Ownership: A Distributed Futures System for Fine-Grained Tasks

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

Berkeley
UNIVERSITY OF CALIFORNIA

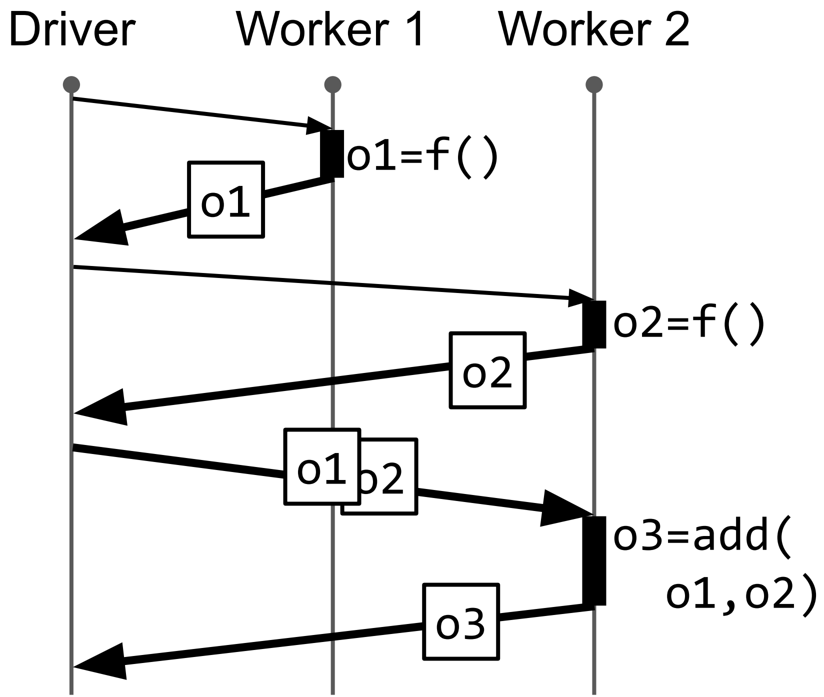
riselab
UC Berkeley

anyscale

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

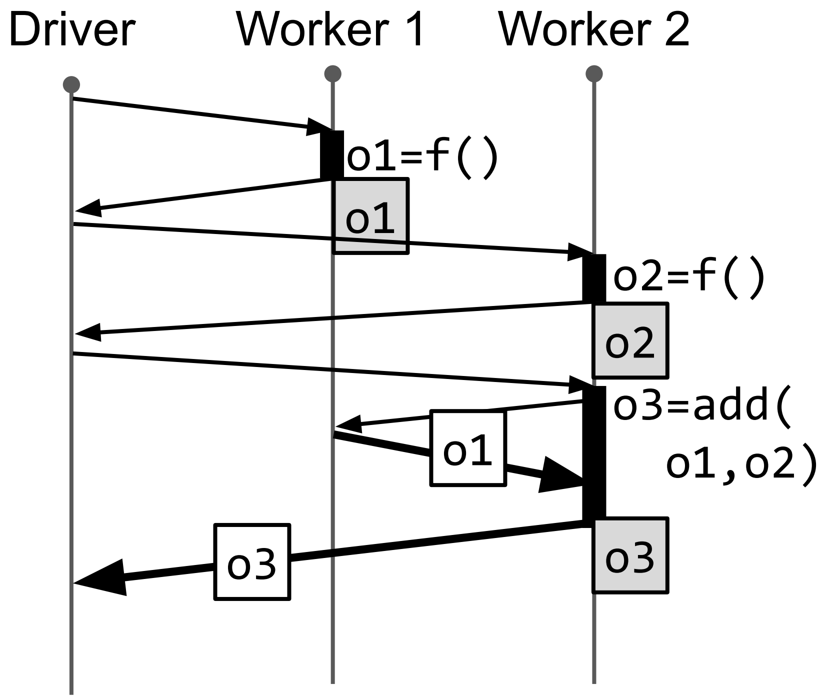



```
o1 = f()
o2 = f()
o3 = add(o1, o2)
```

Problems:

- Data movement
- Parallelism

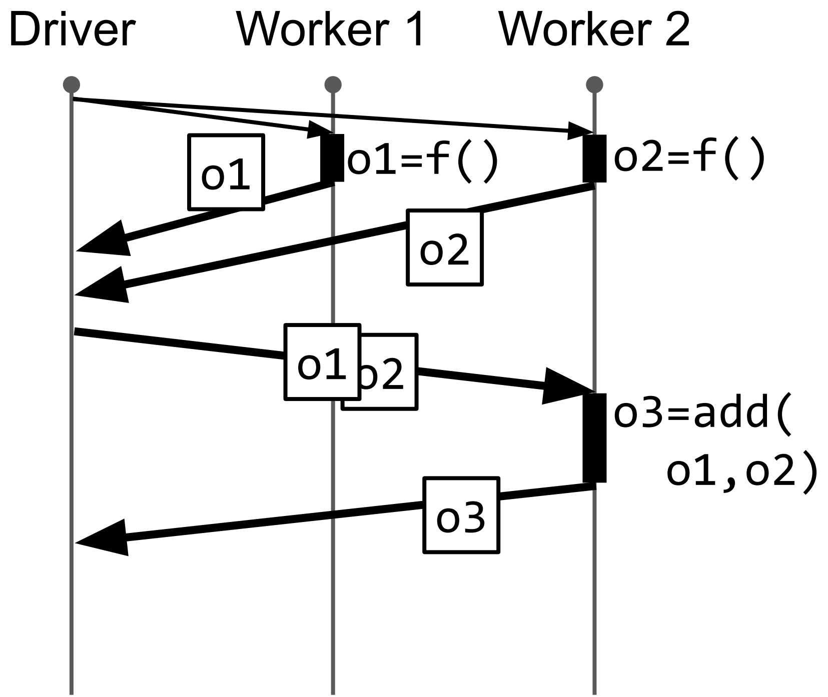
Data movement: RPC model **+distributed memory**



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

- Application can **pass by reference**
- System manages **data movement**

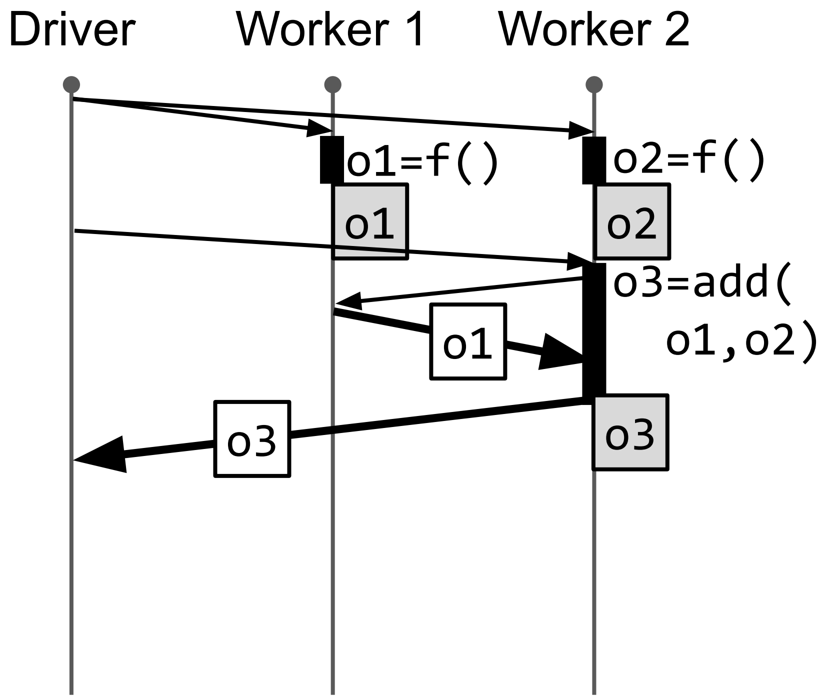
Parallelism: RPC model **+futures**



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?

