making distributed futures apt. Unfortunately, existing systems for distributed futures are limited to *coarse-grained* tasks [37].

In this paper, we present a distributed futures system for fine-grained tasks. While others [34,37,45] have implemented distributed futures before, our contribution is in identifying and addressing the challenges of providing *fault tolerance for fine-grained tasks without sacrificing performance*.

The primary challenge is that distributed futures introduce *shared state* between processes. In particular, an object and its metadata are shared by its reference holder(s), the RPC executor that creates the object, and its physical location(s). To ensure that each reference holder can dereference the value, the processes must coordinate, a difficult problem in the presence of failures. In contrast, traditional RPC has no shared state, since data is passed by value, and naturally avoids coordination, which is critical to scalability and low latency.

For example, in Figure 2a, once worker 1 copies `a` to the driver, it does not need to be involved in the execution of the downstream `add` task. In contrast, worker 1 stores `a` in Figure 2d, so the two workers must coordinate to ensure that `a` is available long enough for worker 2 to read. Also, worker 1 must garbage-collect `a` once worker 2 executes `add` and there are no other references. Finally, the processes must coordinate to detect and recover from the failure of another process.

The common solution in previous systems is to use a centralized master to store system state and coordinate these operations [34, 37]. A simple way to ensure fault tolerance is to record and replicate metadata at the master *synchronously* with the associated operation. For example, in Figure 2d, the master would record that `add` is scheduled to worker 2 *before* dispatching the task. Then, it can correctly detect *c*'s failure if worker 2 fails. However, this adds significant overhead for applications with a high volume of fine-grained tasks [].

Thus, decentralizing the system state is necessary for scalability. The question is how to do so without complicating coordination. The key insight in our work is to exploit the application structure: a distributed future may be shared by passing by reference, but *most distributed futures are shared within the scope of the caller*. For example, in Figure 1, `a_future` is created then passed to `add` in the same scope.

We thus propose *ownership*, a method of decentralizing system state across the RPC *executors*. In particular, the caller of a task is the *owner* of the returned future and all related metadata. In Figure 2d, the driver owns `a`, `b`, and `c`.

This solution has three advantages. First, for horizontal scalability, the application can use nested tasks to "shard" system state across the workers. Second, since a future's owner is the task's caller, task latency is low because the required metadata writes, though synchronous, are local. This is in contrast to an application-agnostic method of sharding, such as consistent hashing. Third, each worker becomes in effect a centralized master for the distributed futures that it owns, simplifying failure handling.

The system guarantees that if the owner of a future is alive, any task that holds a reference to that future can eventually dereference the value. This is because the owner will coordinate system operations such as reference counting, for memory safety, and lineage reconstruction, for recovery. Of course, this is not sufficient if the owner fails.

Here, we rely on lineage reconstruction and a second key insight into the application structure: in many cases, the references to a distributed future are held by tasks that are a descendant of the failed owner. The failed task can be recreated through lineage reconstruction by *its* owner, and the descendant tasks will also be recreated in the process. Therefore, it is safe to *fate-share* any tasks that have a reference to a distributed future with the future's owner. As we expect failures to be relatively rare, we argue that this reduction in system overheads and complexity outweighs the cost of additional re-execution upon a failure.

In summary, our contributions are:

- A decentralized system for distributed futures with transparent recovery and automatic memory management.
- A lightweight technique for transparent recovery based on lineage reconstruction and fate sharing.
- An implementation in the Ray system [34] that provides high throughput, low latency, and fast recovery.

2 Distributed Futures

2.1 API

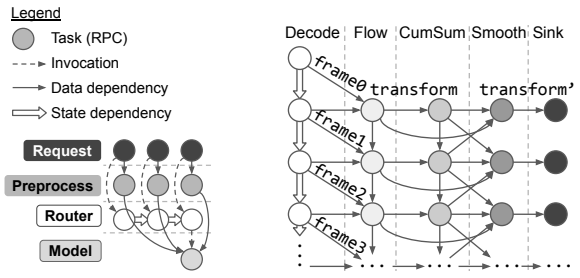
The key benefit of distributed futures is that the system can transparently manage parallelism and data movement on behalf of the application. Here, we describe the API (Table 1).

To spawn a *task*, the caller invokes a *remote function* that immediately returns a `DFut` (Table 1). The spawned task comprises the function and its arguments, resource requirements, etc. The returned `DFut` refers to the *object* whose value will be returned by the function. The caller can *dereference* the `DFut` through `get`, a blocking call that returns a copy of the object. The caller can *delete* the `DFut`, removing it from scope and allowing the system to reclaim the value. Like other systems [34, 37, 45], all objects are *immutable*.

After the creation of a `DFut` through task invocation, the caller can create other references in two ways. First, the caller can pass the `DFut` as an argument to another task. `DFut` task arguments are implicitly dereferenced by the system. Thus, the task will only begin once all upstream tasks have finished, and the executor sees only the `DFut` *values*.

Operation	Semantics
`f(DFut x) → DFut`	Invoke the remote procedure `f`, and pass `x` by reference. The system implicitly dereferences `x` to its `Value` before execution. Creates and returns a distributed future, whose value is returned by `f`.
`get(DFut x) → Value`	Dereference a distributed future. Blocks until the value is computed and local.
`del(DFut x)`	Delete a reference to a distributed future from the caller's scope. Must be called by the program.
`Actor.f(DFut x) → DFut`	Invoke a stateful remote procedure. `f` must execute on the actor referred to by `Actor`.
`shared(DFut x) → SharedDFut`	Returns a `SharedDFut` that can be used to pass `x` to another worker, without dereferencing the value.
`f(SharedDFut x) → DFut`	Passes `x` as a first-class `DFut`: The system dereferences `x` to the corresponding `DFut` instead of the `Value`.

Table 1: Distributed futures API. The full API also includes an actor creation call. A task may also return a `DFut` to its caller (nested `DFuts` are automatically flattened).



(a) Model serving (b) Video processing

Figure 3: Distributed futures applications.

Second, the `DFut` can be passed or returned as a *first-class value* [21], i.e. passed to another task without dereferencing. Table 1 shows how to cast a `DFut` to a `SharedDFut`, so the system can differentiate when to dereference arguments. We call the process that receives the `DFut` a *borrower*, to differentiate it from the original caller. Like the original caller, a borrower may create other references by passing the `DFut` or casting again to a `SharedDFut` (creating further borrowers).

Like recent systems [4, 34, 45], we support *stateful* computation with actors. The caller creates an actor by invoking a *remote constructor function*. This immediately returns a reference to the actor (an `ARef`) and asynchronously executes the constructor on a remote process. The `ARef` can be used to spawn tasks bound to the same process. Similar to `DFuts`, `ARefs` are first-class, i.e. the caller may return or pass the `ARef` to another task, and the system automatically collects the actor process once all `ARefs` have gone out of scope.

2.2 Applications

Typical applications of distributed futures are those for whom performance requires the flexibility of RPC, as well as optimization of data movement and parallelism. We describe some examples here and evaluate them in Section 5.2.

Distributed futures have previously been explored for data-intensive applications that cannot be expressed or executed efficiently as data-parallel programs [34, 37]. Ciel identified the key ability to *dynamically* specify tasks during execution, e.g., based on previous results, rather than specify the entire graph upfront [37]. This enabled new workloads such as dynamic programming, which is recursive by nature [54].

Our goal is to expand the application scope to include those with *fine-grained* tasks that run in the milliseconds. We also explore the use of actors and first-class distributed futures.

Model serving. The goal is to reduce request latency while maximizing throughput, often by using model *replicas*. Depending on the model, a latency target might be 10-100ms [20]. Typically, an application-level scheduling policy is required, e.g., for staged rollout of new models [46].

Figure 3a shows an example of a GPU-based image classification pipeline. Each client passes its input image to a `Preprocess` task, e.g., for resizing, then shares the returned `DFut` with a `Router` actor. `Router` implements the scheduling policy and passes the `DFut` by reference to the chosen `Model` actor. `Router` then returns the results to the clients.

Actors improve performance in two ways: (1) each `Model` keeps weights warm in its local GPU memory, and (2) `Router` buffers the preprocessed `DFuts` until it has a batch of requests to pass to a `Model`, to leverage GPU parallelism for throughput. With *dynamic* tasks, the `Router` can also choose to flush its buffer on a timeout, to reduce latency from batching.

