Introduction

- Large distributed applications run across multiple machines to improve performance and scalability
    - Model serving
    - Online video processing
    - Distributed training
    - etc.
- These applications can be split into smaller tasks and actors
    - What is the difference between a task and an actor?
- A driver process launches the application and coordinates everything
    - It is essentially the “head” or the root node

- How do these tasks/actors communicate? With RPCs
    - Discuss similarities/differences between RPCs and local function calls
- Pass by value vs. pass by reference
- Synchronous vs. asynchronous RPCs
    - Parallelism
    - Overlapping compute with communication (latency hiding)
- Promises and futures

- Distributed futures are not a new idea, though previous implementations require significant coordination and overhead in sharing state between processes
    - **This coordination is necessary to ensure the application/system can recover from failures. Otherwise, little-to-no coordination would be necessary.**
    - This is fine for tasks that require significant compute
    - But this doesn’t work well for fine-grained tasks
    - Why not?
        - Because the system overheads become responsible for a much larger portion of the application runtime, as the application compute is much smaller
    - AIFM suffers from a similar problem, and thus offloads compute to a remote node

- Previous solutions:
    - Centralized master
        - Record coordination data at centralized master
            - Simple implementation, but does not scale because master becomes a centralized bottleneck
    - Leases

- Ownership with distributed futures
    - Scales horizontally
        - Essentially shard the work across all nodes instead of at the master
            - e.g., nested tasks

    - Local metadata writes at the task’s caller (which is also the object’s owner)
    - Simple failure handling
        - Each worker is essentially a “centralized master” for the objects it owns
- Ownership tracks object lifetimes with distributed reference counting
    - The system garbage collects objects with a reference count of 0
    - What about reference cycles? e.g., an actor invokes a task, which in turn invokes the actor.
        - class A:
        -   def call(self, B):
        -     self.x_ref = B.foo.remote()
        -
        -   def foo(self):
        -     return self.x_ref
- Uses lineage reconstruction to recover objects upon worker or object store failure
    - Basically, the tasks are run again to produce the objects again
    - Tasks must be idempotent
    - Only the minimal subset of tasks are rerun
    - Tasks fate-share with the owners of any objects they reference
- **Key insight: uses application semantics for better performance**
    - Brief discussion of AIFM

API

- Show the API on iPad

Applications

- Model serving
- Distributed processing
- If the students mention another workload, we should sketch out the workload graph structure

Overview

- Requirements
    - Automatic memory management
    - Failure detection
    - Failure recovery

- Automatic memory management
    - Reclaims objects via garbage collection
    - Tracks object via reference counting
- Failure detection
    - Why isn’t this easy? Shouldn’t a worker be able to tell when another work crashes?
    - Distributed futures complicate this

        - A worker doesn't necessarily know where a value it wants to load will be located
        - This is because maybe the other worker that will generate the value hasn't been scheduled yet
            - Or maybe it has, but then the scheduling decision was updated
    - Systems records location of all objects and **all tasks** (i.e., pending objects)
- Failure recovery
    - Want failure recovery to be transparent to the application
    - Need to keep metadata up to date. Metadata includes:
        - Location of each object (so it can be retrieved by people with references)
        - Whether the object is still referenced (for garbage collection)
        - Location of each pending object
        - Object lineage
    - **Existing solutions:**
        - Centralized master
        - Distributed leases
            - What are distributed leases?
            - What are their downsides? i.e., why are they not a sufficient approach here?
                - Slower to detect failure (need to wait for lease to expire)
                - Upon failure, workers need to coordinate among themselves to determine who should recover/regenerate an object

- **Solution: Ownership**
    - Essentially distributes the control plane across all the workers
    - Leverages insight into application semantics to do this efficiently
    - The task's caller is the owner of the task and the object it produces
        - Why?
            - Task owner is likely to write metadata the most, so it can do local writes
            - If object stays only in owner's scope, then garbage collection is easier and has lower overhead because there is no need to do distributed reference counting
        - A couple issues to solve though:
            - First-class futures
                - So a future may leave an owner's scope… need to account for this with reference counting
                    - Centralized reference counting doesn't scale, so need a distributed mechanism
            - Owner recovery
                - When an owner fails, what do we do about dangling references?

        - We have the object and any reference holders with the owner
            - When the owner dies, those are killed
            - System uses lineage reconstruction to regenerate the objects

- **Ownership Design**
    - Each **worker** has an **ownership table**
        - It tracks each *future* it has in this table
        - An **owner** tracks everything about the object in the table
        - A **borrower** tracks a subset of this data
    - There is a distributed task scheduler
    - There is a distributed memory layer
        - This and the scheduler will be explained more in the Ray paper this week
        - Why are objects immutable? Doesn't this reduce the utility of the system?

    - **Task scheduler**
        - Ray has a distributed scheduler (more on Wednesday)
        - An owner first requests resources from its local scheduler
        - If there are no local resources available, the scheduler has the owner contact the scheduler on a remote node for resources
        - Once resources are found, the scheduler grants the owner a lease
        - The owner updates its ownership table
        - The owner can bypass the scheduler and reuse the resources for something else if the lease is still active
    - **Memory management**
        - Objects are stored in a distributed object store
        - Small objects are passed by value (< 100 KiB) while large objects are stored in the object store
        - The primary copy on the owner is *pinned*, and other objects that are not pinned can be evicted via LRU when the system is under memory pressure
        - Objects are reclaimed when their reference count is 0
            - No tasks on the owner are using the object
            - And there are no dependent tasks that are using or borrowing the object
    - **Failure recovery**
        - When a worker (not a node!) fails, the local scheduler publishes this to other workers and nodes
        - Nodes (not workers!) exchange heartbeats, so this can detect when a node itself fails
        - The owner does lineage reconstruction
            - In other words, it scans its ownership table to determine the minimal set of things to re-run

            - You could always just re-run the task without consulting the ownership table, but this could induce extra unnecessary overhead
        - **Object recovery**
            - Basically just run the tasks again
        - **Owner recovery**
            - All reference holders fate share with the owner
            - This includes children and **ancestors** of the owner
                - An owner can pass a DFut or a SharedDFut to a child
                - An owner can also return a value to an ancestor
                - Thus, any borrower can be a child or an ancestor
            - Actor recovery is outside the scope of the paper
                - It can reuse the same mechanism
                - But local actor state must be recovered, which cannot use the technique in this paper