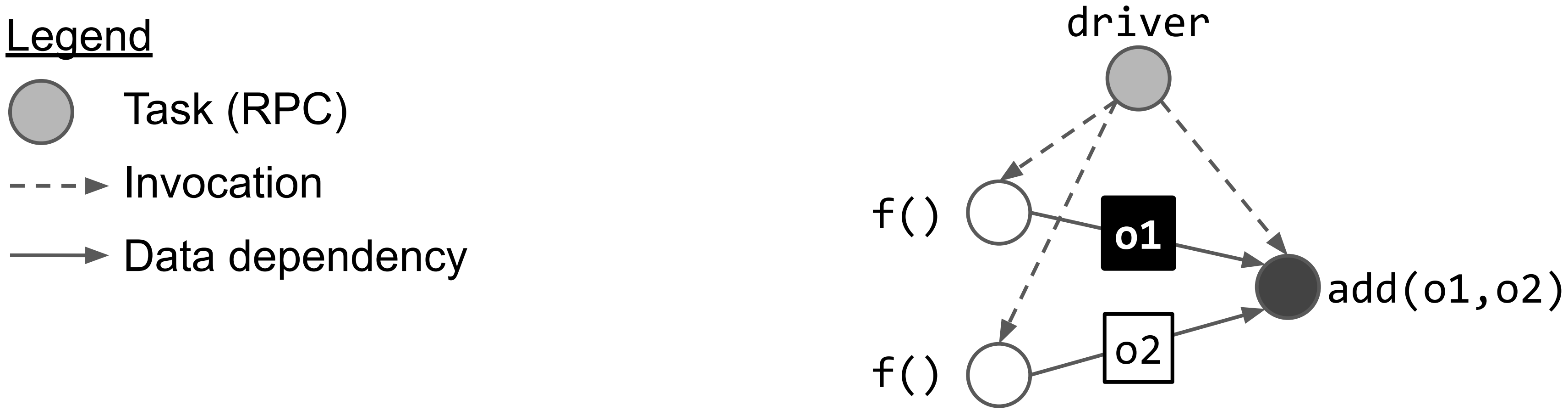
Goal: Build a distributed futures system that guarantees **fault tolerance** with **low task overhead**.

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

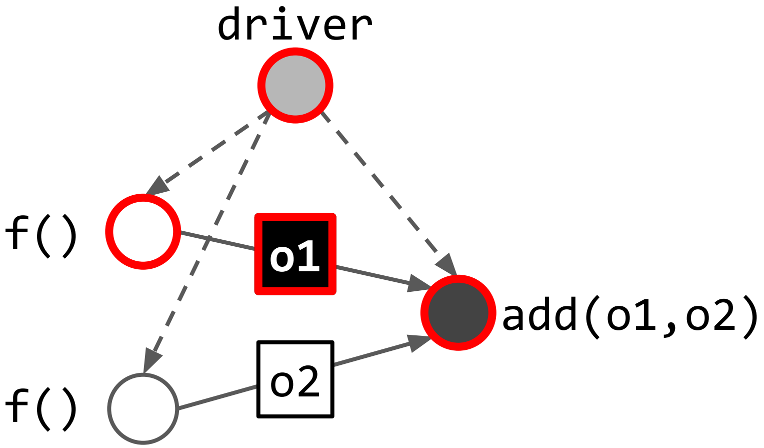
Distributed futures introduce shared state



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

Re

-
-

Re

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

-
-
- **Persistence:** Metadata should survive failures.

Existing solutions

Architecture	Coordination	Performance
Centralized master	**Master** records all metadata updates and handles all failures.	Can scale through sharding, but high overhead due to **synchronous** updates.
Leases (decentralized)	**Workers coordinate**. For example, use leases to detect a task failure.	**Asynchronous** metadata updates. Scale by adding more worker nodes.

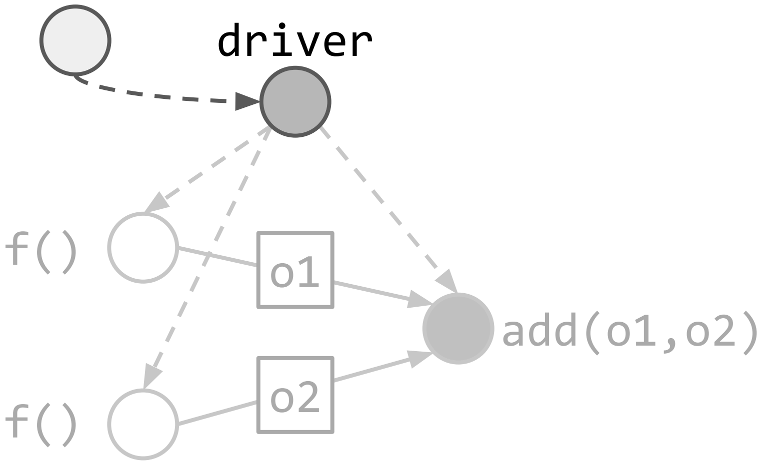
Outline

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.

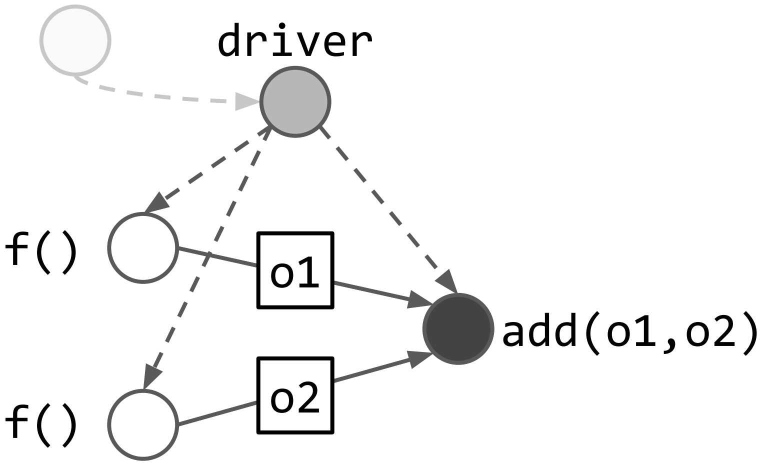
1. Task graphs are hierarchical.
2. A distributed future is often passed within the scope of the caller.



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.

Architecture	Failure handling	Performance
Ownership: The worker that calls a task *owns* the returned distributed future.	Each **worker** is a **“centralized master”** for the objects that it owns.	**No additional writes** on the critical path of task execution. Scaling through **nested function calls**.

Ownership: Challenges

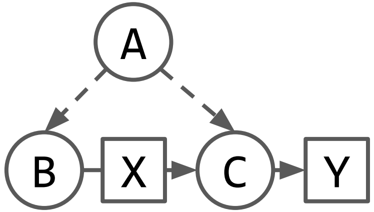
- Failure recovery
  - Recovering a lost worker
  - Recovering a lost owner
- Garbage collection and memory safety
- Handling *first-class distributed futures*, i.e. distributed futures that leave the caller's scope