First-class distributed futures are important to reduce routing overhead. They allows the `Router` to pass the references of the preprocessed images to the `Model` actors, instead of copying these images. This avoids creating a bottleneck at the `Router`, which we evaluate in Figure 15a. While the application could use an intermediate storage system for preprocessed images, it would then have to manage additional concerns such as garbage collection and failures.

Online video processing. Video processing algorithms often have complex data dependencies that are not well supported by data-parallel systems such as Apache Spark [22, 43]. For example, video stabilization (Figure 3b) works by tracking objects between frames (Flow), taking a cumulative sum of these trajectories (CumSum), then applying a moving average (Smooth). Frame-to-frame dependencies are common, such as the video decoding state stored in an actor in Figure 3b. Each stage runs in 1-10s of milliseconds per frame.

Safe and timely garbage collection in this setting can be challenging because a single object (e.g., a video frame) may be referenced by multiple tasks. Live video processing is also latency-sensitive: output must be produced at the same frame rate as the input. Low latency relies on pipelined parallelism between frames, as the application cannot afford to wait for multiple input frames to appear before beginning execution.

With distributed futures, the application can specify the logical task graph *dynamically*, as input frames appear. Meanwhile, the system manages the physical execution, i.e. pipelined parallelism and garbage collection, according to the specified graph. Concurrent video streams can easily be

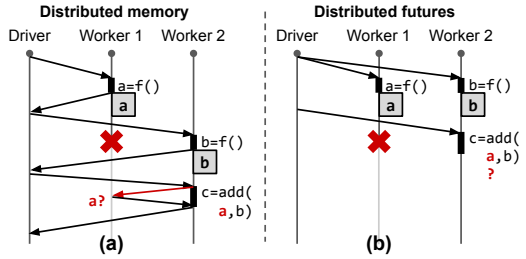


Figure 4: Failure detection. **(a)** `a`'s location is known by the time worker 2 receives the reference. **(b)** `a`'s location may not be known when worker 2 receives `add`, so worker 2 cannot detect the failure.

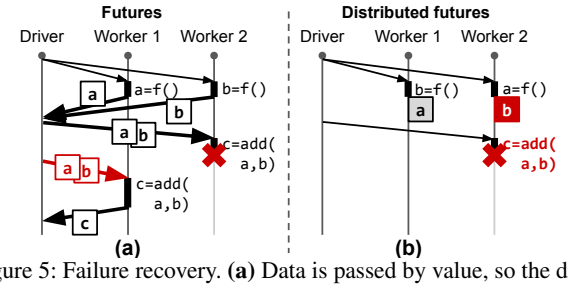


Figure 5: Failure recovery. **(a)** Data is passed by value, so the driver recovers by resubmitting `add`. **(b)** `b` is also lost. `f`'s description must be recorded during runtime so that `b` can be recomputed.

supported using nested tasks, one "driver" per stream. The system can then manage inter-video parallelism.

3 Overview

3.1 Requirements

The system guarantees that each `DFut` can be *dereferenced* to its value. This involves three problems: automatic memory management, failure detection, and failure recovery.

Automatic memory management is a system for dynamic memory allocation and reclamation of objects. The system must decide at run time whether an object is currently referenced by a live process, e.g., through reference counting [42].

Failure detection is the minimum functionality needed to ensure progress in the presence of failures. The system detects when a `DFut` cannot be dereferenced due to worker failure.

With distributed memory but no futures, this is straightforward because *the location of the value is known by the time the reference is created*. In Figure 4a, for example, the driver learns that `a` is stored on worker 1 and could then attach the location when passing `a` to worker 2. Then, when worker 2 receives `add`, it can detect `a`'s failure.

The addition of futures complicates failure detection because references can be created *before* the value. Even the future *location* of the value may not be known at reference creation time. Of course, the system could wait until a task has been scheduled before returning the reference to the caller. However, this would defeat the purpose of futures as an asynchronous construct. It is also impractical because a realistic scheduler must be able to update its decision at run time, e.g., according to changes in the environment such as resource availability and worker failures.

Thus, it is possible that there are no locations for `a` when worker 2 receives the `add` RPC in Figure 4b. Then, worker 2 must decide whether `f` is still executing, or if it has failed. If it is the former, then worker 2 should wait. But if there is a failure, then the system must recover `a`. To solve this problem, the system must record the locations of all *tasks*, i.e. pending objects, in addition to created objects.

Failure recovery. The system must also provide a method of recovering from a failed `DFut`. The minimum requirement is to throw an error to the application if it tries to dereference a failed `DFut`. We further provide an option for *transparent* recovery, i.e. the system will recover a failed `DFut`'s value.

With futures but no distributed memory, if a process fails, then we will lose the reply of any pending task on that process. Assuming idempotence, this can be recovered through retries, a common approach for pass-by-value RPC. For example, in Figure 5a, the driver recovers by resubmitting `add(a,b)`. Failure recovery is simple because *all data is passed by value*.

With distributed memory, however, tasks can also contain arguments passed by *reference*. Therefore, a node failure can cause the loss of an object value that is still referenced, as `b` is in Figure 4b. A common approach to this problem is to record each object's *lineage*, or the subgraph that produced the object, during runtime [17, 30, 56]. The system then walks a lost object's lineage and recursively reconstructs the object and its dependencies through task re-execution. This approach reduces the runtime overhead of logging, since the data itself is not recorded, and the work that must be redone after a partial failure, since objects cached in distributed memory do not need to be recomputed. Still, achieving low run-time overhead is difficult because the lineage itself must be recorded and collected at run time and it must survive failures.

Note that we focus specifically on *object recovery* and, like previous systems [34, 37, 56], assume *idempotence* for correctness. Thus, our techniques are directly applicable to idempotent functions and actors with read-only, checkpointable, or transient state, as we evaluate in Figure 15c. Although it is not our focus, these techniques may also be used in conjunction with known recovery techniques for actor state [17, 34] such as recovery for nondeterministic execution [52].

Metadata requirements. In summary, during normal operation, the system must at minimum record (1) the location(s) of each object's value, so that reference holders can retrieve it, and (2) whether the object is still referenced, for safe garbage collection. For failure detection and recovery, the system must further record, respectively, (3) the location of each *pending* object, i.e. the task location, and (4) the object lineage.

The key question is where and when to record this system metadata such that it is *consistent*[1] and *fault-tolerant*. By consistent, we mean that the system metadata matches the

[1] Unrelated to the more standard definition of replica consistency [50].

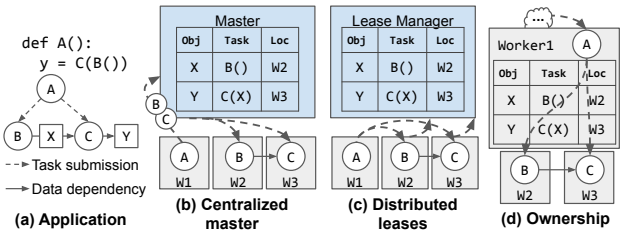


Figure 6: Distributed futures systems. **(a)** An application. **(b)** Master manages metadata and object failures. **(c)** Workers write metadata asynchronously, coordinate failure handling with leases. **(d)** Workers manage metadata. Worker 1 handles failures for workers 2 and 3. Worker 1 failure is handled by `A`'s owner elsewhere in the cluster.

current physical state of the cluster. By fault-tolerant, we mean that the metadata should survive individual node failures.

In some cases, it is safe for metadata to be *asynchronously updated*, i.e. there is a transient mismatch between the system metadata and the system state. For example, the system may transiently believe that an object `x` is still on node *A* even though it has been removed. This is safe because a reference holder can resolve the inconsistency by asking *A* if it has `x`.

On the other hand, metadata needed for failure handling should ideally be *synchronously updated*. For example, the metadata should never say that a task *T* is on node *A* when it is really on node *B*. In particular, if node *A* then fails, the system would incorrectly conclude that *T* has failed. As we will see next, synchrony simplifies fault tolerance but can add significant runtime overhead if done naively.

