Persistent Actor State in Ray

Robert Imschweiler
Technical University Munich

Christoph Schnabl
Technical University Munich

Abstract

In the following report, we briefly summarize the paper Ownership: A Distributed Futures System For Fine-Grained Tasks [9]. We give an introduction to its architecture, an overview of its key protocols, and a concise evaluation. We then propose a concept for better actor integration allowing for fault tolerance for actors and the ownership table. Finally, we introduce an implementation that simplifies configurable application-level recovery for actors in Ray [7] and benchmark it against the implementation without any actor recovery to quantify its overhead.

1 Introduction

With the ever-increasing need for distributed computing, applications face challenges regarding scalability and fault tolerance. This is even more important for use cases with high data parallelism and low latency requirements such as model serving in AI or online video processing.

Ownership is a system to implement distributed futures to encompass this use case. The authors implemented and evaluated their proposed solution within the Ray framework for building distributed applications.

In the following, we give a brief overview of ownership's design and explain those aspects most relevant for our proposed adaptations. We are first going to explain its API, architecture, and key concepts. We then introduce ownership's protocols for task scheduling, memory management, and failure recovery.

Ownership implements distributed futures with the semantics of non-blocking remote procedure calls (RPCs) and distributed memory. In comparison to blocking calls, additional computation can happen asynchronously thus allowing for a potential higher throughput. Arguments below a threshold size are passed by value. Bigger objects are passed with a reference to their location in distributed memory. This can reduce the amount of data that needs to be copied over the network for certain use cases.

2 Ownership's architecture

Ownership requires idempotency for its *tasks* and immutability for its *objects*. *Tasks* are created by invoking a remote procedure call that immediately returns a *distributed future* (`DFut`) referencing an *object* in *distributed memory*. Its value may be dereferenced by invoking `get` on the `DFut` blocking until the result of the computation is present.

Ownership is called ownership because the caller of a *task* is the owner of the returned `DFut` and as such is responsible for its metadata and lifetime. On passing the `DFut` to another task, it gets dereferenced by default or can be converted into a shared distributed future (`SharedDFut`). Holders of a `SharedDFut` are called *borrowers* and only pass the reference to tasks further down the call tree avoiding redundant network traffic.

Actors are *stateful tasks* that are local to one worker. In programming terms, they are instances of a class created by invoking a remote constructor function returning a reference to this newly created actor (`ARef`).

This architecture poses several requirements to the *metadata* for each object. It has to be stored in a fault-tolerant way and represent the current physical cluster state. In the context of the normal operation, the metadata gets asynchronously updated in order to resolve the location of a future's value and for reference counting within garbage collection. In the context of failure detection and recovery, the metadata needs to provide appropriate information such as the location of currently executing tasks and the owner hierarchy, the lineage, of futures.

A cluster in ownership consists of multiple nodes, each node itself consisting of one *local scheduler*, one *object store* that is part of the cluster's distributed memory, and multiple *workers*. Each *worker* executes tasks and actors and manages their metadata in its *ownership table*. For each of their objects, *borrowers* store the ID, owner, its value (if present), and a list of sub-tasks that reference this object. *Owners* additionally include the location of the object (if not yet dereferenced) and the callees arguments.

2.1 Task Scheduling

For each execution of a task, the owner of the emerging future requests resources from its local scheduler. The request contains information about resources such as required objects. The scheduler responds with the new location of the task, either a local worker or a remote node. Before dispatching the new location is synchronously updated in the owner's ownership table. Depending on the location the task is then dispatched to a local or remote worker.

2.2 Memory management

Depending on the object size, an object is either transferred by value or by reference. Additional copies of a pass-by-value object are not used for recovery. There is, however, the need to balance the speed of dereferencing versus the overhead of additional copies of an object. Regarding the reclamation of memory, there is a distributed reference counter for every object passed by reference, so that the owner of the object can detect whenever the object's memory can be released. This information can be retrieved from the ownership table's `References` column. The distributed memory layer exposes a Key-Value interface and is implemented by the Plasma object store within Ray [2].