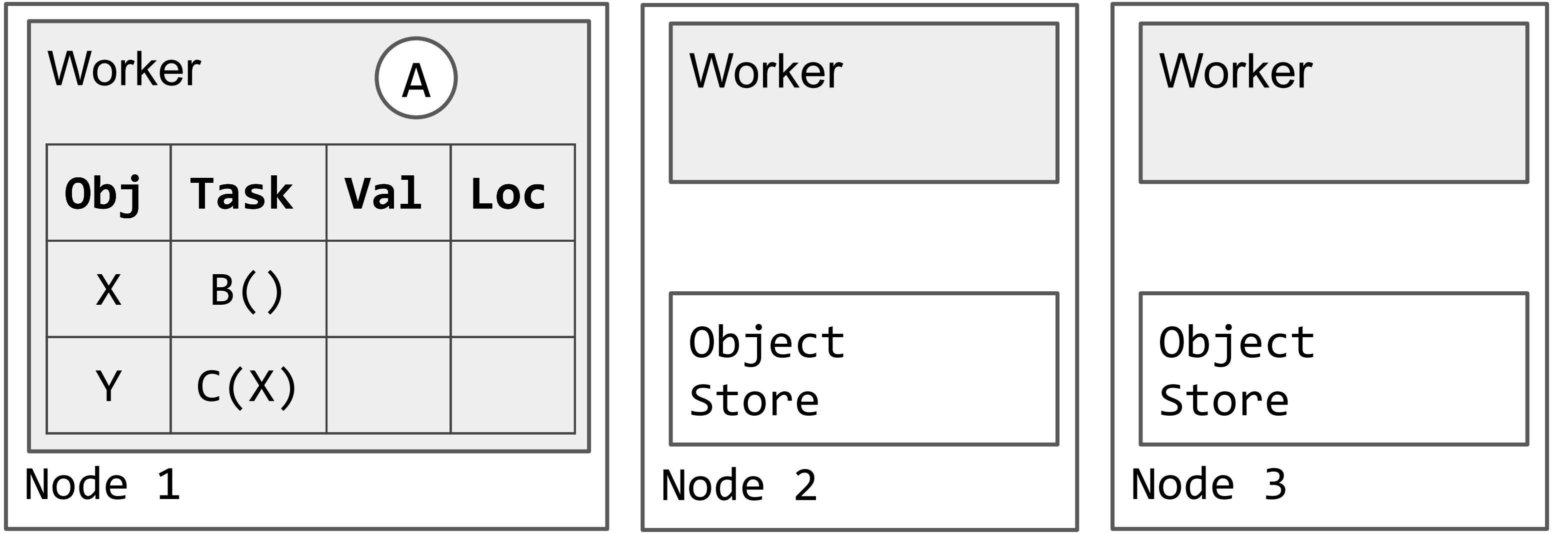
Ownership: Challenges

- **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

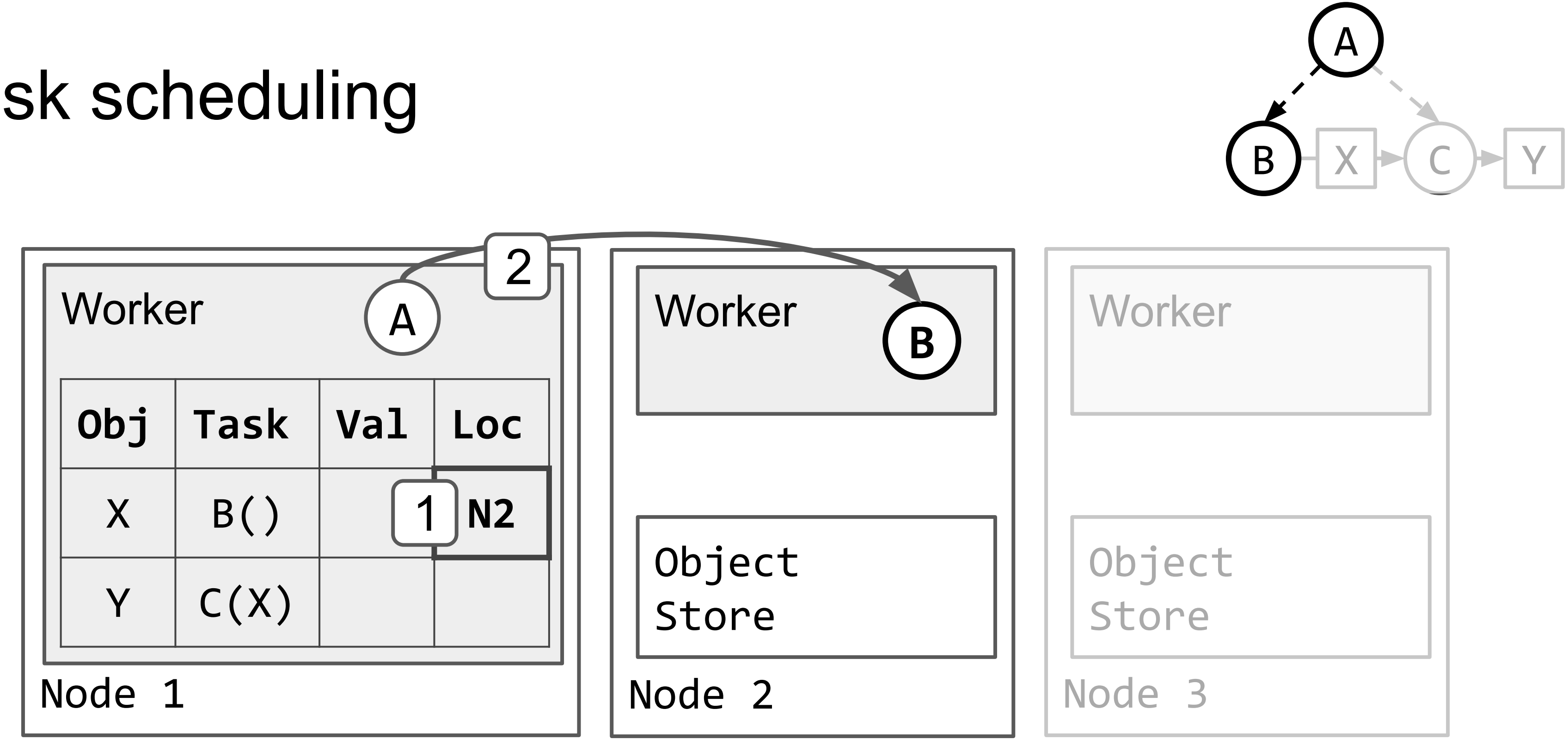
→ **Check out the paper for more details!**

Task scheduling



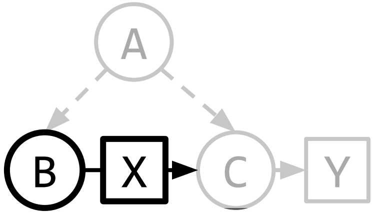


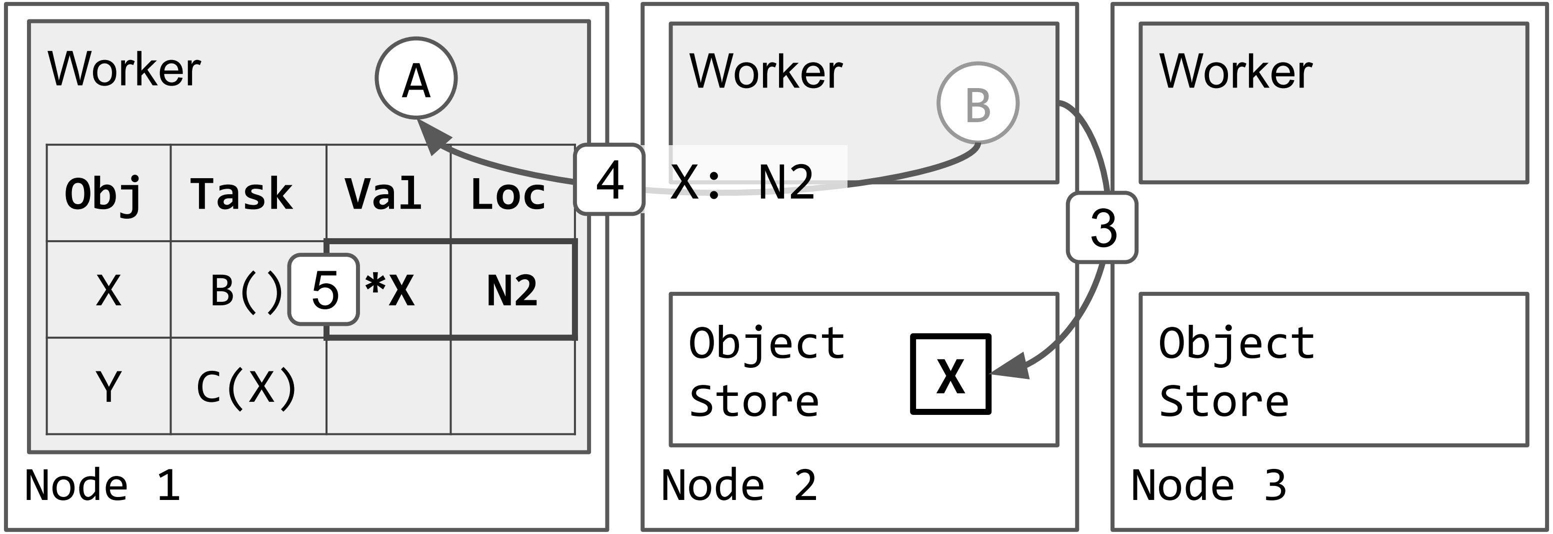
Task scheduling



A task's pending location is written locally at the owner.