3.2 Existing solutions

Centralized master. Failure handling is simple with a synchronously updated centralized master, but this design can also add significant runtime overhead. For example, failure detection requires that the master record a task's scheduled location *before* dispatch (Figure 6b). Similarly, the master must record every new reference before it can be used. This makes the master a bottleneck for scalability and latency.

The master can be sharded for scalability, but this can complicate operations that coordinate multiple objects, such as garbage collection and lineage reconstruction. Also, the latency overhead is fundamental. Each task invocation must first contact the master, adding at minimum one round-trip to the critical path of execution, even without replicating the metadata for fault tolerance. This overhead can be detrimental when the task itself is milliseconds long, and especially so if the return value is small enough to be passed by value. Small values may be stored in the master directly as an optimization, but still require 1 RTT for retrieval [38].

Distributed leases. Decentralization can remove such bottlenecks, but often leads to complex coordination schemes. One approach is to use *distributed leases* [19]. This is similar to a centralized master that is updated *asynchronously*.

As an example, consider asynchronous task location updates (Figure 6c). To account for a possibly stale master, the worker nodes must coordinate to detect task failures, in this case using leases. Each worker node acquires a lease for each locally queued task and repeatedly renews the lease until the task has finished. For example, in Figure 6c, worker 3 can detect a failure of `B` by waiting for worker 2's lease to expire.

This design is horizontally scalable through sharding and reduces task latency, since metadata is written asynchronously. However, the reliance on timing to reconcile system state can slow recovery (Figure 14). Furthermore, this method of decentralization introduces a new problem: the workers must also coordinate on *who* should recover an object, i.e. re-execute the creating task. This is trivial in the centralized scheme, since the master coordinates all recovery operations.

3.3 Our solution: Ownership

The key insight in our work is to "shard" the centralized master, for scalability, but to do so based on the application structure, for low run-time overhead and simple failure handling. In ownership, the worker that calls a task stores the metadata related to the returned `DFut`. Like a centralized master, it coordinates operations such as task scheduling, to ensure it knows the task location, and garbage collection. For example, in Figure 6d, worker 1 owns `X` and `Y`.

The reason for choosing the task's caller as the owner is that in general, it is the worker that accesses the metadata most frequently. The caller is involved in the initial creation of the `DFut`, via task invocation, as well as the creation of other references, by passing the `DFut` to other RPCs. Thus, task invocation latency is minimal because the scheduled location is written locally. Similarly, if the `DFut` stays in the owner's scope, the overhead of garbage collection is low because the `DFut`'s reference count can be updated locally when the owner passes the `DFut` to another RPC. These overheads can be further reduced for small objects, which can be passed by value as if without distributed memory (see Section 4.2).

Of course, if all tasks are submitted by a single driver, as in BSP programs, ownership will not scale beyond the driver's throughput. Nor indeed will any system for dynamic tasks. However, with ownership, the *application* can scale horizontally by distributing its control logic across multiple nested tasks, as opposed to an application-agnostic method such as consistent hashing (Figure 12e). Furthermore, the worker processes hold much of the system metadata. This is in contrast to previous solutions that push all metadata into the system's centralized or per-node processes, limiting the *vertical* scalability of a single node with many worker processes (Figure 12).

However, there are problems that are simpler to solve with a fully centralized design, assuming sufficient performance:

First-class futures. First-class futures (Section 2) allow non-owning processes to reference a `DFut`. While many applications can be written without first-class futures (Figure 3b),

they are sometimes essential for performance. For example, the model serving application in Figure 3a uses first-class futures to delegate task invocation to a nested task, without having to dereference and copy the arguments.

A first-class `DFut` may leave the owner's scope, so we must account for this during garbage collection. We avoid centralizing the reference count at the owner, as this would defeat the purpose of delegation. Instead, we use a distributed hierarchical reference counting protocol (Section 4.2). Each borrower stores a local reference count for the `DFut` on behalf of the owner (Table 2) and notifies the owner when the local reference count reaches zero. The owner decides when the object is safe to reclaim. We use a reference counting approach as opposed to tracing [42] to avoid global pauses.

Owner recovery. If a worker fails, then we will also lose its owned metadata. For transparent recovery, the system must recover the worker's state on a new process and reassociate state related to the previously owned `DFuts`, including any copies of the value, reference holders, and pending tasks.

We choose a minimal approach that guarantees progress, at the potential cost of additional re-execution on a failure: we *fate share* the object and any reference holders with the owner, then use *lineage reconstruction* to recover the object and any of the owner's fate-shared children tasks (Section 4.3). This method adds minimal run-time overhead and is correct, i.e. the application will recover to a previous state and the system guarantees against resource leakage. A future extension is to persist the owner's state to minimize recovery time at the cost of additional recovery complexity and run-time overhead.

4 Ownership Design

Each node in the cluster hosts one to many workers (usually one per core), one scheduler, and one object store (Figure 7). These processes implement future resolution, resource management, and distributed memory, respectively. Each node and worker process is assigned a unique ID.

Workers are responsible for the resolution, reference counting, and failure handling of distributed futures. Each worker executes one task at a time and can invoke other tasks. The root task is executed by the "driver".

Each task has a unique `TaskID` that is a hash of the parent task's ID and the number of tasks invoked by the parent task so far. The root `TaskID` is assigned randomly. Each task may return multiple objects, each of which is assigned an `ObjectID` that concatenates the `TaskID` and the object's index. A `DFut` is a tuple of the `ObjectID` and the owner's address (`Owner`).

The worker stores one record per future that it has in scope in its local *ownership table* (Table 2). A `DFut` borrower records a subset of these fields (* in Table 2). When a `DFut` is passed as an argument to a task, the system implicitly resolves the future's value, and the executing worker stores only the `ID`, `Owner`, and `Value` for the task duration. The worker also caches the owner's stored `Locations`.

Field	Value
`*ID`	The `ObjectID`. Also used as a distributed memory key.
`*Owner`	Address of the owner (IP address, port, `WorkerID`).
`*Value`	(1) Empty if not yet computed, (2) Pointer if in distributed memory, or (3) Inlined value, for small objects (Section 4.2).
`*References`	A list of reference holders: Number of dependent tasks and a list of borrower addresses (Section 4.2 and appendix A).
`Task`	Specification for the creating task. Includes the `ObjectIDs` and `Owners` of any `DFuts` passed as arguments.
`Locations`	If `Value` is empty, the location of the task. If `Value` is a pointer to distributed memory, then the locations of the object.

Table 2: Ownership table. The owner stores all fields. A borrower (Section 3.2) only stores fields indicated by the *.

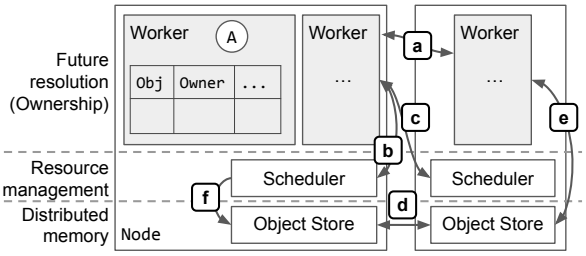


Figure 7: Architecture and protocol overview. **(a)** Task execution. **(b)** Local task scheduling. **(c)** Remote task scheduling. **(d)** Object transfer. **(e)** Task output storage and input retrieval. Ownership layer manages distributed memory garbage collection and recovery. **(f)** Scheduler fetches objects in distributed memory to fulfill task dependencies.

An actor is a stateful task that can be invoked multiple times. Like objects, an actor is created through task invocation and *owned* by the caller. The ownership table is also used to locate and manage actors: the `Location` is the actor's address. Like a `DFut`, an `ARef` (an actor reference) is a tuple of the `ID` and `Owner` and can be passed as a first-class value to other tasks.

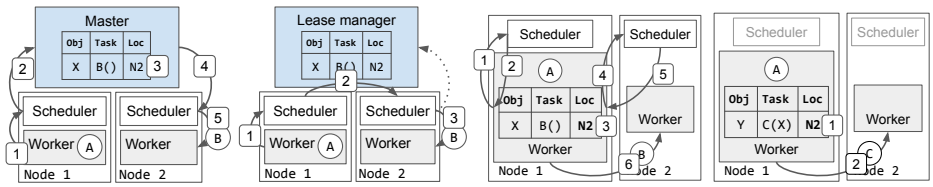
A worker requests resources from the scheduling layer to determine task placement (Section 4.1). We assume a decentralized scheduler for scalability: each scheduler manages local resources, can serve requests from remote workers, and can redirect a worker to a remote scheduler.

