Ownership: A Distributed Futures System for Fine-Grained Tasks

Stephanie Wang, Eric Liang, Edward Oakes, Ben Hindman, Frank Luan, Audrey Cheng, Ion Stoica
Outline

1. An overview of distributed futures
2. System requirements and challenges
3. Ownership: Achieving fault tolerance without giving up performance
4. Evaluation
RPC model

\[ o_1 = f() \]
\[ o_2 = f() \]
\[ o_3 = \text{add}(o_1, o_2) \]

Problems:

- Data movement
- Parallelism
Data movement: RPC model +distributed memory

Distributed memory: Ability to reference data stored in the memory of a remote process.

- Application can pass by reference
- System manages data movement
Parallelism: RPC model +futures

Futures: Ability to reference data that has not yet been computed.

- Application can specify parallelism and data dependencies
- System manages task scheduling
Distributed futures

- **Performance:** System handles data movement and parallelism
- **Generality:** RPC-like interface (data is immutable). Application does not specify when or where computation should execute.
Distributed futures today

Distributed futures are growing in popularity, with applications in a variety of domains:

- Data processing: CIEL, Dask
- Machine learning: Ray, Distributed PyTorch

Most systems focus on coarse-grained tasks (>100ms):

- A centralized master for system metadata.
- Lineage reconstruction (re-execution of the tasks that created an object) for fault tolerance.
A distributed futures system for fine-grained tasks

For generality, the system must impose low overhead.

Analogy: gRPC can execute millions of tasks/s. Can we do the same for distributed futures?

Goal: Build a distributed futures system that guarantees fault tolerance with low task overhead.

Enable applications that dynamically generate fine-grained tasks. → Check out the paper for more details!
Outline

1. An overview of distributed futures
2. System requirements and challenges
3. Ownership: Achieving fault tolerance without giving up performance
4. Evaluation
Distributed futures introduce shared state

Legend

- Task (RPC)
- Invocation
- Data dependency

Diagram:
- Driver
- Function calls: f() o1
- Data dependency: add(o1, o2)
Distributed futures introduce shared state

Multiple processes refer to the same value.

1. The process that specifies how the value is created and used.
2. The process that creates the value.
3. The process that uses the value.
4. The physical location of the value.

Dereferencing a distributed future requires coordination.
System requirements

Requirements for dereferencing a value:

- **Retrieval**: The location of the value
- **Garbage collection**: Whether the value is referenced

Requirements in the presence of failures:

- **Detection**: The location of the task that returns the value.
- **Recovery**: A description of the task and its dependencies.
- **Persistence**: Metadata should survive failures.
System requirements

Requirements for dereferencing a value:

- **Retrieval:** The location of the value
- **Garbage collection:** Whether the value is referenced

Requirements in the presence of failures:

- **Detection:** The location of the task that returns the value.
- **Recovery:** A description of the task and its dependencies.
- **Persistence:** Metadata should survive failures.

**Challenge:** Recording this metadata, while ensuring latency and throughput for dynamic and fine-grained tasks.

**Persistence:** Metadata should survive failures.
## Existing solutions

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Coordination</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized master</td>
<td><strong>Master</strong> records all metadata updates and handles all failures.</td>
<td>Can scale through sharding, but high overhead due to synchronous updates.</td>
</tr>
<tr>
<td>Leases (decentralized)</td>
<td><strong>Workers coordinate</strong>. For example, use leases to detect a task failure.</td>
<td><strong>Asynchronous</strong> metadata updates. Scale by adding more worker nodes.</td>
</tr>
</tbody>
</table>
Outline

1. An overview of distributed futures
2. System requirements and challenges
3. **Ownership: Achieving fault tolerance without giving up performance**
4. Evaluation
Our approach: Ownership

Existing solutions do not take advantage of the inherent **structure** of a distributed futures application.

