In Reference to RPC: It's Time to Add Distributed Memory

Stephanie Wang, Benjamin Hindman, Ion Stoica
UC Berkeley

ACM Reference Format:
Stephanie Wang, Benjamin Hindman, Ion Stoica. 2021. In Reference to RPC: It's Time to Add Distributed Memory. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31-June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465302

1 Introduction

RPC has been remarkably successful. Most distributed applications built today use an RPC runtime such as gRPC [3] or Apache Thrift [2]. The key behind RPC's success is the simple but powerful semantics of its programming model. In particular, RPC has *no shared state*: arguments and return values are *passed by value* between processes, meaning that they must be copied into the request or reply. Thus, arguments and return values are inherently *immutable*. These simple semantics facilitate highly efficient and reliable implementations, as no distributed coordination is required, while remaining useful for a general set of distributed applications. The generality of RPC also enables *interoperability*: any application that speaks RPC can communicate with another application that understands RPC.

However, the lack of shared state, and in particular a shared address space, deters data-intensive applications from using RPC directly. Pass-by-value works well when data values are small. When data values are large, as is often the case with data-intensive computations, it may cause inefficient data transfers. For example, if a client invokes $x = f()$ then $y = g(x)$, it would have to receive x and copy x to g's executor, even if f executed on the same server as g.

There has been more than one proposal to address this problem by introducing a shared address space *at the application level* [14]. The common approach is to enable an RPC procedure to store its results in a shared data store and then return a *reference*, i.e., some metadata that acts as a pointer to the stored data. The RPC caller can then use the reference to retrieve the actual value when it needs it, or pass the reference on in a subsequent RPC request.

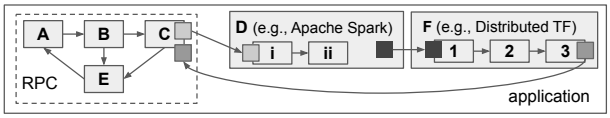


Figure 1: A single "application" actually consists of many components and distinct frameworks. With no shared address space, data (squares) must be copied between different components.

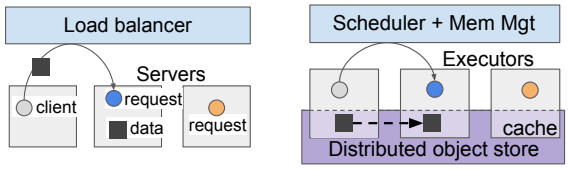


Figure 2: Logical RPC architecture: (a) today, and (b) with a shared address space and automatic memory management.

While a shared address space can eliminate inefficient or unnecessary data copies, doing it at the application level places a significant burden on the application programmer to manage the data. For example, the application must decide when some stored data is no longer used and can be safely deleted. This is a difficult problem, analogous to manual memory management in a non-distributed program.

Thus, instead of using RPC directly, distributed data-intensive applications tend to be built on top of specialized frameworks like Apache Spark [43] for big data processing or Distributed TensorFlow [6] for machine learning. These frameworks handle difficult systems problems such as memory management on behalf of the application. However, with no common foundation like RPC, *interoperability* between these frameworks is a problem. Resulting applications resemble Figure 1, where some components communicate via RPC and others communicate via a framework-specific protocol. This often results in redundant copies of the same data siloed in different parts of an application.

We argue that RPC itself should be extended with a shared address space and *first-class* references. This has two application benefits: 1) by allowing data-intensive applications to

HotOS '21, May 31-June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465302

be built directly on RPC, we promote interoperability, and 2) by shifting automatic memory management to a common system, we can reduce duplicated work between specialized frameworks. Indeed, we are already starting to see this realized by the latest generation of data systems, including Ciel [27], Ray [26], and Distributed PyTorch [4]. All implement an RPC-like interface with a shared address space. Our aim is to bring attention to the common threads and challenges of these and future systems.