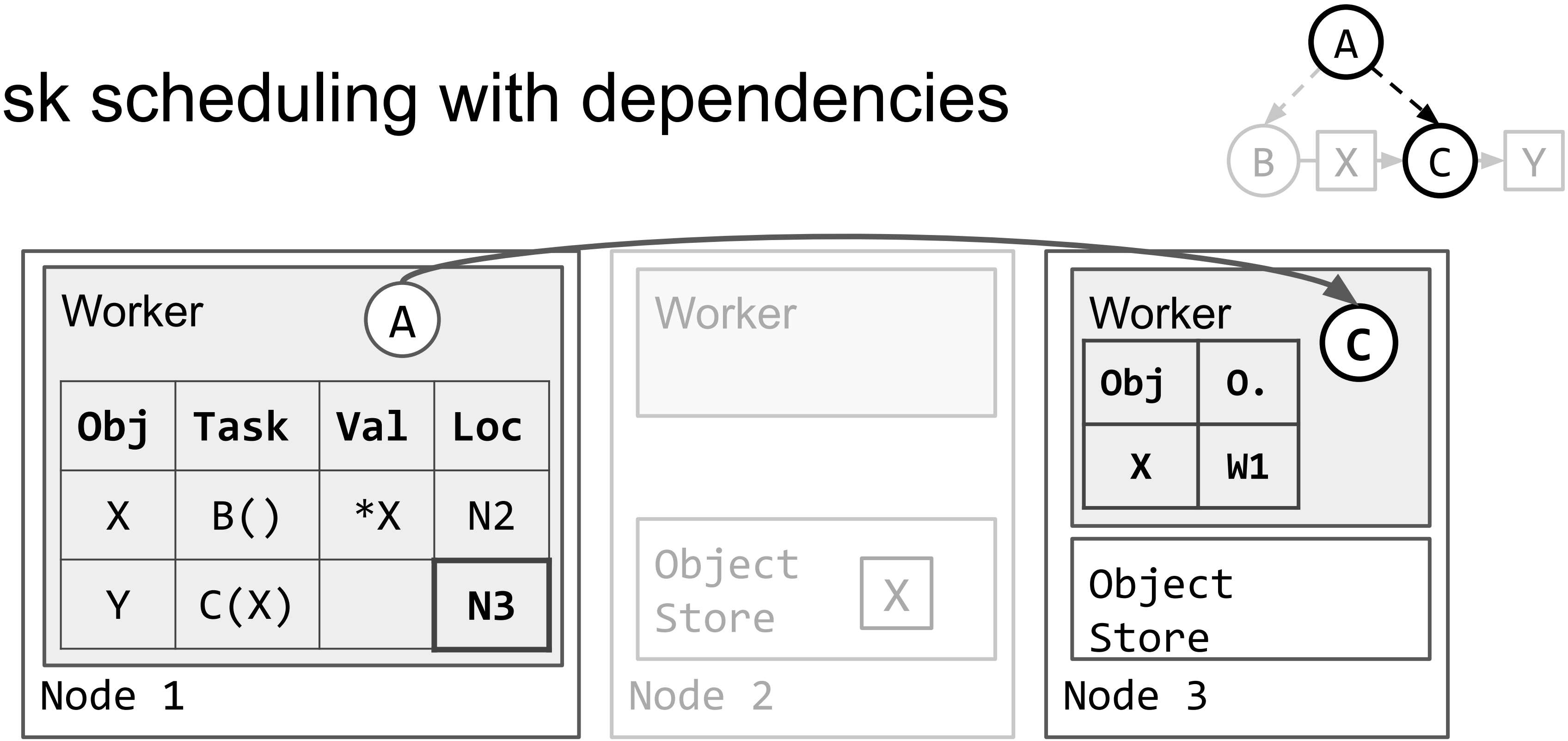
Distributed memory management



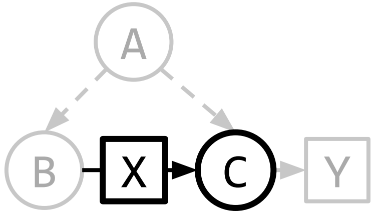


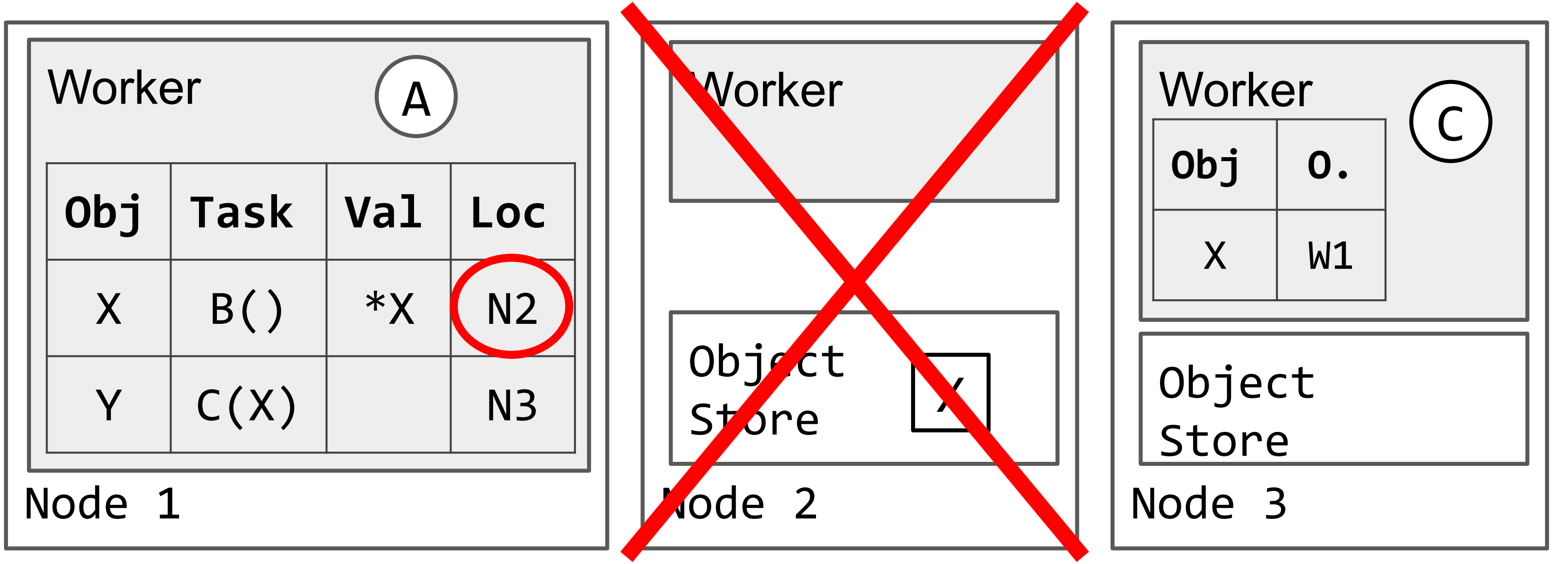
Owner tracks locations of objects stored in distributed memory.