2.3 Failure recovery

Failure Recovery with a centralized master is simple to implement, however, leads to a bottleneck in scalability, even when sharding the master.

Detecting a memory failure with blocking RPC's and distributed memory is easy as callees know the location of data when receiving the reference whereas this is not the case with distributed futures. Recovery of failed idempotent futures only includes resubmitting the future. With distributed futures, the reference to distributed memory is lost as it is only known upon receiving the reference.

2.3.1 Failure detection

A failed worker can be detected because the scheduler sitting on the worker's node publishes a failure notification. A failed node can be detected by the other nodes because there is a heartbeat exchange between the nodes. After a failure, every worker checks their ownership table in order to see whether one of its distributed futures is affected by the failure. This is the case if either the underlying object or its owner is lost. Both recovery cases are explained in more detail below.

2.3.2 Object recovery

Object Recovery is implemented through *lineage reconstruction* in case the object is permanently lost from distributed memory. First, if *secondary copies* are still available in any of the nodes' caches expensive reconstruction can be avoided. This metadata is retrieved from Ray's *Global Control Store*. Secondary copies of objects are only released from memory *lazily* under memory pressure to increase the chances to skip lineage reconstruction.

If that is not the case lineage reconstruction is accomplished by storing each invoked task and its arguments in the *ownership table*. Then the task that created the object is resubmitted. This is done *recursively* on the arguments of this task [4].

2.3.3 Owner recovery

In order to avoid dangling `DFut` pointers fate-sharing is used as described in more detail in section 4.2. More specifically fate-sharing within owner recovery involves that all objects from the owner and also all references to these objects are freed from memory. Lineage reconstruction is used to recover the lost state. Actor recovery as well as application-level checkpoints and recovery of the ownership table are not implemented in the ownership paper.

3 Evaluation

To summarize, the presented system uses ownership in order to leverage distributed futures in a way that combines automatic memory management, failure detection and recovery based on lineage reconstruction and fate-sharing, and the ability to execute fine-grained tasks without posing too much overhead.

The authors provide a working implementation of their system with the enhancement of the existing Ray framework and thus, their approach already found widespread adoption. They ran benchmarks to compare against previous versions of Ray. As a very concise takeaway - ownership performs very well in terms of throughput for large objects or nested tasks. For nested tasks utilizing borrowing improves latency even for remote tasks and actor execution.

However transparent recovery is not yet widely used within the Ray community due to the following reasons. Applications may have custom failure recovery or need recovery for scenarios that have not been implemented yet in Ray. Lastly, the cost of transparent recovery may simply be too high. With our approach, we tackle the first two of these points to contribute to the discussion on how to improve Ray and ownership.

4 Actor fault tolerance

Fault tolerance for actors is important, as their reconstruction is more expensive compared to tasks due to the additional local state. In our proposal we stick to the constraints imposed by ownership, namely fine-grained tasks with milliseconds of latency, high data parallelism that makes pass-by-value semantics processing expensive, and data sizes of up to multiple megabytes.

4.1 Actors in Ray

While tasks are stateless functions similar to the functional programming paradigm, actors are objects having an internal state which may change on method invocations. Usually, actors communicate with each other using messages and then act according to the message content. Additionally, every actor can also create other actors and tasks. Actors in Ray are typically used for orchestration and scheduling of tasks.

In addition to sticking to ownership's initial assumptions as well as keeping changes as non-invasive as possible, our proposal adheres to the following design goals:

1. **Everything is an actor** - Tasks are actors which do not use their internal state. Workers are actors with the ownership table as their state. Actors can execute other tasks and actors.

2. **Avoid shared state** as this would imply even costlier synchronization mechanisms not suitable for fine-grained tasks.