At first glance, introducing a shared address space to RPC seems to directly contradict the original authors, who argued that doing so would make the semantics more complex and the implementation less efficient [12]. We believe, however, that these concerns stem not from a shared address space itself, but rather from supporting *mutability*. By adding an *immutable* shared address space, *we can preserve RPC's original semantics while enabling data-intensive applications.*

Immutability simplifies the system design, but what concretely would supporting a shared address space require? To answer this question, we first consider the design of RPC systems today. While RPC is often assumed to be a point-to-point communication between the caller and a specific callee, today's RPC systems look more like Figure 2a: a load balancer *schedules* an RPC to one of several servers (which can themselves act as an RPC client). Even the original authors of RPC provided such an option, known as *dynamic binding* [12]. Extending this architecture with an immutable shared address space would mean: (1) augmenting each "executor" with a local data store or cache, and (2) enhancing the load balancer to be *memory-aware* (Figure 2b).

In the rest of this paper we explain what it means to add first-class support for immutable shared state and pass-by-reference semantics to RPC. We give examples of applications that already rely on this interface today. Then, drawing from recent data systems, we discuss the challenges and design options in implementing such an interface.

2 API

There are two goals for the API: (1) It should preserve the simple semantics of RPC, and (2) It should allow the system to manage memory on behalf of the application. We use immutability to achieve the former. It is the simplest approach because the system does not need to define and implement a *consistency model* as part of the API. Meanwhile, the application remains free to implement mutability with local state.

We also briefly remark on the need for *parallelism*. Parallelism is of course critical to the performance of many data-intensive applications. It has already been addressed in part by the addition of *asynchrony* to RPC, which is why we focus here on the problem of memory management instead.

`Ref r`	A first-class type. Points to a value which may not exist yet and may be located on a remote node.
`shared(Ref r)→SharedRef`	Returns a copy of `r` that can be shared with another client by passing to an RPC.
`f.remote(Ref r) → Ref`	Invoke `f`. Pass the argument by *reference*: the executor receives the dereferenced argument. Returns a `Ref` pointing to the eventual reply.
`f.remote( SharedRef r) → Ref`	Invoke `f`. Pass the argument by *shared reference*: the executor receives the corresponding `Ref`. Returns a `Ref` pointing to the eventual reply.
`get(Ref r) → Val`	Dereference `r`. This blocks until the underlying value is computed and fetched to local memory.
`delete(Ref r)`	*Called implicitly when* `r` *goes out scope.* The client may not use `r` in subsequent API calls.

Table 1: A language-agnostic pass-by-reference API.

A common asynchronous API has each RPC invocation immediately return a *future*, or a pointer to the eventual reply [10, 25]. In Section 3, we explain how this in conjunction with a *reference*, i.e. a pointer to a possibly remote value, gives the system insight into the application, by giving a view of the client's future requests. We will assume a futures API but focus primarily on the introduction of first-class references.

First-class references. A *first-class* primitive is one that is part of the system API. For references, it means that the RPC system is involved in the creation and destruction of all clients' references. Compared to the original RPC proposal [12], there are three key differences in the API (Table 1):

First, all RPC invocations *return a reference* (of type `Ref`). The caller can dereference an RPC's reply when it needs it by calling `get`. We choose to have all RPC invocations return a reference so that the application never needs to decide whether an executor should return by value or by reference. Instead, this is decided transparently by the system. References are logical, so a system implementation can choose to pass back all replies by value, by reference, or both, e.g., depending on the size of the reply. A future extension to the API could allow applications to control this decision, if needed.

Second, the client can pass a reference as an RPC argument, in addition to normal values. There are two options for dereferencing `Ref` arguments. If a function with the signature `f(int x)` is passed a `Ref` argument, the system implicitly dereferences the `Ref` to its `int` value before dispatching `f` to its executor. Thus, the executor never sees the `Ref`. A function with the signature `f(Ref r)` would instead *share* the `Ref` argument passed by its caller, meaning that the executor can pass the `Ref` to another RPC or call `get`. The caller specifies the latter behavior by passing a `SharedRef` to `f`.

The developer must consider certain tradeoffs in this decision. With implicit dereferencing, the callee cannot begin execution until *all* of its `Ref` arguments are local. Implicit dereferencing also prevents efficient *delegation*: the callee does not

see the `Ref`, so it would have to copy the value to pass it to another RPC. On the other hand, with `SharedRefs`, the system has less visibility into how the callee will use the received `Ref`, i.e., whether and when it will call `get`. This has implications on optimizations for memory management (Section 3).

Third, a client uses the `delete` call to notify the system when it no longer needs a value. Note that this is *implicit*: it is not exposed to the application and should be called automatically by the language bindings when a `Ref` goes out of scope. This is important for memory safety, as we will see next.

