Ray v2 Architecture

Ray Team, October 2022

This document is public; please use "Viewing" mode to avoid accidental comments.

The goal of this document is to motivate and overview the design of the Ray distributed system (version 2.0+). It is meant as a handbook for:

- Ray users with low-level system questions
- Engineers considering Ray as a backend for new distributed applications
- Contributors to the Ray backend

This document is not meant as an introduction to Ray. For that and any further questions that arise from this document, please refer to the Getting Started Guide, the Ray GitHub repo, and the Ray Slack. You may also want to check out the list of common Ray Design Patterns. This document supersedes previous papers describing Ray; in particular, the underlying architecture has changed considerably from versions 0.7 to 0.8. Since v1.0, the core architecture has remained largely the same, but with significant extensions for applications in machine learning, online serving, and data processing.

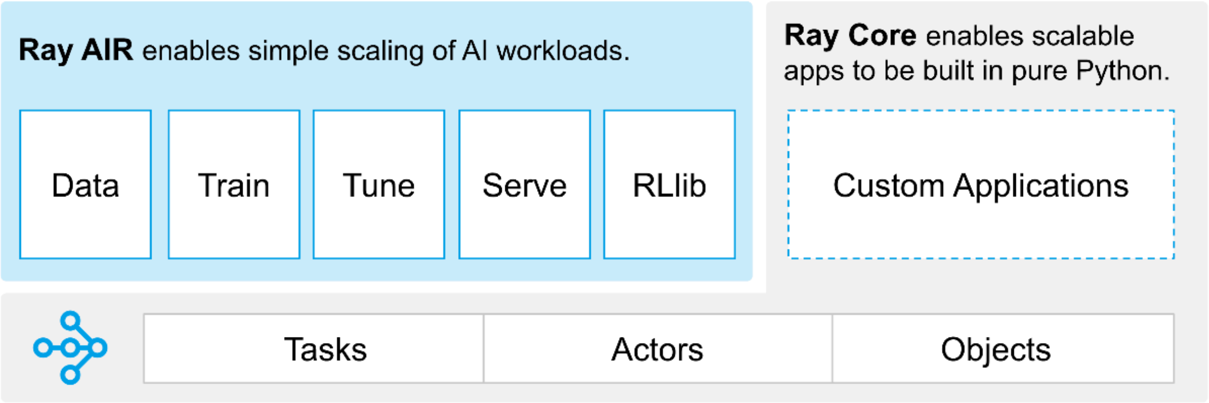
The previous whitepaper can be found here.

Overview

API philosophy

Ray aims to provide a universal API for distributed computing. A core part of achieving this goal is to provide **simple but general programming abstractions**, letting the system do all the hard work. This philosophy is what makes it possible as a developer to use Ray with existing Python libraries and systems.

A Ray programmer expresses their logic with a handful of Python primitives, while the system manages physical execution concerns such as parallelism and distributed memory management. A Ray user thinks about cluster management in terms of resources, while the system manages scheduling and autoscaling based on those resource requests.


Ray AIR enables simple scaling of AI workloads.
Data
Train
Tune
Serve
RLlib
Ray Core enables scalable apps to be built in pure Python.
Custom Applications
Tasks
Actors
Objects

Ray provides a universal API of tasks, actors, and objects for building distributed applications.

Some applications may require a different set of system-level tradeoffs that cannot be expressed through the core set of abstractions. Thus, a second goal in Ray's API is to allow the application **fine-grained control over system behavior**. This is accomplished through a set of configurable parameters that can be used to modify system behaviors such as task placement, fault handling, and application lifetime.

System scope

Ray seeks to enable the development and composition of distributed applications and libraries *in general*. Concretely, this includes coarse-grained elastic workloads (i.e., types of serverless computing), machine learning training (e.g., Ray AIR), online serving (e.g., Ray Serve), data processing (e.g., Ray Datasets, Modin, Dask-on-Ray), and ad-hoc computation (e.g., parallelizing Python apps, gluing together different distributed frameworks).

Ray's API enables developers to easily compose multiple libraries within a single distributed application. For example, Ray tasks and actors may call into or be called from distributed training (e.g., torch.distributed) or online serving workloads also running in Ray. This is how the Ray AI Runtime (AIR) is built, by composing together other Ray libraries under the hood. In this sense, Ray makes for an excellent "distributed glue" system, because its API is general and performant enough to serve as the interface between many different workload types.