Task scheduling with dependencies



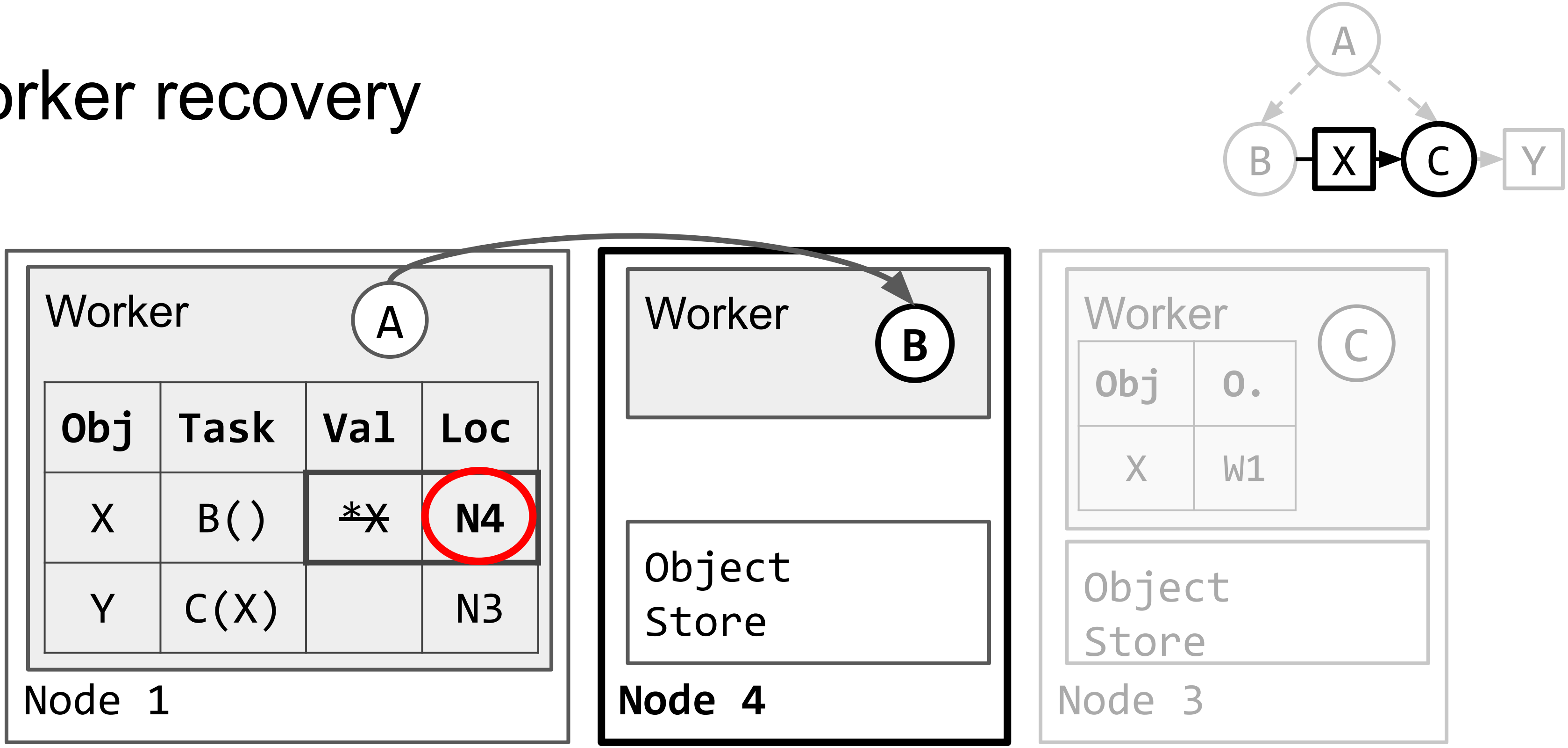
Worker failure





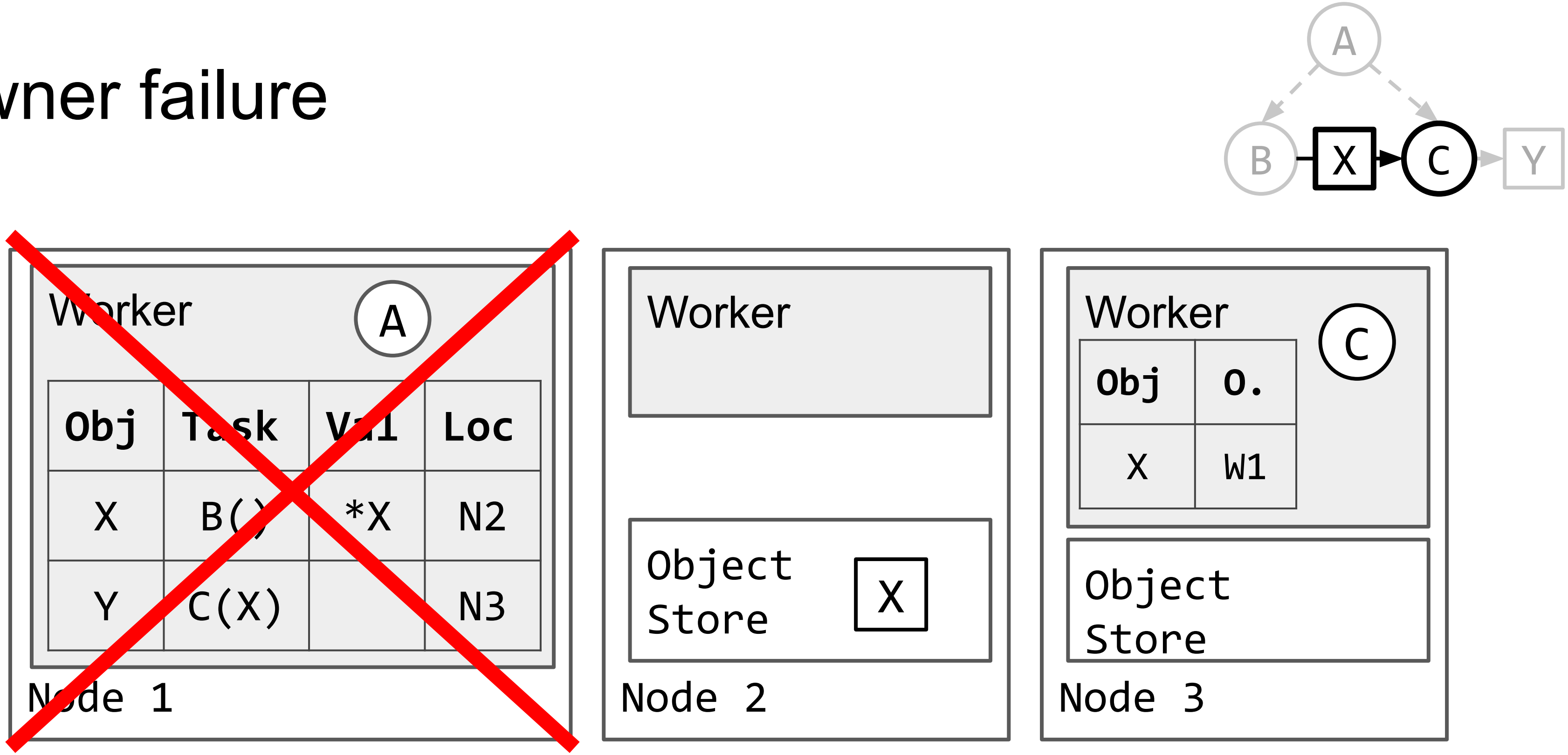
Reference holders only need to check whether the owner is alive.

Worker recovery

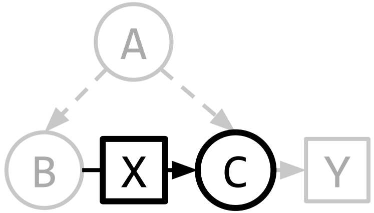


Owner coordinates lineage reconstruction.