1. Task graphs are hierarchical.
2. A distributed future is often passed within the scope of the caller.
Our approach: Ownership

Existing solutions do not take advantage of the inherent structure of a distributed futures application.

1. Task graphs are hierarchical.

2. A distributed future is often passed within the scope of the caller.

Insight: By leveraging the structure of distributed futures applications, we can decentralize without requiring expensive coordination.
Our approach: Ownership

Insight: By leveraging the structure of distributed futures applications, we can decentralize without requiring expensive coordination.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Failure handling</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Ownership:</strong> The worker that calls a task <strong>owns</strong> the returned distributed future.</td>
<td>Each <strong>worker</strong> is a “<strong>centralized master</strong>” for the objects that it owns.</td>
<td><strong>No additional writes</strong> on the critical path of task execution. Scaling through <strong>nested function calls</strong>.</td>
</tr>
</tbody>
</table>
Ownership: Challenges

- Failure recovery
  - Recovering a lost worker
  - Recovering a lost owner
- Garbage collection and memory safety
- Handling *first-class distributed futures*, i.e. distributed futures that leave the caller’s scope
Ownership: Challenges

- Failure recovery
  - Recovering a lost worker
  - Recovering a lost owner
- Garbage collection and memory safety
- Handling first-class distributed futures, i.e. distributed futures that leave the caller’s scope

→ Check out the paper for more details!
Task scheduling

Node 1

<table>
<thead>
<tr>
<th>Obj</th>
<th>Task</th>
<th>Val</th>
<th>Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>B()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>C(X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node 2

Worker

Object Store

Node 3

Worker

Object Store

Diagram:
- Node 1: Worker (A)
- Node 2: Object Store
- Node 3: Worker

Diagram shows dependencies with arrows:
- Node A to B
- Node B to X
- Node C to X
- Node X to Y
Task scheduling

A task’s pending location is written locally at the owner.
Owner tracks locations of objects stored in distributed memory.
Task scheduling with dependencies

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Worker</strong></td>
<td><strong>Object Store</strong></td>
<td><strong>Worker</strong></td>
</tr>
<tr>
<td><strong>Obj</strong></td>
<td><strong>Task</strong></td>
<td><strong>Val</strong></td>
</tr>
<tr>
<td>X</td>
<td>B()</td>
<td>*X</td>
</tr>
</tbody>
</table>

Diagram:
- Node 1: Worker with tasks B() and C(X)
- Node 2: Object Store with task X
- Node 3: Worker with object 0.
Worker failure

Reference holders only need to check whether the owner is alive.
Worker recovery

Owner coordinates lineage reconstruction.
Owner failure
Owner recovery

References fate-share with the object’s owner.
Owner recovery

References fate-share with the object’s owner.
Owner recovery

References fate-share with the object’s owner.
Leveraging the application’s hierarchical structure: the owner of A recovers A.
Outline

1. An overview of distributed futures
2. System requirements and challenges
3. Ownership: Achieving fault tolerance without giving up performance
4. Evaluation RAY
Evaluation: Online video processing

Legend
- Task (RPC)
-Invocation
-> State dependency

1. Tasks in the milliseconds
2. Complex data dependencies
3. Pipelined parallelism
Evaluation: Online video processing (60 videos)

Centralized = Ray modified with writes to a centralized metadata store
Evaluation: Online video processing (60 videos)

Latency with ownership is lower because each video has a different owner.
Evaluation: Online video processing with failures

Recovery when the owner is intact, with lineage reconstruction.
Evaluation: Online video processing with failures

Recovery from owner failure using application-level checkpoints to bound re-execution.
Conclusion

**Key insight:** Decentralize system operations according to the application structure.

**Ownership:** A decentralized system for distributed futures that achieves transparent recovery and automatic memory management.

  Enables data-intensive applications with *fine-grained* tasks.

github.com/stephanie-wang/ownership-nsdi2021-artifact

github.com/ray-project/ray

Email: swang@cs.berkeley.edu