The distributed memory layer (Section 4.2) consists of an immutable distributed object store (Figure 7d) with `Locations` stored at the owner. The `Locations` are updated asynchronously. The object store uses shared memory to reduce copies between reference holders on the same node.

Workers store, retrieve, reclaim, and recover large objects in distributed memory (Figure 7f). The scheduling layer sends requests to distributed memory to fetch objects between nodes according to worker requests (Figure 7g).

4.1 Task scheduling

We describe how the owner coordinates task scheduling. At a high level, the owner dispatches each task to a location chosen by the distributed scheduler. This ensures that the task location in the ownership table is updated synchronously with dispatch. We assume an abstract scheduling policy that takes



(a) (b) (c) (d)

	Node	L. RTTs	R. RTTs
Master	L.	1	1
	R.	1	1
Leases	L.	1	0
	R.	1	0.5
Owner-ship	L.	0.5 or 1.5	0
	R.	0 or 1	0.5 or 1.5

(e)

Figure 8: Task scheduling and the method of recording a task's location for the program in Figure 6a. **(a)** Centralized master. **(b)** Distributed leases. **(c)** Scheduling with ownership. (1-2) Local scheduler redirects owner to node 2. (3) Update task location. (4-5) Remote scheduler grants worker lease. (6) Task dispatch. **(d)** Direct scheduling by the owner, using the worker and resources leased from node 2 in (c). **(e)** Length of critical path of local (L) and remote (R) task execution, in terms of local (L) and remote (R) RTTs.

in resource requests and returns the ID of a node where the resources should be allocated. The policy may also update its decision, e.g., due to changes in resource availability.

Figure 8c shows the protocol to dispatch a task. Upon task invocation, the caller, i.e. the owner of the returned `DFut`, first requests resources from its local scheduler[2]. The request is a tuple of the task's required resources (e.g., `{"CPU": 1}`) and arguments in distributed memory. If the policy chooses the local node, the scheduler accepts the request: it fetches the arguments, allocates the resources, then *leases* a local worker to the owner. Else, the scheduler rejects the request and redirects the owner to the node chosen by the policy.

In both cases, the scheduler responds to the owner with the new location: either the ID of the leased worker or the ID of another node. The owner stores this new location in its local ownership table before dispatching the task to that location. If the request was granted, the owner sends the task directly to the leased worker for execution; otherwise, it repeats the protocol at the next scheduler.

Thus, the owner always dispatches the task to its next location, ensuring that the task's pending `Location` (Table 2) is synchronously updated. This also allows the owner to *bypass* the scheduler by dispatching a task directly to an already leased worker, if the task's resource requirements are met. For example, in Figure 8d, worker 1 reuses the resources leased from node 2 in Figure 8c to execute `C`. The owner returns the lease after a configurable expiration time, or when it has no more tasks to dispatch. We currently do not reuse resources for tasks with different distributed memory dependencies, since these are fetched by the scheduler. We leave other policies for lease revocation and worker reuse for future work.

The *worst-case* number of RTTs before a task executes is higher than in previous solutions because each policy decision is returned to the owner (Figure 8e). However, the throughput of previous solutions is limited (Figure 12) because they cannot support direct worker-to-worker scheduling (Figure 8d). This is because workers do not store system state, and thus all tasks must be routed through the master or per-node scheduler to update the task location (Figures 8a and 8b).

Actor scheduling. The system schedules actor constructor tasks much like normal tasks. After completion, however, the owner holds the worker's lease until the actor is no longer referenced (Section 4.2) and the worker can only execute actor tasks submitted through a corresponding `ARef`.

A caller requests the actor's location from the owner using the `ARef`'s `Owner` field. The location can be cached and requested again if the actor restarts (Section 4.3). The caller can then dispatch tasks directly to the actor, as in Figure 8d, since the resources are leased for the actor's lifetime. For a given caller, the actor executes tasks in the order submitted.

4.2 Memory management

Allocation. The distributed memory layer consists of a set of object store nodes, with locations stored at the owner (Figures 9b to 9d). It exposes a key-value interface (Figure 9a). The object store may replicate objects for efficiency but is not required to handle recovery: if there are no copies of an object, a `Get` call will block until a client (i.e. a worker) `Creates` the object.

Small objects may be faster to copy than to pass through distributed memory, which requires updating the object directory, fetching the object from a remote node, etc. Thus, at object creation time, the system transparently chooses based on size whether to pass by value or by reference.

Objects over a configurable threshold are stored in the distributed object store (step 1, Figure 9b) and returned by reference to the owner (step 2). This reduces the total number of copies, at the cost of requiring at least one IPC to the distributed object store for `Get` (steps 4-5, Figure 9c). Small objects are returned by value to the owner (step 6, Figure 9c), and each reference holder is given its own copy. This produces more copies in return for faster dereferencing.

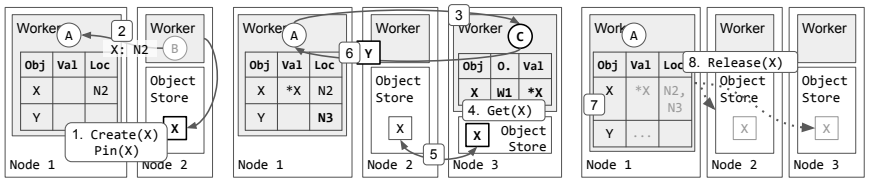
The initial copy of a large object is known as the *primary*. This copy is pinned (step 1, Figure 9b) until the owner releases the object (step 8, Figure 9d) or fails. This allows the object store to treat additional capacity as an LRU cache without having to consult the owners about which objects are safe to evict. For example, the *secondary* copy of `X` created on node 3 in Figure 9c is cached to reduce `Get` and recovery time (Section 4.3) but can be evicted under memory pressure.

Dereferencing. The system dereferences a task's `DFut` arguments before execution. The task's caller first waits for the `Value` field in its local ownership table to be populated (Fig-

[2]The owner can also choose a remote scheduler, e.g., for data locality.

Operation	Semantics
`Create(ObjID o, Value v)`	Store an object.
`Pin(ObjID o, NodeID loc)→ bool`	Pin `o` on `loc` until released. Returns false if `loc` failed.
`Release(ObjID o)`	Object `o` is safe to evict.
`Get(ObjID o)→ Value`	Get the object value. May fetch copy from remote node.

(a)



(b) (c) (d)

Figure 9: **(a)** Distributed memory store API, and **(b-d)** Memory management for the program in Figure 6a. (1-2) `B` returns a large object `X` in distributed memory. The primary copy is pinned until all references have been deleted. (3) Worker 1 dispatches `C` once `X` is available. (4-5) `Get` the value from distributed memory (location lookup not shown). (6) `C` returns a small object `Y` directly to the owner. (7-8) Object reclamation.

ure 9b), then copies the `Value` into the dispatched task description. The executing worker then copies the received `Value` into its local table (Figure 9c). For large objects, the sent value is a pointer to distributed memory, so the worker must also call `Get` to retrieve the actual value (step 4, Figure 9c).

If the task's caller is also the owner of its `DFut` arguments, the above protocol is sufficient. If the task's caller is *borrowing* an argument, then it must populate the `Value` field through a protocol with the owner. Upon receiving a `DFut`, the borrower sends the associated `Owner` a request for the `Value`. The owner replies with the `Value` (either the inlined value or a pointer) once populated. The borrower populates its local `Value` field by copying the reply.

Reclamation. The owner reclaims the object memory once there are no more reference holders (Figure 9d) by deleting its local `Value` field (step 7) and, if necessary, calling `Release` on the distributed object store (step 8). An object's reference holders are tracked with a distributed reference count maintained by the owner and borrowers.

Each process with a `DFut` instance keeps a local count of submitted tasks (`References`, Table 2). The task count is incremented each time the process invokes a dependent task and decremented when the task completes. Each process also keeps a local set of the worker IDs of any borrowers that it created, by passing the `DFut` as a first-class value. This forms a tree of borrowers with the owner at the root (see Appendix A). The owner releases the object once there are no more submitted tasks or borrowers anywhere in the cluster.