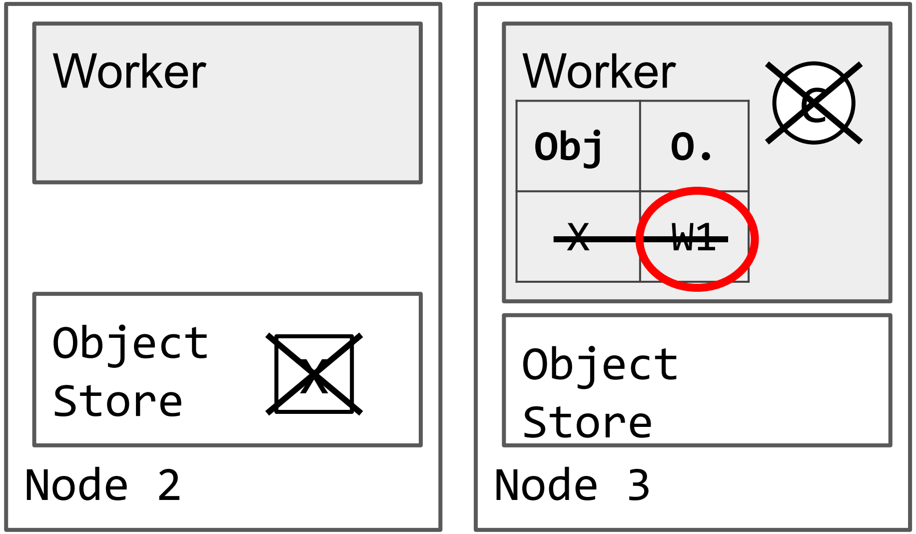
Owner failure



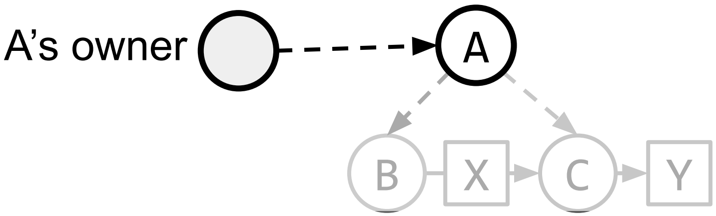
Owner recovery



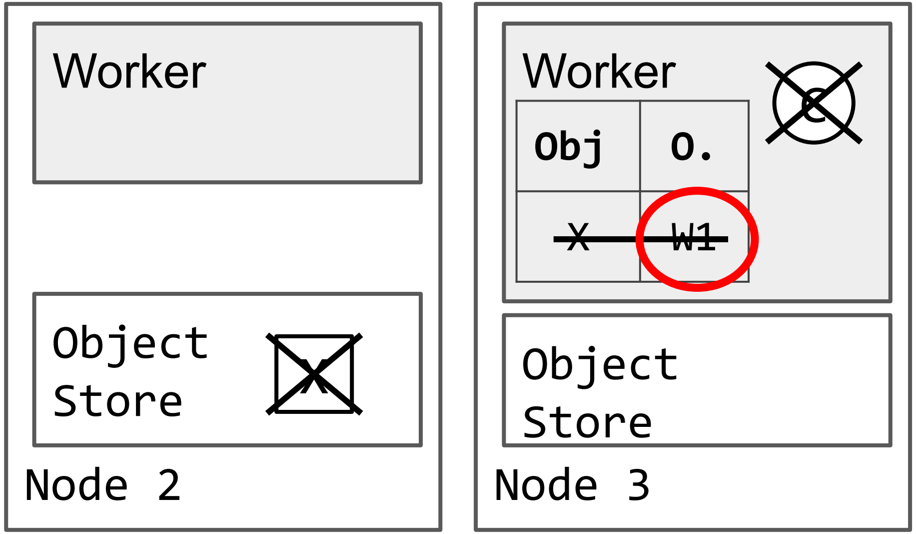
References fate-share with the object's owner.



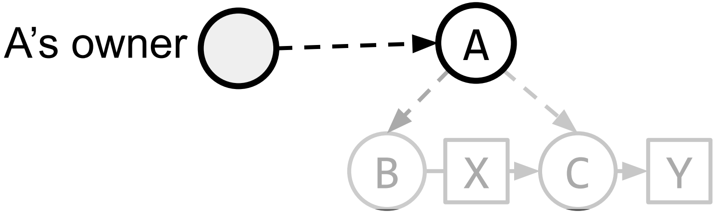
Owner recovery



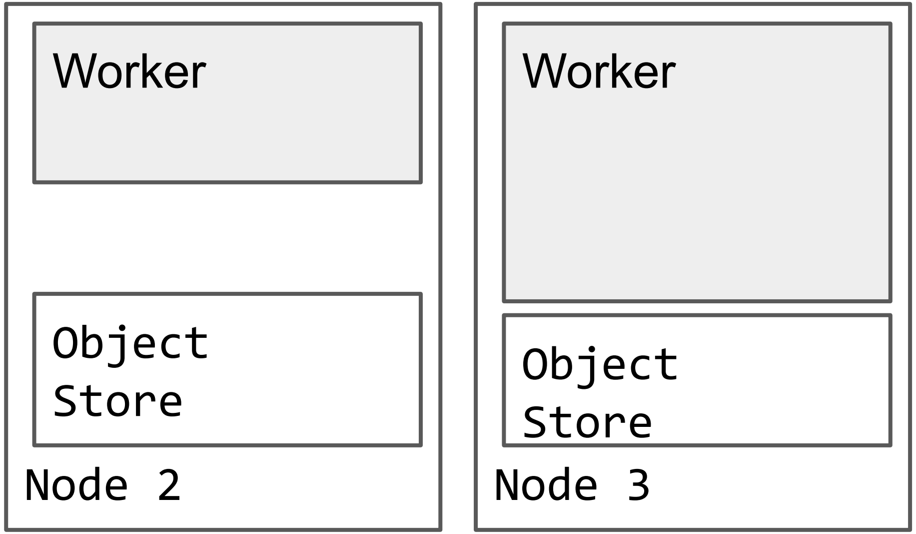
References fate-share with the object's owner.