3 Automatic memory management

First-class references allow the system to manage distributed memory on behalf of the application. For contrast, we will consider an application-level shared address space implementation that combines a key-value store with an existing RPC system. The application uses keys as references. We will call these *raw references* because their operations are not encapsulated by the RPC API. Thus, the system is fully or partially unaware of operations such as reference creation or deletion. We consider four key operations in memory management:

Allocation. The minimum requirement is the ability to allocate memory without specifying *where*. This is analogous to `malloc`, which handles problems such as fragmentation in a single-process program. This requirement is easily met by both raw and first-class references, as key-value stores do not require the client to specify where to put a value.

Reclamation. Reclaiming memory once there are no more references is a key requirement for applications with non-trivial memory usage. This is challenging in a distributed setting. It requires a fault-tolerant protocol for distributed garbage collection [33]. A key benefit of first-class references is that the API allows the system to implement this protocol on behalf of the application because *all reference creation and destruction operations are encapsulated in the API.*

In contrast, it is virtually impossible for the system to determine whether the application still holds a raw reference, analogous to determining whether a raw pointer is still in scope in a single-progress program. The problem is that raw references allow and even encourage the application to create a reference at any time, e.g., by hard-coding a string key. Thus, correctness requires manual memory management.

Movement. The primary motivation of pass-by-reference semantics is to eliminate unnecessary distributed data movement. The use of raw references shifts the responsibility of data movement to the system: the application has to specify when to move data (i.e., by calling `get`), but not how or where. The use of *first-class* references in combination with futures allows the system to also decide *when* to move data. By coordinating data movement with request scheduling, the system has greater control in optimizing data movement.

For example, a key system feature is *data locality*: when the data needed by a request is large, the system can choose an executor near the data rather than moving the data to the executor. This cannot be implemented with a `put`/`get` key-value store interface. It is straightforward with first-class references because each request specifies the `Refs` that it needs to the system, *before* placement.

Even if a key-value store were extended for data locality, we would still miss out on valuable optimizations, such as *pipelining of I/O and compute*. For example, if an executor had incoming requests $f()$ and $g(x)$, the system could schedule $f()$ while fetching x in parallel. The use of futures enables this optimization for requests from the *same* client, by allowing a client to make multiple requests in parallel.

Thus, the `Ref` API gives the system visibility into each request's data dependencies, affording unique opportunities in optimizing data movement with request scheduling. This also motivates our choice to expose two options for argument dereferencing, either implicit (by passing a `Ref`) or explicit (by passing a `SharedRef`). Much like raw references, there is ambiguity around if and when an executor will dereference a `SharedRef`, which affects the usefulness of system optimizations. For example, a request with `SharedRef` arguments may simply pass the references to another RPC instead of dereferencing them directly. In this case, scheduling the request according to data locality brings no benefit.

Memory pressure. To improve throughput, a single server machine generally executes multiple RPC requests concurrently. If the total memory footprint is higher than the machine's capacity, at least one process will be killed or swapped to disk by the OS, incurring high overheads [5, 37]. For example, with pure pass-by-value, the scheduler would not be able to queue new requests once the local memory capacity was reached. With raw references, each request would contain only references to its dependencies, so additional requests could be queued. However, this only defers the problem: the concurrent requests would still overwhelm the machine once they began execution and called `get` on their dependencies.

A *memory-aware scheduler* can ensure stability and performance by coordinating the memory usage of concurrent requests. This is true independent of a shared address space. However, a key challenge is determining each request's memory requirements. One could require developers to specify memory requirements, but in practice, this is very difficult.

First-class references give the system rich information about each request's memory requirements, *with no developer effort.* This is again because the scheduler has visibility into each request's dependencies. Thus, the scheduler could for example choose a subset of requests for which to fetch the dependencies, based on dependency size and the available memory.

Thus, only a system with first-class references can adequately handle reclamation, movement, and memory pressure for the application. In fact, we argue that *RPC systems with a shared address space should not expose raw references to the application.* If they do, it is at the developer's risk.

4 Is the API enough for applications?

The latest generation of distributed data systems shows us how valuable automatic memory management is to applications. We argue that these systems are in fact RPC systems with a *shared address space* (Table 2), even if they may not call themselves as such. All have a function invocation-like interface. Most use an immutable shared address space, and most use first-class references.