Actors. Actors are reference-counted with the same protocol used to track borrowers of a `DFut`. Once the set of reference holders is empty, the owner of the actor reclaims the actor resources by returning the worker lease (Section 4.1).

4.3 Failure recovery

The system guarantees that any reference holder will eventually be able to resolve the value in the presence of failures.

Failure detection. Failure notifications containing a worker or node ID are published to all workers. Workers do not exchange heartbeats; a worker failure is published by its local scheduler. Node failure is detected by exchanging heartbeats between nodes, and all workers fate-share with their node.

Upon receiving a node or worker failure notification, each worker scans its local ownership table to detect a `DFut` failure. A `DFut` is considered failed in two cases: 1) loss of an owned object (Figure 10a), by comparing the `Location` field, or 2) loss of an owner (Figure 11a), by comparing the `Owner` field. We discuss the handling for these two cases next, using lineage reconstruction and fate sharing, respectively.

Note that a non-owner does not need to detect the loss of an object. For example, in Figure 10a, node 2 fails just as worker 3 receives `C`. When worker 3 looks up `X` at the owner, it may not find any locations. From worker 3's perspective, this means that either node 2's write to the directory was delayed, or node 2 failed. Worker 3 does not need to decide which it is; it simply waits for `X`'s owner to handle the failure.

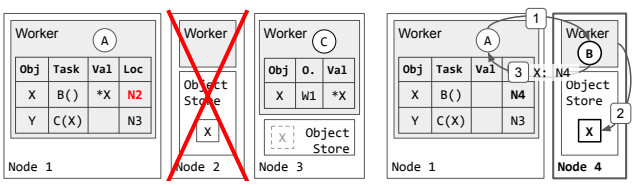
Object recovery. The owner recovers a lost value through lineage reconstruction. During execution, the owner records the object's lineage by storing each invoked `Task` in its ownership table (Table 2). Then, upon detecting a `DFut` failure, the owner resubmits the corresponding task (Figure 10b). The task's arguments are recursively reconstructed, if needed.

Like previous systems [34, 37, 56], we can avoid lineage reconstruction if other copies of a required object still exist. Thus, when reconstructing an object, the owner will first try to locate and designate a secondary copy as the new primary. To increase the odds of finding a secondary copy, object reclamation (Section 4.2) is done lazily: the owner releases the primary copy once there are no more reference holders, but the copy is not evicted until there is memory pressure.

Often, the owner of an object will also own the objects in its lineage (Section 5.2). Thus, upon failure, the owner can locally determine the set of tasks to resubmit, with a recursive lookup of the `Task` fields. In some cases, an object's lineage may also contain *borrowed* references. Then, the borrower requests reconstruction from the owner.

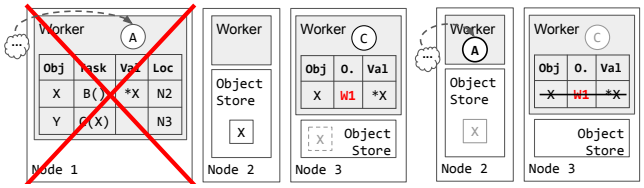
The owner can delete the `Task` field once the task has finished and all objects returned by reference will never be reconstructed again. When a worker returns an object by value, the owner can immediately delete the corresponding `Task` field. This is safe because objects passed by value do not require reconstruction (Section 3.1).

For an object passed by reference, the owner keeps a *lineage reference count* to determine when to collect the `Task`. The



(a) Failure detection. (b) Lineage reconstruction.

Figure 10: Object recovery.



(a) Failure detection. (b) Fate sharing.

Figure 11: Owner recovery.

count is incremented each time the `DFut` is passed to another task and decremented when that `Task` is itself collected. The owner collects a record after collecting both the `Task` and `Value` (Section 4.2) fields. We also plan to support object checkpointing to allow the lineage to be collected early.

Owner recovery. An owner failure can result in a "dangling pointer": a `DFut` that cannot be dereferenced. This can happen if the object is simultaneously lost from distributed memory. For example, `C` in Figure 11a will hang if node 2 also fails.

We use *fate sharing* to ensure that the system can make progress upon an owner's failure. First, all resources held by the owner and any reference holders are reclaimed. Specifically, upon notification of the owner's failure, either the distributed object store frees the object (if it exists) or the scheduling layer reclaims the worker lease (if the object is pending), shown in Figure 11b. All reference holders, i.e. borrowers and dependent tasks, also fate-share with the owner.

Then, to recover the fate-shared state, we rely on *lineage reconstruction*. In particular, the task or actor that was executing on the failed owner must itself have been owned by another process. That process will eventually resubmit the failed task. As the new owner re-executes, it will recreate its previous state, with no system intervention needed. For example, the owner of `A` in Figure 11a will eventually resubmit `A` (Figure 11b), which will again submit `B` and `C`.

For correctness, we show that all previous reference holders are recreated, with the address of the new owner. Consider task T that computes the value of a `DFut` x. T initially executes on worker W and re-executes on W' during recovery. The API (Section 2) gives three ways to create another reference to x: (1) pass x as a task argument, (2) cast x to a `SharedDFut` then pass as a task argument, and (3) return x from T.

In the two former cases, the new reference holder must be a child task of T. In case (2), when x is passed as a first-class value, the child task can create additional reference holders by passing x again. All such reference holders are therefore descendants of T. Then, when T re-executes on W', W' will recreate T's descendants.

T can also return x, which can be useful for returning a child task's result without dereferencing with `get`. Suppose T returns x to its parent task P. Then, P's worker becomes a borrower and will fate-share with W. In this case, P is recovered by *its* owner, and again submits T and receives x.

Thus, because any borrower of `x` must be a child or ancestor of T, fate-sharing and re-execution guarantees that the borrower will be recreated with W' as the new owner. Note that for actors, this requires that an actor not store borrowed `DFuts` in its local state. Of course, this is only required for transparent recovery; the application may also choose to handle failures manually and rely on the system for failure detection only.

While fate-sharing and lineage reconstruction add minimal run-time overhead, it is not suitable for all applications. In particular, the application will fate-share with the driver. In fact, this is the same failure model offered by some BSP systems [3], which can be written as a distributed futures program in which the driver submits all tasks. As shown by these systems, this approach can be extended to reduce the re-execution needed during recovery. We leave such extensions, including application-level checkpointing (Section 5.2), and persistence of the ownership table, for future work.

Actor recovery. Actor recovery is handled through the same protocols. If an actor fails, its owner restarts the actor through lineage reconstruction, i.e. resubmitting the constructor task. If the owner fails, the actor and any `ARef` holders fate-share.

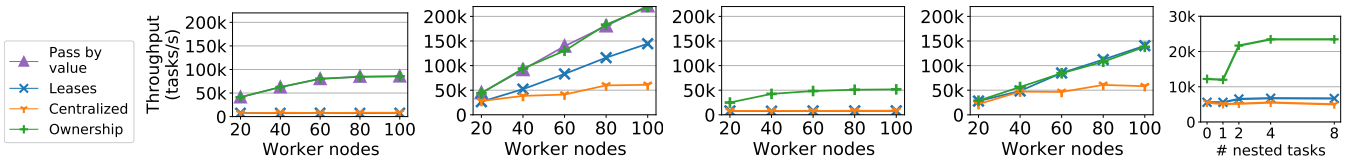
Unlike functions, actors have local state that may require recovery. This is out of scope for this work, but is an interesting future direction. Ownership provides the infrastructure to manage and restart actors, while other methods can be layered on top for transparent recovery of local state [17, 34, 52].

5 Evaluation

We study the following questions:

1. Under what scenarios is distributed futures beneficial compared to pass-by-value RPC?
2. How does the ownership architecture compare against existing solutions for distributed futures, in terms of throughput, latency, and recovery time?
3. What benefits does ownership provide for applications with dynamic, fine-grained parallelism?