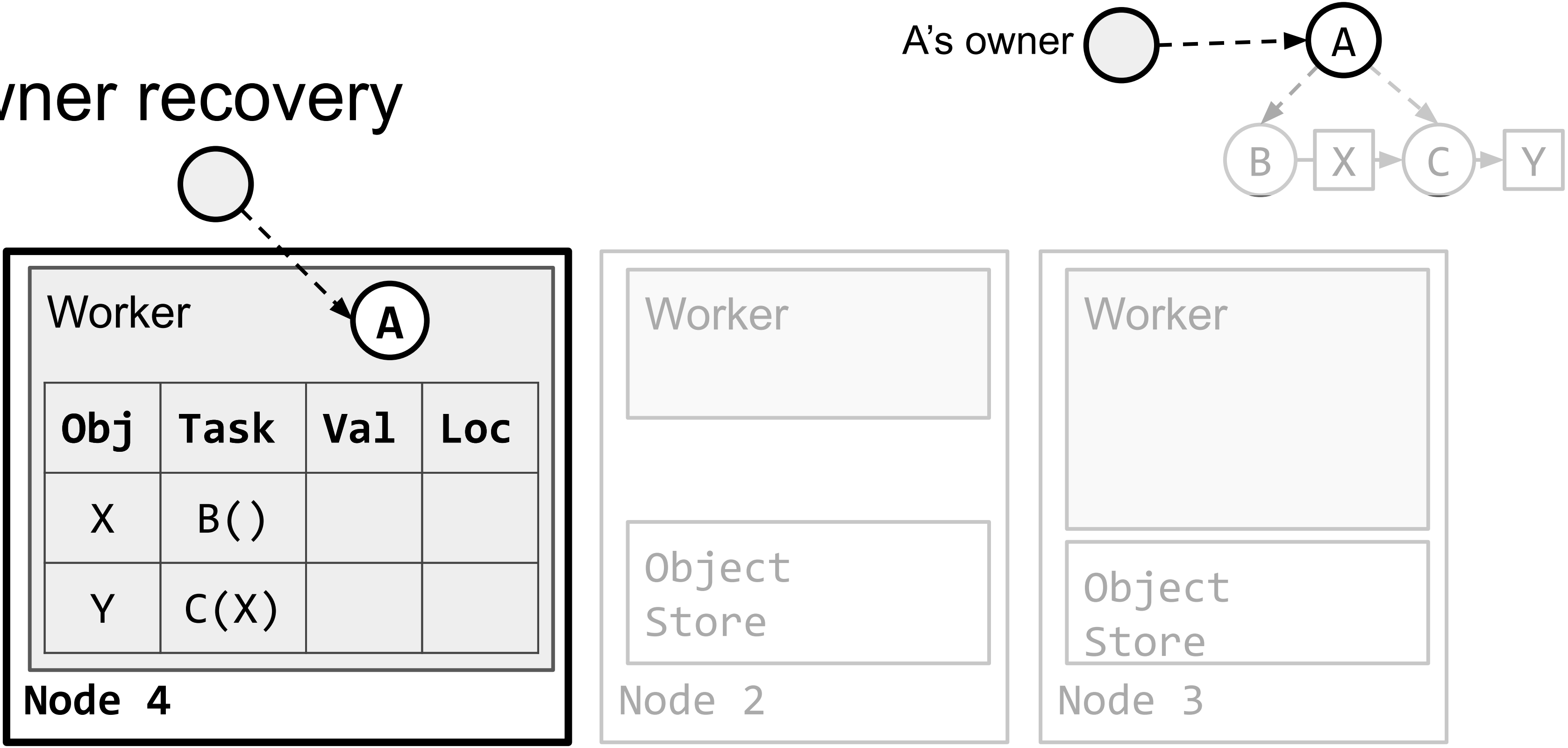
Owner recovery



References fate-share with the object's owner.



Owner recovery



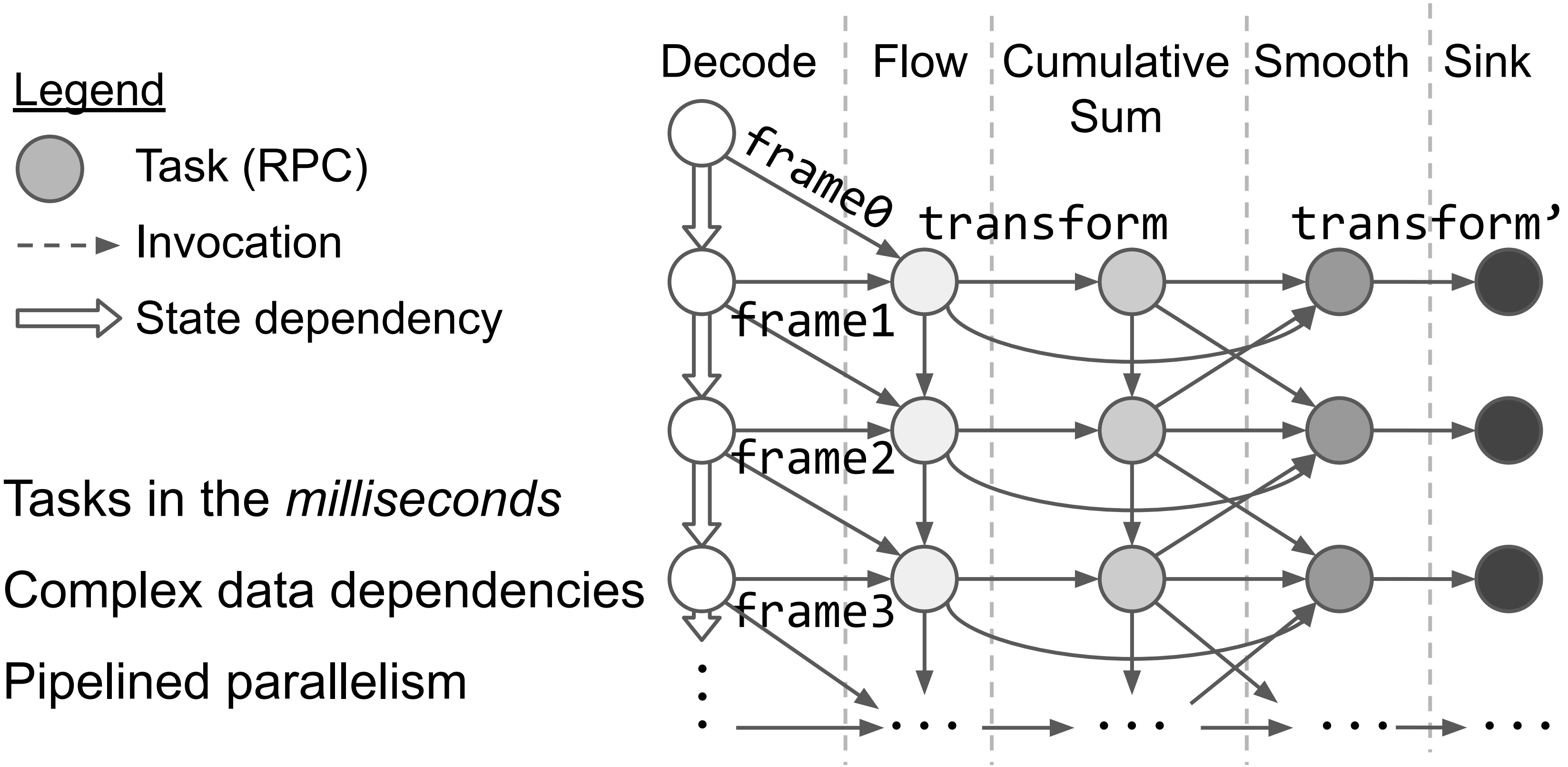
Obj	Task	Val	Loc
X	B()		
Y	C(X)		

Leveraging the application's hierarchical structure: the owner of A recovers A.

Outline

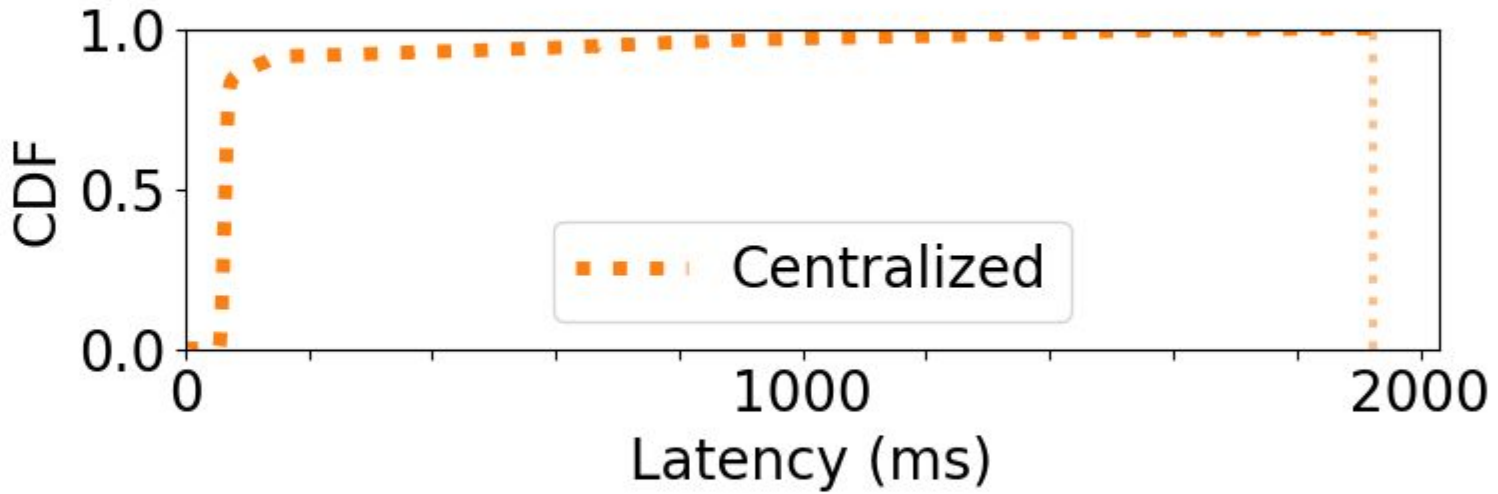
1. An overview of distributed futures
2. System requirements and challenges
3. Ownership: Achieving fault tolerance without giving up performance
4. **Evaluation** RAY