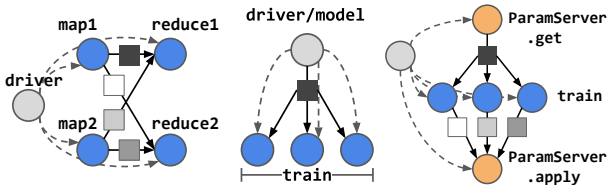
Despite the many API commonalities, these systems were originally proposed for a wide range of application domains, from data analytics to machine learning. We argue that the API proposed in Table 1 is sufficiently general because it can be used to express any application targeted by one of the systems in Table 2. To illustrate this, we will describe three concrete applications (Figure 3) that drove the development of some of these systems.

Data processing. Ciel is a universal execution engine for distributed dataflow processing that uses an API virtually identical to that proposed [27]. Unlike data processing systems such as Apache Hadoop [40] and Spark [43], which implement a *data-parallel* interface, Ciel uses *task parallelism*: each task is a function that executes on a data partition. For example, Figure 3a shows a simple map-reduce application with two tasks per stage. Ciel programs can also have *nested* tasks, similar to an RPC executor that itself invokes RPCs.

Ciel shows equal or better performance than Hadoop for synchronous data processing [27]. Futures allow the system to gain a full view of the dataflow graph before scheduling. First-class references enable management of data movement, e.g., by exploiting data locality. Finally, Ciel shows that its interface is also general enough for applications that cannot be expressed easily as data-parallel programs, such as dynamic programming problems [27].

Reinforcement learning. Ray [26] uses an API with futures and references to support emerging AI applications such as reinforcement learning (RL). RL requires a combination of *asynchronous* and *stateful* computation [30]. A typical algorithm proceeds in asynchronous stages: a driver sends the current model to a number of `train` tasks (Figure 3b). The `train` tasks are stateful because the executors hold an environment simulator in local memory.

Futures allow the driver to process the `train` results asynchronously[1]. First-class references are used to reduce redundant data copies when sending the model weights to the `train` tasks (Figure 3b). Ray's distributed object store optimizes this with: (1) shared memory to eliminate copies between executors on the same machine, and (2) a protocol for large data transfer between machines [26].



(a) Data processing (b) RL (c) Parameter server

```
# (a) MapReduce.
map_out = [map.remote(i) for i in range(m)]
out = [reduce.remote(map_out[i][j] for i in range(m))
       for j in range(r)]
get(out)

# (b) RL, one round.
refs = [train.remote(i, weights_ref) for i in range(3)]
while ready_ref = wait(refs):
    result = get(ready_ref)
    # ... apply the result ...

# (c) Parameter server, one round.
weights_ref = ps.get.remote()
refs = [train.remote(i, weights_ref) for i in range(3)]
ps.apply.remote(refs)
```

(d) Pseudocode using a pass-by-reference API (Table 1).

Figure 3: Applications for a pass-by-reference API. Legend: gray circle is the client, other circles are RPCs, dashed arrows are RPC invocation, solid squares are data, solid arrows are dataflow.

Parameter server. A primary motivation for Distributed PyTorch [4] and Distributed TensorFlow [7] is model training. A standard algorithm is synchronous stochastic gradient descent (SGD), using a parameter server to store the model weights [15, 23] (Figure 3c). In each round, each worker gets the current weights from the parameter server, computes a gradient, and sends the gradient to the parameter server to be aggregated. Synchrony is important to ensure that gradients are not stale [15].

Similar to Figure 3b, first-class references allow the system to optimize the broadcast of the current weights. The driver can also use references to concisely *coordinate* each round without having the data local (Figure 3d).

Both Distributed PyTorch and Distributed TensorFlow use an API similar to Table 1, extended with higher-level primitives specific to machine learning, e.g., optimization strategies [6, 24]. Distributed PyTorch allows mutable memory, and Distributed TensorFlow requires the developer to specify a static graph instead of using futures[2].

[1] Ray extends the API proposed in Table 1 with a `wait` call that returns the first ready reference, similar to `get` with a timeout.

[2] Distributed TensorFlow v2 supports eager execution, which produces a dynamic graph, similar to futures.