We compare against three baselines: (1) a pass-by-value model with futures but no distributed memory, similar to Figure 2c, (2) a decentralized lease-based system for distributed futures (Ray v0.7), and (3) a centralized master for distributed futures (Ray v0.7 modified to write to a centralized master before task execution). All distributed futures systems use sharded, unreplicated Redis for the global metadata store, with asynchronous requests. All systems use the Ray distributed scheduler and (where applicable) distributed object store. Ownership and pass-by-value use gRPC [2] for worker-



(a) Small objects, colocated. (b) Small objects, spread. (c) Large objects, colocated. (d) Large objects, spread. (e) Single node, nested tasks.

Figure 12: Throughput and scalability. **(a-d)** Task submission is divided across multiple intermediate drivers, either colocated on the m5.8xlarge head node or spread with one m5.8xlarge node per driver. 1 intermediate driver is added per 5 worker nodes. Each task returns either a small (short binary string) or large (1MB blob) object. **(e)** Scaling task submission using nested tasks and first-class distributed futures.

to-worker communication. All benchmarks schedule tasks to predetermined nodes to reduce scheduling variation.

All experiments are run on AWS EC2. Global system metadata, such as an object directory, is hosted on the same node as the driver, where applicable. Unless stated otherwise, this "head node" is an m5.16xlarge instance. Other node configuration is listed inline. All benchmark code is available at [53].

5.1 Microbenchmarks

Throughput and scalability. The driver submits one nested task for every 5 worker nodes (m5.8xlarge). Each intermediate "driver" submits no-op tasks to its 5 worker nodes. We report the total throughput of the leaf tasks, which return either a short string (Figures 12a and 12b) or a 1MB blob (Figures 12c and 12d). The drivers are either colocated (Figures 12a and 12c) on the same m5.8xlarge node as the root driver, or spread (Figures 12b and 12d), each on its own m5.8xlarge node. We could not produce stable results for pass-by-value with large objects due to the lack of backpressure in our implementation.

At <60 nodes, the centralized and lease-based architectures achieve about the same throughput because the centralized master is not yet a bottleneck. In general, ownership achieves better throughput than either because it distributes some system operations to the workers. In contrast, the baselines handle all system operations in the global or per-node processes.

The gap between ownership and the baselines is more significant with small return values (Figures 12a and 12b). For these, ownership matches pass-by-value because small objects are returned directly to their owner. The baseline systems could implement a similar optimization, e.g., by inlining small objects in the object directory (Section 4.2), but this would still require at minimum one RPC per read.

When the drivers are spread (Figures 12b and 12d), ownership and leases both scale linearly. Ownership scales better than leases in Figure 12b because more work is offloaded onto the worker processes. Ownership and leases achieve similar throughput in Figure 12d, but the ownership system also includes memory safety (Section 4.2). The centralized design (2 shards) scales linearly to ∼60 nodes. Adding more shards would raise this threshold, but only by a constant amount.

When the drivers are colocated (Figures 12a and 12c), both baselines flatline because of a centralized bottleneck: the scheduler on the drivers' node. Ownership also shows this, but there is less scheduler load overall because the drivers reuse resources for multiple tasks (Section 4.1). A comparable optimization for the baselines would require each driver to batch task submission, at the cost of latency. Throughput for ownership is lower in Figure 12c than in Figure 12a due to the overhead of garbage collection.

Thus, because ownership decentralizes system state among the workers, it can achieve vertical (Figures 12a and 12c) and horizontal (Figures 12b and 12d) scalability. Also, it matches the performance of pass-by-value RPC while enabling new workloads through distributed memory (Section 2.2).

Scaling through borrowing. We show how first-class futures enable delegation. Figure 12e shows the task throughput for an application that submits 100K no-op tasks that each depend on the same 1MB object created by the driver. The tasks are submitted either by the driver (x=0) or by a number of nested tasks that each *borrow* a reference to the driver's object. All workers are colocated on an m5.16xlarge node.

For all systems, the throughput with a single borrower (x=1) is about the same as when the driver submits all tasks directly (x=0). Distributing task submission across multiple borrowers results in a 2× improvement for ownership and negligible improvement for the baselines. Thus, with ownership, an application can scale past the task dispatch throughput of a single worker by *delegating* to nested tasks. This is due to (1) support for first-class distributed futures, and (2) the hierarchical distributed reference counting protocol, which distributes an object's reference count among its borrowers instead of centralizing it at the owner (Section 4.2). In contrast, the baselines would require additional nodes to scale.

Latency. Figure 13 measures task latency with a single worker, hosted either on the same node as the driver ("local"), or on a separate m5.16xlarge node ("remote"). The driver submits 3k tasks that each take the same 1MB object as an argument and that immediately returns a short string. We report the average duration before each task starts execution.

First, distributed memory achieves better latency than pass-by-value in all cases because these systems avoid unnecessary copies of the task argument from the driver to the worker.

Second, compared to centralized and leases, ownership achieves on average 1.6× lower latency. This is due to (1)

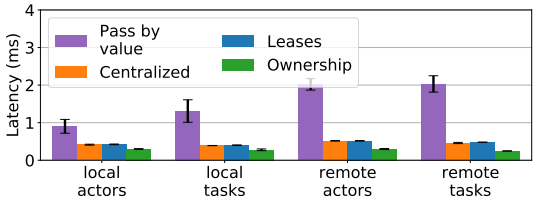
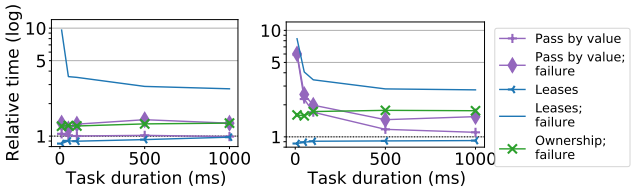


Figure 13: Task latency. Local means that the worker and driver are on the same node. Error bars for standard deviation (across 3k tasks).



(a) Small objects. (b) Large objects.

Figure 14: Total run time (log-scale), relative to ownership without failures. The application is a chain of dependent tasks that execute on one node. Each task sleeps for the duration on the x-axis (total 10s) and returns either **(a)** a short binary string, or **(b)** a 10MB blob.

the ability to write metadata locally at the owner instead of a remote process, and (2) the ability to reuse leased resources, in many cases bypassing the scheduling layer (Section 4.1).

Recovery. This benchmark submits a chain of tasks that execute on a remote m5.xlarge node. Each task depends on the previous, sleeps for the time on the x-axis (total duration 10s), and returns either a short binary string (Figure 14a) or a 10MB blob (Figure 14b). We report the run time relative to ownership without failures. To test recovery, the worker node is killed and restarted 5s into the job (1s heartbeat timeout). We do not include centralized due to implementation effort.

Normal run time for leases is up to 1.18× faster than ownership, but recovery time is more than double, worse than restarting the application. This is because a task's lease must expire before it can be re-executed, adding delay for short tasks. The recovery delay for longer tasks is also high because the implementation (Ray v0.7) repeatedly doubles a lease's expiration time to reduce renewal overhead. A shorter lease interval would reduce recovery delay but can be unstable.

Ownership recovers within 2× the normal run time. Recovery time is the same as pass-by-value for small objects because only in-flight tasks are re-executed (Figure 14a). For large objects (Figure 14b), ownership achieves better normal run time than pass-by-value because arguments are passed by reference; the gap decreases as task execution dominates.

Thus, ownership can achieve the same or better normal run-time performance as leases *and* pass-by-value, while also guaranteeing timely recovery through lineage reconstruction.

5.2 End-to-end applications

Model serving. We implement Figure 3a. Figure 15a shows the latency on 4 p3.16xlarge nodes, each with 1 `Router` and 8 ResNet-50 [23] `Models`. We use a GPU batch size of 16 and generate 2300 requests/s. Ownership and centralized achieve the same median latency (54ms), but the tail latency for centralized is 9× higher (1s vs. 108ms). We also show the utility of first-class distributed futures: in "-borrow", the `Router` receives the image *values* and must copy these to the `Model`. As expected, the `Router` is a bottleneck (p50=80ms, p100=3.2s).

Online video processing. We implement Figure 3b with 60 concurrent videos. The tasks for each stream are executed on an m5.xlarge "worker" node (1 per stream) and submitted by a driver task on a separate m5.xlarge "owner" node. Each owner node hosts 4 drivers. Each video source uses an actor to hold frame-to-frame decoder state. However, tasks are idempotent: a previous frame may be reread with some latency penalty. We use a YouTube video with a frame rate of 29 frames/s and a radius of 1s for the moving average.