System design goals

The core principles that drive Ray's architecture are **API simplicity** and **generality**, while the core system goals are **performance** (low overhead and horizontal scalability) and **reliability**. At times, we are willing to sacrifice other desirable goals such as *architectural simplicity* in return for these core goals. For example, Ray includes components such as distributed reference counting and distributed memory, which add to architectural complexity, but are needed for performance and reliability.

For performance, Ray is built on top of gRPC and can in many cases match or exceed the performance of naive use of gRPC. Compared to gRPC alone, Ray makes it simpler for an application to leverage parallel and distributed execution, distributed memory sharing (via a shared memory object store), and dynamic creation of lightweight services (i.e. actors).

For reliability, Ray's internal protocols are designed to ensure correctness during failures while adding low overhead to the common case. Ray implements a distributed reference counting protocol to ensure memory safety and provides various options to recover from failures.

Since a Ray user thinks about expressing their computation in terms of resources instead of machines, Ray applications can transparently scale from a laptop to a cluster without any code changes. Ray's distributed scheduler and object manager are designed to enable this seamless scaling, with low overheads.

Related systems

The following table compares Ray to several related system categories. Note that we omit higher-level library comparisons (e.g., RLlib, Tune, RaySGD, Serve, Modin, Dask-on-Ray, MARS-on-Ray); such comparisons are outside the scope of this document, which focuses on Ray core only. You may also refer to the full list of community libraries on Ray.

Cluster Orchestrators	Ray can run on top of cluster orchestrators like Kubernetes or SLURM to offer lighter weight, language integrated primitives, i.e., tasks and actors instead of containers and services.
Parallelization Frameworks	Compared to Python parallelization frameworks such as multiprocessing or Celery, Ray offers a more general, higher-performance API. The Ray system also explicitly supports memory sharing.
Data Processing Frameworks	Compared to data processing frameworks such as Spark, Flink, MARS, or Dask, Ray offers a lower-level and narrower API. This makes the API more flexible and more suited as a "distributed glue" framework. On the other hand, Ray has no inherent understanding of data schemas, relational tables, or streaming dataflow; such functionality is provided through libraries only (e.g., Modin, Dask-on-Ray, MARS-on-Ray).
Actor Frameworks	Unlike specialized actor frameworks such as Erlang and Akka, Ray integrates with existing programming languages, enabling cross language operation and the use of language native libraries. The Ray

	system also transparently manages parallelism of stateless computation and explicitly supports memory sharing between actors.
HPC Systems	Many HPC systems expose a message-passing interface, which is a lower-level interface than tasks and actors. This can allow the application greater flexibility, but potentially at the cost of developer effort. Many of these systems and libraries (e.g., NCCL, MPI) also offer optimized collective communication primitives (e.g., allreduce). Ray apps can leverage such primitives by initializing communication groups between sets of Ray actors (e.g, as RaySGD does with torch distributed).

New in 2.0 whitepaper

Since the 1.x whitepaper:

- The Global Control Store is now known as the Global Control **Service** (GCS) and features a completely updated design to simplify coordination and reliability.
- The distributed scheduler offers expanded functionality and flexibility, including scheduling policies and placement groups.
- General improvements in reliability and fault tolerance, including object reconstruction to recover from node failures and GCS fault tolerance.
- Expanded toolset for managing and interacting with Ray clusters, featuring job submission, KubeRay (Ray on Kubernetes), and application observability.