System	Applications	Immutable	First-class refs	Futures	Shared refs	Stateless fns	Stateful fns
Ciel [27]	Data processing	✓	✓	✓	✓	✓	×
Ray [26]	RL, ML	✓	✓	✓	✓	✓	✓
Dask [36]	Data analytics	✓	✓	✓	×	✓	✓/×
Distributed PyTorch [4]	ML	×	✓	✓	✓	×	✓
Distributed TensorFlow [6]	ML	✓	✓	×	×	✓	✓
Cloudburst [38]	Stateful serverless	×	✓/×	✓	✓	✓	✓

Table 2: RPC-like systems that expose a shared address space. Each system was designed for the listed application domain.

Summary. Given the overlap in API, we believe that these systems have encountered many of the same challenges in automatic memory management. The result is duplicated effort and inconsistent feature support. For example, CIEL handles memory pressure by using a disk-based object store, but has no method of reclamation [28]. Distributed PyTorch implements distributed reference counting for reclamation but throws out-of-memory errors to the application [4]. Thus, we ask: *how can we create a common foundation for data-intensive applications that is simple, efficient, and general enough*?

5 System Implementation

In practice, a shared address space needs a *distributed object store*. We recommend adding in-memory storage to each server, co-located with the "executor" (Figure 2b). For example, this could be a cache backed by a key-value store. In this case, why not use a key-value store directly? We argue that *to realize the benefits of pass-by-reference, we must* ***co-design*** *the scheduler and memory management systems*. We discuss this in terms of the key memory management operations:

Reclamation. Reclamation must ensure *memory safety*, i.e. referenced values should not be reclaimed, and *liveness*, i.e. values that are not referenced should eventually be reclaimed. A common approach is reference counting, to avoid the need for global pauses [33]. Concretely, this means that for each `Ref`, the system tracks: (1) whether the caller still has the `Ref` in scope, and (2) whether any in-flight requests have the `Ref` as an argument. The latter requires cooperation between memory management and the scheduler. *Shared references* (Section 2) further require the system to track (3) which *other* clients have the `Ref` in scope.

Movement. Co-design of scheduling and memory management gives the system greater control over data movement and sharing within the distributed object store. For example, a key-value store can improve data access time for skewed workloads by replicating a hot key. However, it must do this reactively, according to how often a key is used and where the keys are needed [9]. In contrast, a co-designed system could simultaneously schedule the consumers of some data with a broadcast protocol, e.g., using a dynamic multicast tree. There is also much previous work here that could be leveraged, including collective communication from HPC [19] and peer-to-peer networking systems such as BitTorrent [34].

Memory pressure. First-class references allow the system to control how much memory is used by the arguments of concurrent requests (Section 3). However, this is not enough to ensure available memory, as the size of a request's *outputs* are not known until run time. Thus, barring user annotations, the system must have a method for detecting and handling when additional memory is required by the application.

The system could: 1) throw an out-of-memory error, 2) kill and re-schedule a memory-hungry request, or 3) swap out memory (at an object granularity) to external storage. Options 1 and 2 are simple but cannot guarantee progress. Option 3, a standard feature in big data frameworks [35, 40, 43], guarantees progress but can impose high performance overheads. In some cases, simply limiting request parallelism can guarantee progress without having to resort to swapping. One challenge is in designing a scheduler that can efficiently identify, avoid, and/or handle out-of-memory scenarios.

5.1 Fault tolerance

Failures are arguably the most complex part of introducing a shared address space, as it implies that a reference and its value can have separate failure domains. We give some options in relation to current RPC failure handling.

A minimal RPC implementation guarantees *at-most-once* semantics: the system detects failures for in-flight requests and returns an error to the application. The difference with pass-by-reference is that an error can be thrown *after* the original function has completed, as failures can also cause the *value* to be lost from the distributed object store.

RPC libraries often support automatic retries, or *at-least-once* semantics. This is enough to transparently recover idempotent functions. The key difference in a pass-by-reference API is that, in addition to the failed RPC, any arguments passed by reference may also require recovery. Many systems in Table 2 support this through lineage reconstruction [26, 27, 36], replication [41], and/or persistence. Compared to pass-by-value RPC, these methods require storing and maintaining additional system state, as an object may be referenced well past the RPC invocation that created it.