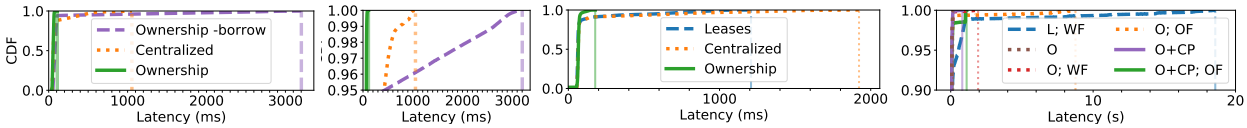
Figure 15b shows latency without failures. All systems achieve similar median latency (∼65ms), but leases and centralized have a long tail (1208ms and 1923ms, respectively). Figure 15c shows latency during an injected failure, 5s after the start, of the `Decoder` actor (Figure 3b). Lease-based recovery is slow because the decoder actor must replay all tasks, and each task accumulates overhead from lease expiration. Checkpointing the actor was infeasible because the leases implementation does not safely garbage-collect lineage.

Figure 15c also shows different failure scenarios for ownership, with a failure after 10s. The owner uses lineage reconstruction to recover quickly from a worker failure (1.9s in O;WF). Owner recovery is slower because the failed owner must re-execute from the beginning (8.8s in O;OF). To bound re-execution, we use application-level checkpoints (O+CP, checkpoints to a remote Redis instance once per second). Each checkpoint includes all intermediate state needed to transform the given frame, such as the cumulative sum so far (Figure 3b). When the sink receives the transformed frame, it "commits" the checkpoint by writing the frame's index to Redis. This results in negligible overhead (O vs. O+CP) and faster recovery (1.1s in O+CP;OF).

6 Related Work

Distributed futures. Several systems [4, 34, 37, 45, 48, 52] have implemented a distributed futures model. Most [37, 45] use a centralized master (Section 3.2). In contrast, ownership is a decentralized design that stores system state directly in the workers that invoke the tasks. Ray [34] shards the centralized state, but must still write to the centralized store *before* task execution and does not support automatic memory management. Lineage stash [52] is a complementary technique for recovering nondeterministic execution; ownership provides infrastructure for failure detection and memory management.

Other dataflow systems. Distributed data-parallel systems provide high-throughput batch computation and transparent data recovery [15, 25, 54, 56]. Many of our techniques build on



(a) (b) (c)

Figure 15: End-to-end benchmarks. **(a)** Image classification latency (right is p95-p100). **(b)** Online video stabilization latency. **(c)** Online video stabilization latency with failures (starting at p90). L=leases; O=ownership; CP=checkpointing; WF=worker failure; OF=owner failure.

these systems, in particular the use of distributed memory [25, 56] and lineage re-execution [15, 25, 54, 56]. Indeed, a data-parallel program is equivalent to a distributed futures program with no nested functions.

Most distributed data-parallel systems [15, 25, 54, 56] employ some form of centralized master, a bottleneck for applications with *fine-grained* tasks [32, 44, 51]. Naiad [35, 36] and Canary [44] support fine-grained tasks but, like other data-parallel systems, implement a *static* task graph, i.e. all tasks must be specified upfront. In contrast, distributed futures are an extension of RPC, which allows tasks to be dynamically invoked. Nimbus [32] supports both fine-grained *and* dynamic tasks with a centralized controller by leveraging *execution templates* for iterative computations. In contrast, ownership distributes the control plane and schedules tasks one at a time. These approaches are complementary; an interesting future direction is to apply execution templates to distributed futures.

Actor systems. Distributed futures are compatible with the actor model [7, 24]. Other actor frameworks [1, 12] already use futures for asynchrony, but with pass-by-value semantics, making it expensive to process large data. Actors can be extended with distributed memory to enable pass-by-reference semantics. Since distributed memory is immutable, it does not violate the condition of no shared state.

Our fault tolerance model is inspired by *supervision* in actor systems [7]. In this model, a supervisor actor delegates work to its children actors and is responsible for handling any failures among its children. By default, an actor also fate-shares with its supervisor. Our contribution is in extending the supervision model to *objects* and object recovery.

Parallel programming systems. MPI [18] exposes a low-level pass-by-value interface. In contrast, distributed futures supports pass-by-reference and heterogenerous processes.

Distributed futures are more similar in interface to other parallel programming runtimes [10, 14, 21, 31, 47]: the user annotates a sequential program to designate procedures that can be executed in parallel. Out of these systems, ownership is perhaps most similar to Legion [10], in that the developer specifies a task hierarchy that dictates system behavior. Our contribution is in identifying and addressing the challenges of failure detection and recovery for distributed futures.

Distributed memory. Distributed shared memory [40] provides the illusion of a single globally shared and *mutable* address space across a physically distributed system. Transparency has historically been difficult to achieve without adding exorbitant runtime overhead. Mutability makes consistency a major problem [11, 26, 28, 40], and fault tolerance has never been satisfactorily addressed [40].

More recent distributed memory systems [6, 9, 16, 27, 41] implement a higher-level key-value store interface. Most target a combination of performance, consistency, and durability. Similar to our use of distributed memory (Section 4.2), in-memory data replicas are used to improve durability and recovery time. Indeed, many of these systems could likely be used in place of our distributed memory subsystem.

However, the requirements of our distributed memory subsystem are minimal compared to previous work, e.g., durability is only an optimization. This is because we target an even higher-level interface that integrates directly with the programming language: unlike a key, a `DFut` can be used to express rich application semantics to the system, such as an RPC's data dependencies. Also, like previous data processing systems [15, 37, 56], data is immutable. Thus, fine-grained mutations are expensive, but consistency is not a problem.

7 Discussion

Ownership is the basis of the Ray architecture in v1.0+ [5], implemented in ∼14k C++ LoC. Previously, Ray used a sharded global metadata store [34]. There were two problems with this approach: (1) latency, and (2) worker nodes still had to coordinate for operations such as failure detection. Ray v0.7 introduced leases (Section 3.2), which solved the latency problem but not coordination. It became impractical to introduce distributed protocols involving multiple objects, such as for garbage collection. We designed ownership for this purpose.

While transparent recovery is an explicit goal of this paper, it is not the only benefit of ownership. Anecdotally, the two main benefits of ownership for Ray users are performance and reliability. In particular, reliability includes correct and timely failure detection and garbage collection. Notably, ownership-based transparent recovery is not yet widely used.

We believe that this is due to: (1) applications having custom recovery requirements that cannot be met with lineage reconstruction alone, and (2) the cost of transparent recovery. Thus, one design goal was to ensure that only applications that needed transparent recovery would have to pay the cost. Ownership is a first step towards this: it provides reliability to all applications and transparent object recovery as an option.

In the future, we hope to extend this work to support a *spectrum* of application recovery requirements. For example, we could extend ownership with options to recover actor state.

Acknowledgements

We thank our anonymous reviewers and our shepherd Ryan Huang for their insightful feedback. We also thank Alvin Cheung, Michael Whittaker, Joe Hellerstein, and many others at the RISELab for their helpful discussion and comments. In addition to NSF CISE Expeditions Award CCF-1730628, this research is supported by gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

References

[1] Akka. https://akka.io/.

[2] gRPC. https://grpc.io.

[3] Improved Fault-tolerance and Zero Data Loss in Apache Spark Streaming. https://databricks.com/blog/2015/01/15/improved-driver-fault-tolerance-and-zero-data-loss-in-spark-streaming.html.

[4] PyTorch - Remote Reference Protocol. https://pytorch.org/docs/stable/notes/rref.html.

[5] Ray v1.0. https://github.com/ray-project/ray/releases/tag/ray-1.0.0.

[6] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 1–14, 2009.

[7] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[8] Henry C Baker Jr and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGART Bulletin*, (64):55–59, 1977.

[9] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. {CORFU}: A shared log design for flash clusters. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 1–14, 2012.

[10] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[11] John K Bennett, John B Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, 1990.

[12] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, March 2014.

[13] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[14] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing*, 37(1):55–69, 1996.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[16] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 401–414, 2014.

[17] Elmootazbellah Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[18] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[19] Cary Gray and David Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, 1989.