Evaluation: Online video processing



1. Tasks in the *milliseconds*
2. Complex data dependencies
3. Pipelined parallelism

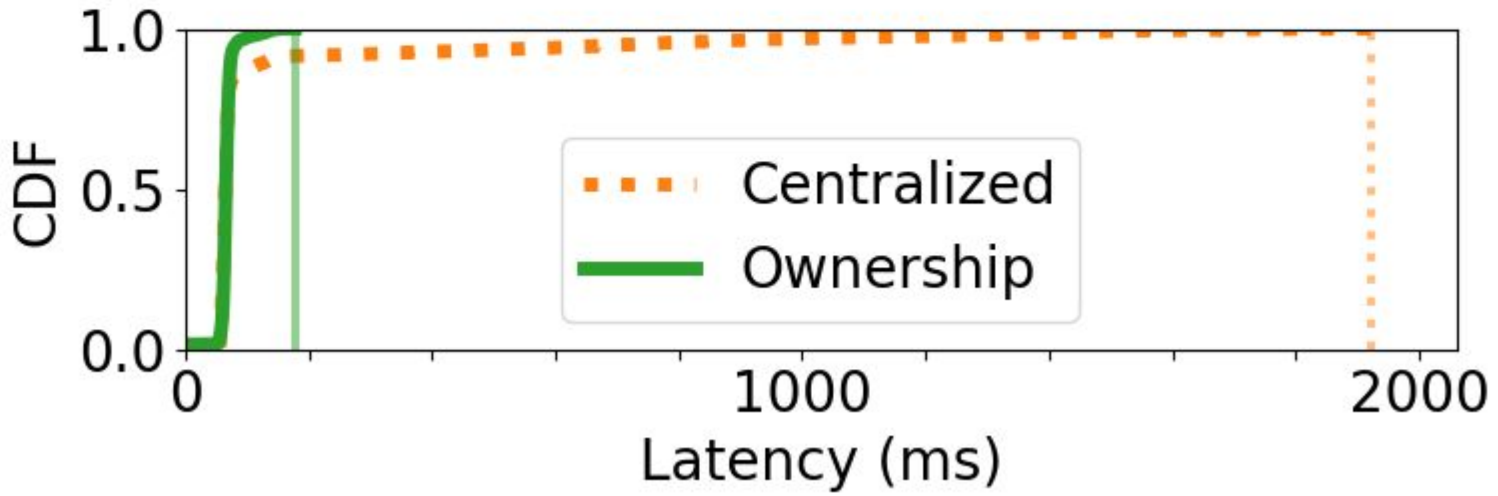
Evaluation: Online video processing (60 videos)



Centralized = Ray modified with writes to a centralized metadata store

RAY

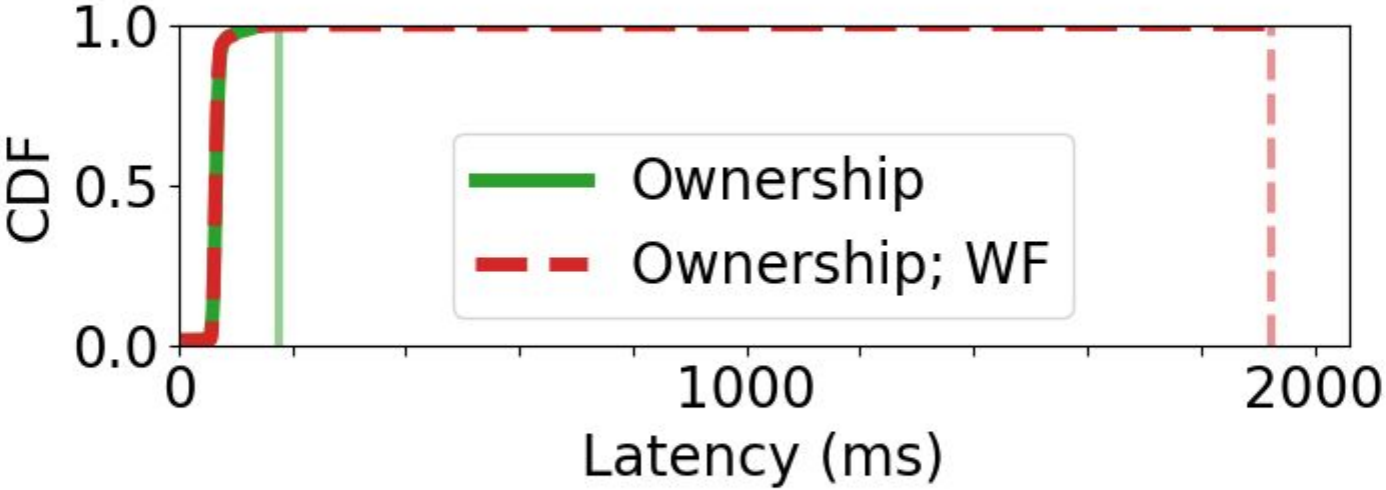
Evaluation: Online video processing (60 videos)



Latency with ownership is lower because each video has a different owner.

RAY

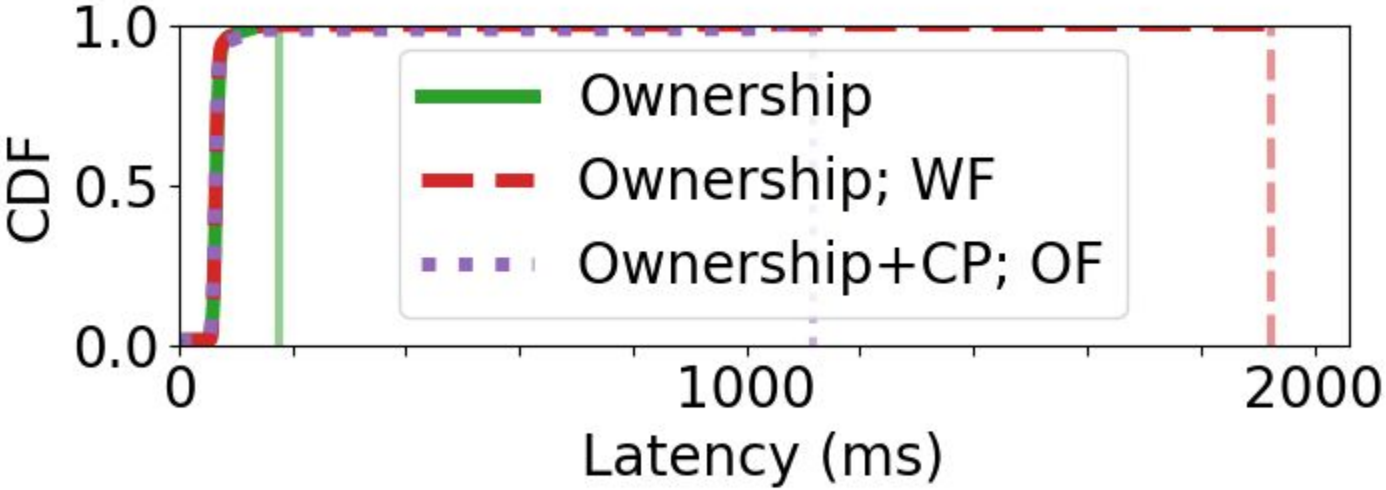
Evaluation: Online video processing with failures



Recovery when the owner is intact, with lineage reconstruction.

RAY

Evaluation: Online video processing with failures



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

Live input video

Stabilized video

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

RAY