3. **Failures do occur frequently** Recovery is configurable from an API point of view offering more user flexibility.

4. Data in the Object Store remains **immutable**.

4.2 Proposed approach

Our idea unifies the entities used by Ray's ownership system. Instead of differentiating between workers, tasks, and actors, every working component is an actor. This does not only allow us to make actors first-class citizens and provide them with basic built-in failure recovery but also enables enhanced possibilities for failure recovery beyond single actors. We are also going to elaborate on the yet missing recovery of the ownership table.

Our changes do not break the API but impose several changes to Ray's implementation to encompass the additional fault recovery requirements. Similarly, the protocols for scheduling, memory management, and failure detection do not require changes. Our approach would both allow for actor recovery to be implemented and owner recovery to be improved by using the same object recovery mechanism.

Currently, in case of a worker failure, all objects of this owner fate-share and will be released and the scheduler revokes the owner's computing time. Because of fate-sharing, we cannot avoid lineage reconstruction, as the involved objects simply do not exist anymore. This is indeed necessary for the original implementation to avoid dangling pointers to a `DFut` caused by simultaneous object and owner failure. Consequently, the owner of the failed owner has to reconstruct all dependencies and task descendants, which is fatal for tasks with a deep call tree. Because of this limitation actors are also not allowed to store references in their local state.

Treating workers as actors with the ownership table as their local state allows us to you utilize actor fault recovery to also persist the ownership table. Thus for a failed owner, we propose the following protocol.

First, restart the worker process of the owner and load its persisted ownership table. If all required objects still exist in distributed memory, which can easily be queried from the Global Control Store in Ray, all tasks are going to be executed as usual. In case some objects do not exist anymore, neither as primary copies nor in borrowers caches, we can rely on the default object recovery mechanism as described above and thus avoid expensive lineage reconstruction requisite with fate-sharing.

That way dangling pointers to `DFuts` can be avoided because we can guarantee increased availability of the ownership table by synchronously persisting each change. Looking at Ray's source code, we can back this claim with the following post-variant of the object recovery algorithm. Recovery fails if and only if, (a) we do not own the object, (b) the object still exists as a primary copy, or (c) we do not have any metadata about this object anymore. Note that because of lineage construction this has to hold for all objects in the call tree.

In case *worker C* tries to dereference *object A* owned by *worker A*, but its only copy is stored in the distributed object store at *worker B*. Without failures, *worker A* would redirect the request to *node B* which then would populate the future with the value of X. If the owner of the object, so *worker A* in this example, fails, their ownership table also fails. If the object store of *worker B* fails as well, the value of X is lost.

Worker C again tries to dereference *X* and as *worker A* has no more information about the location of *X*, and subsequently searches all nodes of the distributed object store for additional copies. However, the only object-store node holding a copy has also failed. This error cannot be recovered by lineage reconstruction as the necessary metadata, residing in A's ownership table, is also lost. This is mitigated by a persistent ownership table.

5 Application level fault tolerance

In order to overcome the shortcomings of Ray's actor failure recovery, "checkpointing" has been established among users as a useful way to accomplish application-level recovery for actor state [3]. The advantage of this practice is that the actor state can be persisted very fine-grained and only when needed. On the other hand, this means that the user has to care about all the recovery logic. In that regard, a solution built into the Ray API would be a real benefit, even though it cannot always provide the same optimal performance.

However, an enhancement for the Ray system does well to follow Ray's API philosophy and use simple patterns and decorators to make a basic functionality as accessible as possible, while not preventing users from adding their own, more advanced, custom methods. This implies that it is not practical to follow an approach such as implemented in other systems - Akka [1], for instance - where the API user explicitly declares state changes. Similarly dynamically computing state differences within the API adds run-time overhead that we cannot control because it depends on the concrete data structures employed by the user.