[20] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 443–462, 2020.

[21] Robert H Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[22] Brandon Haynes, Amrita Mazumdar, Armin Alaghi, Magdalena Balazinska, Luis Ceze, and Alvin Cheung. Lightdb: A DBMS for virtual reality video. *Proc. VLDB Endow.*, 11(10):1192–1205, 2018.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[24] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[25] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[26] Pete Keleher, Alan L Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. *Distributed Shared Memory: Concepts and Systems*, pages 211–227, 1994.

[27] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[28] Kai Li. Ivy: A shared virtual memory system for parallel computing. *ICPP (2)*, 88:94, 1988.

[29] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7):260–267, 1988.

[30] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory oltp recovery. In *2014 IEEE 30th International Conference on Data Engineering*, pages 604–615. IEEE, 2014.

[31] Simon Marlow. *Parallel and concurrent programming in Haskell: Techniques for multicore and multithreaded programming*. " O'Reilly Media, Inc.", 2013.

[32] Omid Mashayekhi, Hang Qu, Chinmayee Shah, and Philip Levis. Execution templates: Caching control plane decisions for strong scaling of data analytics. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 513–526, 2017.

[33] Luc Moreau. Hierarchical distributed reference counting. In *Proceedings of the 1st international symposium on Memory management*, pages 57–67, 1998.

[34] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[35] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[36] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. Incremental, iterative data processing with timely dataflow. *Commun. ACM*, 59(10):75–83, September 2016.

[37] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 113–126, Berkeley, CA, USA, 2011. USENIX Association.

[38] D.G. Murray. *A Distributed Execution Engine Supporting Data-dependent Control Flow*. University of Cambridge, 2012.

[39] Robert Nishihara, Philipp Moritz, Stephanie Wang, Alexey Tumanov, William Paul, Johann Schleier-Smith, Richard Liaw, Mehrdad Niknami, Michael I. Jordan, and Ion Stoica. Real-time machine learning: The missing pieces. In *Workshop on Hot Topics in Operating Systems*, 2017.

[40] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52–60, 1991.

[41] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The RAMCloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):7, 2015.

[42] David Plainfossé and Marc Shapiro. A survey of distributed garbage collection techniques. In *International Workshop on Memory Management*, pages 211–249. Springer, 1995.

[43] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. Scanner: Efficient video analysis at scale. *ACM Trans. Graph.*, 37(4):138:1–138:13, July 2018.

[44] Hang Qu, Omid Mashayekhi, Chinmayee Shah, and Philip Levis. Decoupling the control plane from program control flow for flexibility and performance in cloud computing. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery.

[45] Matthew Rocklin. Dask: Parallel computation with blocked algorithms and task scheduling. In Kathryn Huff and James Bergstra, editors, *Proceedings of the 14th Python in Science Conference*, pages 130 – 136, 2015.

[46] Danilo Sato, Arif Wider, and Windheuser Christoph. Continuous delivery for machine learning, Sep 2019.

[47] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[48] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[49] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. Optimizing prediction serving on low-latency serverless dataflow. *arXiv preprint arXiv:2007.05832*, 2020.

[50] Andrew S Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[51] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Ali Ghodsi, Michael Armbrust, Benjamin Recht, Michael Franklin, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the Twenty-Sixth ACM Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017.

[52] Stephanie Wang, John Liagouris, Robert Nishihara, Philipp Moritz, Ujval Misra, Alexey Tumanov, and Ion Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 338–352, 2019.

[53] Stephanie Wang, Edward Oakes, and Frank Luan. Ownership nsdi'21 artifact. https://github.com/stephanie-wang/ownership-nsdi2021-artifact.

[54] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[55] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *In Correct Hardware Design and Verification Methods (CHARME '99), Laurence Pierre and Thomas Kropf editors. Lecture Notes in Computer Science, Springer-Verlag.*, volume 1703, pages 54–66, June 1999.

[56] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

A Distributed Reference Counting

Type	Description
Local reference	A flag indicating whether the `DFut` has gone out of the process's scope.
Submitted task count	Number of tasks that depend on the object that were submitted by this process and that have not yet completed execution.
Borrowers	The set of worker IDs of the borrowers created by this process, by passing the `DFut` as a first-class value.
Nested `DFuts`	The set of `DFuts` that are in scope and whose values contain this `DFut`.
Lineage count	Number of `Tasks` that depend on this `DFut` that may get re-executed. This count only determines when the lineage (the `Task` field) should be released; the *value* can be released even when this count is nonzero.

Table 3: Full description of the `References` field in Table 2. Every process with an instance of the `DFut` (either the owner or a borrower) maintains these fields.

If a `DFut` never leaves the scope of its owner, it does not require a distributed reference count. This is because the owner always has full information about which pending tasks require the object. However, since our API allows passing `DFuts` to other tasks as first-class values, we use a distributed reference count to decide when the object is out of scope.

Our reference counting protocol is similar to existing solutions [33, 42]. As explained in Section 4.2, the reference count is maintained with a tree of processes. Each process keeps a local set of borrower worker IDs, i.e. its children nodes in the tree. Most of the messages needed to maintain the tree are piggy-backed on existing protocols, such as for task scheduling.

A borrower is created when a task returns a `SharedDFut` to its parent task, or passes a `SharedDFut` to a child task. In both cases, the process executing the task adds the ID of the worker that executes the parent or child task to its local borrower set.

In many cases, a child task will finish borrowing the `DFut` by the time it has finished execution. Concretely, this means that the worker executing the child task will no longer have a local reference to the `DFut`, nor will it have any pending dependent tasks. Thus, when the worker returns the task's result to its owner, the owner can remove the worker from its local set of borrowers, with no additional messages needed. This optimization is important for distributing load imposed by reference counting among the borrowers, rather than requiring all reference holders to be tracked by the owner.

However, in some cases, the worker may borrow the `DFut` past the duration of the child task. There are two cases: (1) the worker passed the `DFut` as an argument to a task that is still pending execution, or (2) the worker is an actor and stored the `DFut` in its local state. In these cases, the worker notifies the owner that it is still borrowing the `DFut` when replying with the task's return value.

Eventually, the owner must collect all of the borrowers in its local set. It does this by sending a request to each borrower to reply once the borrower's reference count has gone to zero. Borrowers themselves never delete from their local set of borrowers. Once a borrower no longer has a reference or any pending dependent tasks, it replies to the owner with its accumulated local borrower set. The owner then removes the borrower, merges the received borrowers into its local set and repeats the same process with any new borrowers. If a borrower dies before it can be removed, the owner removes it upon being notified of the borrower's death.

When a `DFut` is *returned* by a task, it results in a nested `DFut`. Nested `DFuts` can be automatically flattened, e.g., when submitting a dependent task, but we must still account for nesting during reference counting. We do this by keeping a set of `DFuts` whose values contain the `DFut` in question in the ownership table (Table 2). The `DFut`'s value is pinned if its nested set is non-empty.

B Formal Specification

We developed a formal specification for the ownership-based system architecture [53]. It models the system state transitions of the ownership table for task scheduling, garbage collection, and worker failures. The goal is to check the correctness of the system design, which is manifested in the following properties:

- Safety: A future's lineage information is preserved as long as a task exists that depends on the value of the future. This is defined recursively: at any time, either the value of a future is stored inline (thus cannot be lost), or all futures that this future depends on for computing its value must be safe. Formally, it means the following invariant holds at any given time: $\forall x$,

$$\begin{aligned}&\text{LineageInScope}(x) \triangleq \\ &\quad \vee x = \texttt{INLINE\_VALUE} \\ &\quad \vee \forall arg \in x.args : \text{LineageInScope}(arg)\end{aligned}$$

- Liveness: The system will eventually execute all tasks and resolve all future values, even in case of failures, i.e., all `Get` calls eventually return.
- No Resource Leakage: The system will eventually clean up all task states and future values, after the all references to futures become out-of-scope.

We checked the model using the TLA$^+$Model Checker [55] for up to 3 levels of recursive remote function calls, where each function creates up to 3 futures, and verified that the safety and liveness properties hold in more than 44 million distinct states. Currently, the model does not include first-class futures or actors; we plan to include these and open-source the full TLA$^+$specification in the future.