Architecture Overview

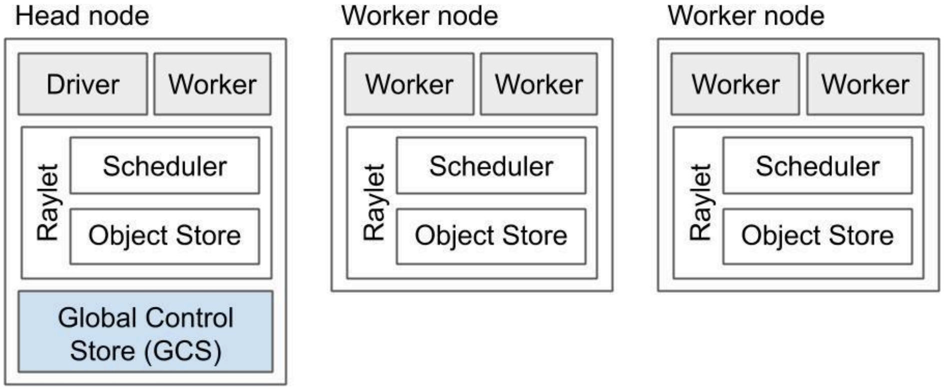
Application concepts

- *Task* - A remote function invocation. This is a single function invocation that executes on a process different from the caller, and potentially on a different machine. A task can be stateless (a `@ray.remote` function) or stateful (a method of a `@ray.remote` class - see *Actor* below). A task is executed asynchronously with the caller: the `.remote()` call immediately returns one or more `ObjectRefs` (futures) that can be used to retrieve the return value(s).
- *Object* - An application value. These are values that are returned by a task or created through `ray.put`. Objects are **immutable**: they cannot be modified once created. A worker can refer to an object using an `*ObjectRef*`.
- *Actor* - a stateful worker process (an instance of a `@ray.remote` class). Actor tasks must be submitted with a *handle*, or a Python reference to a specific instance of an actor, and can modify the actor's internal state during execution.
- *Driver* - The program root, or the "main" program. This is the code that runs `ray.init()`.

- *Job* - The collection of tasks, objects, and actors originating (recursively) from the same driver, and their runtime environment. There is a 1:1 mapping between drivers and jobs.

Design

Components



A Ray cluster.

A Ray cluster consists of one or more worker **nodes**, each of which consists of the following physical processes:

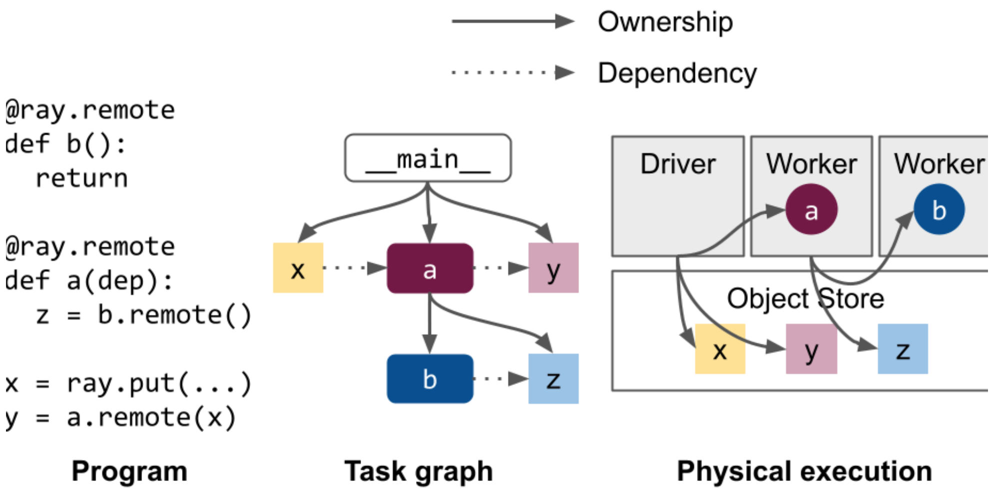
1. One or more **worker processes**, responsible for task submission and execution. A worker process is either stateless (can be reused to execute any @ray.remote function) or an actor (can only execute methods according to its @ray.remote class). Each worker process is associated with a specific job. The default number of initial workers is equal to the number of CPUs on the machine. Each worker stores:
    a. An **ownership table**. System metadata for the objects to which the worker has a reference, e.g., to store ref counts and object locations.
    b. An ***in-process store***, used to store small objects.
2. A **raylet**. The raylet manages shared resources on each node. Unlike worker processes, the raylet is shared among all concurrently running jobs. The raylet has two main components, run on separate threads:
    a. A **scheduler**. Responsible for resource management, task placement, and fulfilling task arguments that are stored in the distributed object store. The individual schedulers in a cluster comprise the Ray **distributed scheduler**.
    b. A **shared-memory object store (also known as the Plasma Object Store)**. Responsible for storing, transferring, and spilling large objects. The individual object stores in a cluster comprise the Ray **distributed object store**.