6 Implementation

Our prototype enhances and makes use of the official Ray Python API [8]. We introduce an internal helper class that is responsible for the background work and a new decorator `safe_state` which can be used by the API user to decorate methods of an actor. The decorator expects a variable number of actor attribute names as parameters and saves those specified attributes after each invocation of the decorated method. For instance, in our demo code [6], the `counter` attribute is saved automatically after each invocation of `increment()`. Furthermore, each attribute gets initialized by the internal `ActorState` object using the previously stored value, or a default value if there is no saved state for this attribute.

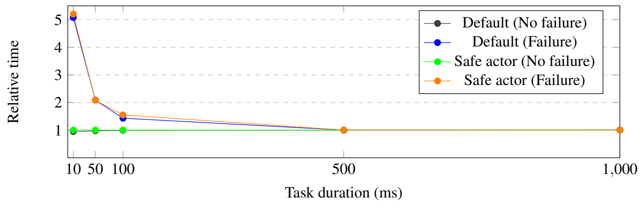

Figure 1: Small actor state (single integer)

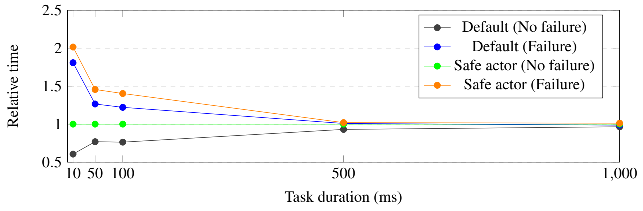

Figure 2: Large actor state (10 MiB numpy array)

Figure 1 and Figure 2 show some benchmarks of our prototype. Compared to the default Ray implementation, the persistent actor state poses a negligible overhead for small actor state. For larger objects to persist (10 MiB in this case), our prototype naturally takes more time than the default Ray system without object reconstruction and without a persistent actor state.

7 Conclusion

To conclude, our approach tries to simplify Ray and the ownership system by building its core components – workers, tasks, actors – on the actor model. By everything being an actor, the failure handling gets unified and actor state recovery, as well as ownership table recovery, can both be implemented using the same protocols. Additionally persisting the ownership table leads to simpler owner recovery saving unnecessary lineage reconstruction and enabling more concise handling of failures. As shown in our implemented prototype, our approach is easy to apply and does not pose much overhead. Further extensions of this work would be the evaluation regarding the performance differences between this approach and the lineage reconstruction by the ownership system and more elaborate ways to persist small updates to large data structures within the actor state.

Acknowledgements

We thank the authors of ownership and Ray for their open-source work. We also thank our seminar colleagues for their input and exchange. Lastly, we would like to express our gratitude to Pramod Bathotia for offering this seminar course and introducing us to systems research.

Availability

Code for the benchmark as well as the implementation of actor recovery is available at [5].

References

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

[2] *Apache Arrow - Plasma Object Store*. https://arrow.apache.org/docs/python/plasma.html.

[3] *Ray Design Patterns*. https://docs.google.com/document/d/167rnnDFIVRhHhK4mznEIemOtj63IOhtIPvSYaPgI4Fg/preview.

[4] *Ray v1.9.1 object recovery source code*. https://github.com/ray-project/ray/blob/4ab059eaa1158137ea3b6fd0e2319a3836283388/src/ray/core_worker/object_recovery_manager.h.

[5] Robert Imschweiler and Christoph Schnabl. *Prototype for Actor Fault Tolerance for Ray*. https://github.com/ro-i/ray_persistent_actor.

[6] Robert Imschweiler and Christoph Schnabl. *Prototype for Actor Fault Tolerance for Ray – Demo*.

https://github.com/ro-i/ray_persistent_actor/blob/main/demo.py.

[7] 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)*, pages 561–577, Carlsbad, CA, October 2018. USENIX Association.

[8] Ray. *API and Package Reference*. https://docs.ray.io/en/latest/package-ref.html.

[9] 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. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 671–686. USENIX Association, April 2021.