Finally, for functions with side effects, application correctness may require the system to provide *exactly-once* semantics. The challenges here are much the same as with RPC without a shared address space [21].

6 Discussion

6.1 Interoperability

Data interoperability is an increasingly relevant problem with emerging solutions in a variety of settings, such as Weld [32] for parallelizing data analytics across libraries, Apache Arrow [1] for serialization of columnar data across programming languages, and (pass-by-value) RPC and REST for communication between distributed microservices [29]. We are interested in interoperability between *distributed applications that share large data*. Previous work provides key pieces to a solution: RPC provides communication while Apache Arrow provides zero-copy deserialization and language interoperability. However, we lack such a common solution for *distributed memory management*.

Many frameworks have solved distributed memory management for a *single* domain and within a *single* application. For example, Apache Spark [43] manages memory for big data applications, while Distributed TensorFlow [7] manages memory for machine learning applications.

However, combining applications across different domains is inevitable, and exchanging data between any two frameworks is a nontrivial problem [17]. For example, a common use case today is loading data that was preprocessed on an Apache Spark cluster into a Distributed TensorFlow cluster for model training or inference [17]. This adds *operations* complexity, as the user must manage two different clusters and their data exchange. There is also a *performance* cost because exchanged data must be copied or materialized, often to some distributed file system. In fact, these problems spurred the multiperson multiyear effort Project Hydrogen [42], a connector for Spark and deep learning frameworks.

As application use cases continue to evolve, we predict that the boundaries between applications will become increasingly blurred and that these scenarios will become increasingly common. The developer effort required to build and maintain a connector for every pair of applications or frameworks will become intractable.

Thus, our challenge is this: How can we factor out distributed memory management into a single system? We already take this for granted for small data, with RPC and REST as the de facto standards for building microservice APIs [29]. Can we develop such a standard for large data, too? Here, we discuss some of the practical challenges in adopting such a standard, compared to pass-by-value RPC.

Integration. How should applications integrate with a pass-by-reference RPC system? The option that we've primarily discussed is to develop specialized frameworks such as Apache Spark directly on the system. The advantages are reduced effort in managing distributed memory and execution, as well as straightforward interoperability with other applications or frameworks built with pass-by-reference RPC.

The challenge with this approach is in achieving the same performance as a custom memory and task management system that is designed for the specific application domain. An interesting direction is in determining what information is needed for different application domains and whether or how that can be specified to the system we propose.

The other option is a shallow integration: the application manages its own memory and execution but uses pass-by-reference RPC to communicate data to other applications. This could be useful for existing frameworks where reimplementing with pass-by-reference RPC is impractical.

The challenge with this approach is ensuring full flexibility for the application while still realizing the performance and interoperability benefits of pass-by-reference RPC. The simplest option that provides full flexibility is to have the application exchange data with the RPC system by copying, but this has the same overheads as pass-by-value RPC. Thus, we must carefully design the interface and division of responsibilities between the application and the RPC system. Integrating a custom object store implementation, for example, may require an API and support for plugins.

Decoupling applications. A key advantage of RPC is that clients and servers are decoupled, i.e., they share no state and can only communicate by passing values. This simplifies the development and deployment of applications such as microservices [29].

With pass-by-reference, one RPC application would create a reference, then pass it to another RPC application that dereferences it. Thus, the applications are *coupled* for at least the lifetime of the reference. Unlike pass-by-value RPC, this can extend past the duration of the RPC invocation, e.g., if the server saves the `Ref` in its local state.

One challenge in system design is in keeping application development and deployment simple. Immutability prevents consistency issues, but it is not a panacea. For example, how should failures in one application, say the executor that was supposed to produce a value, affect the other? What is required to connect two applications that use different recovery semantics for data passed by reference?

6.2 The role of programming languages

Historically, fully transparent distributed memory systems (e.g., DSM) have proven impractical. We believe that systems must at least partially expose distributed memory to programmers. Memory management should be automatic, but developers must be aware of how to best use references. Much like non-distributed programming models, the flexibility of RPC makes this challenging, as there are many ways to write the same program. Yet, flexibility is also a primary reason for RPC's success.

In the future, we also believe there are opportunities in leveraging techniques from programming languages to improve memory management and even some of the interoperability issues described here. For example, the concept of ownership [16] popularized by the language Rust could provide a solution to the problem of tight coupling between applications that share a reference.

6.3 Related abstractions for distributed memory

Distributed shared memory [31] (DSM) provides the illusion of a single globally shared address space across physically distributed threads. Our RPC proposal has two differences: (1) shared memory is *immutable*, and (2) the use of futures and first-class references. The former decision is informed by the historical difficulties of implementing DSM in practice [11, 20, 22, 31]. The latter is valuable for capturing richer application semantics that enable automatic memory management (Section 3).

Some systems have introduced novel and rich abstractions to manage consistency for mutable shared state [8, 18, 41]. For example, FaRM [18] uses distributed transactions while Anna [41] offers a range of consistency levels. We chose a minimal approach that preserves the pass-by-value semantics of RPC and avoids imposing a consistency model. This does not preclude developers from using or implementing other consistency models at the application level.

Other systems introduce new abstractions for accessing remote memory, including distributed data structures [13] and primitives for a single "object" [37]. These are fundamentally different approaches to programming distributed memory. In particular, we call for tightly coupling the notion of *functions* with remote memory (Section 2) and *co-designing* function scheduling and memory management (Section 5).

Section 4 summarizes modern system manifestations of pass-by-reference RPC. Many have handled some but not all of the problems in memory management discussed in Sections 3 and 5. None have fully addressed the challenges of interoperability (Section 6.1), which is unsurprising given that it was not their explicit goal. For example, the concept of ownership in the system Ray handles automatic memory reclamation and recovery, but requires all references to be coupled to their creator, which is a problem for interoperability [39].

6.4 Conclusion

Memory management is a key part of distributed systems. The goal of this work was to extract a common API that has emerged in recent data-intensive systems that could be used to factor out problems in distributed memory management. The result is pass-by-reference RPC. We believe that pass-by-reference RPC has the potential to act as a unified abstraction for "virtual memory" in distributed applications, enabling interoperability and faster development of future applications.

References

[1] Apache Arrow. https://arrow.apache.org/.

[2] Apache thrift. https://thrift.apache.org/.

[3] grpc. https://grpc.io.

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

[5] Taming the OOM killer. https://lwn.net/Articles/317814/.

[6] 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 ({OSDI} 16)*, pages 265–283, 2016.

[7] 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 *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[8] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: a CALM and collected approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260. www.cidrdb.org, 2011.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[10] Henry C. Baker, Jr. and Carl Hewitt. The incremental garbage collection of processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, New York, NY, USA, 1977. ACM.

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

[13] Benjamin Brock, Aydın Buluç, and Katherine Yelick. Bcl: A cross-platform distributed data structures library. In *Proceedings of the 48th International Conference on Parallel Processing*, pages 1–10, 2019.

[14] DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, Eduardo Pinheiro, and Michael L. Scott. Interweave: A middleware system for distributed shared state. In Sandhya Dwarkadas, editor, *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[15] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[16] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.

[17] Jason Jinquan Dai, Yiheng Wang, Xin Qiu, Ding Ding, Yao Zhang, Yanzhang Wang, Xianyan Jia, Cherry Li Zhang, Yan Wan, Zhichao Li, et al. Bigdl: A distributed deep learning framework for big data. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 50–60, 2019.

[18] 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.

[19] 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.

[20] 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.

[21] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86, 2015.

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

[23] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 583–598, 2014.

[24] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

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

[26] 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.

[27] 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.

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

[29] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[30] 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.

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

[32] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman Amarasinghe, and Matei Zaharia. Weld: A common runtime for high performance data analytics. Jan 2017.

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

[34] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Proceedings of the 4th International Conference on Peer-to-Peer Systems*, IPTPS'05, page 205–216, Berlin, Heidelberg, 2005. Springer-Verlag.

[35] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 193–206, 2019.

[36] 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.

[37] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. {AIFM}: High-performance, application-integrated far memory. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 315–332, 2020.

[38] 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.

[39] Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Sifei Luan, Audrey Cheng, and Ion Stoica. Ownership: A distributed futures system for fine-grained tasks. 2021.

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

[41] Chenggang Wu, Jose M. Falerio, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[42] Reynold Xin. Project hydrogen: Unifying state-of-the-art ai and big data in apache spark. Spark + AI Summit, 2018.

[43] 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.