Each worker process and raylet is assigned a unique 28-byte identifier and an IP address and port. The same address and port can be reused by subsequent components (e.g., if a previous worker process dies), but the unique IDs are never reused (i.e., they are tombstoned upon process death). Worker processes fate-share with their local raylet process.

One of the worker nodes is designated as the **head node**. In addition to the above processes, the head node also hosts:

1. The **Global Control Service (GCS)**. The GCS is a server that manages cluster-level metadata, such as the locations of actors, stored as key-value pairs that may be cached locally by workers. The GCS also manages a handful of cluster-level operations, including scheduling for placement groups and actors and determining cluster node membership. In general, the GCS manages metadata that is less frequently accessed but likely to be used by most or all workers in the cluster. This is to ensure that GCS performance is not critical to application performance. GCS fault tolerance is new in Ray 2.0, allowing the GCS to run on any and multiple nodes, instead of a designated head node.
2. The **driver process(es)**. A driver is a special worker process that executes the top-level application (e.g., `__main__` in Python). It can submit tasks, but cannot execute any itself. Note that driver processes can run on any node, but usually run on the head node by default.
3. Other **cluster-level services** that handle job submission, autoscaling, etc.

Ownership



```
@ray.remote
def b():
  return

@ray.remote
def a(dep):
  z = b.remote()

x = ray.put(...)
y = a.remote(x)
```

Most of the system metadata is managed according to a decentralized concept called *ownership.* This concept means that each `ObjectRef` in the application will be managed by a single worker process. This worker, or the “owner” is responsible for ensuring execution of the task that creates the value and facilitating the resolution of an `ObjectRef` to its underlying value.

There are two ways to create an ObjectRef. In both cases, the owner is the worker process of x_ref that calls this code.

1. x_ref = f.remote()
2. x_ref = ray.put()

In other words, the owner is the worker that generates the initial ObjectRef. Note that this may be a different worker from the one that creates the **value** of the ObjectRef. If, for example, the ObjectRef is returned by a task, then the value will be created by a remote worker.

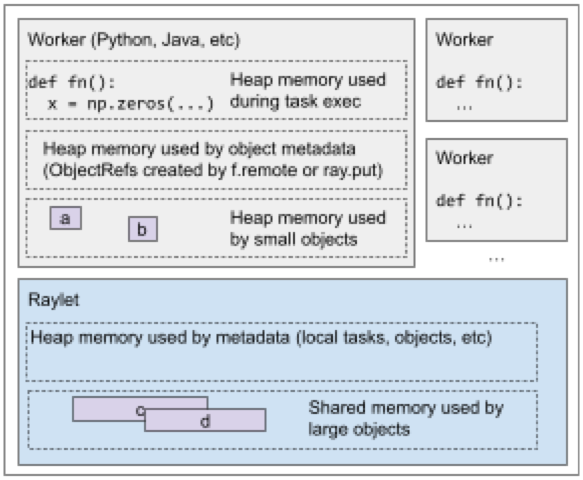
Ownership has the following benefits (compared to the more centralized design used in Ray versions <0.8):

1. Low task latency (~1 RTT, <200us). Frequently accessed system metadata is local to the process that must update it.
2. High throughput (~10k tasks/s per client; linear scaling to millions of tasks/s in a cluster), as system metadata is naturally distributed over multiple worker processes through nested remote function calls.
3. Simplified architecture. The owner centralizes logic needed to safely garbage collect objects and system metadata.
4. Improved reliability. Worker failures can be isolated from one another based on the application structure, e.g., the failure of one remote call will not affect another.

Some of the trade-offs that come with ownership are:

1. To resolve an `ObjectRef`, the object's owner must be **reachable**. This means that an object will fate-share with its owner. See Object failures and Object spilling for more information about object recovery and persistence.
2. Ownership currently cannot be transferred.

Memory model


Worker (Python, Java, etc)
def fn():
x = np.zeros(...)
Heap memory used during task exec
Heap memory used by object metadata (ObjectRefs created by f.remote or ray.put)
a
b
Heap memory used by small objects
Worker
def fn():
...
Worker
def fn():
...
...
Raylet
Heap memory used by metadata (local tasks, objects, etc)
c
d
Shared memory used by large objects

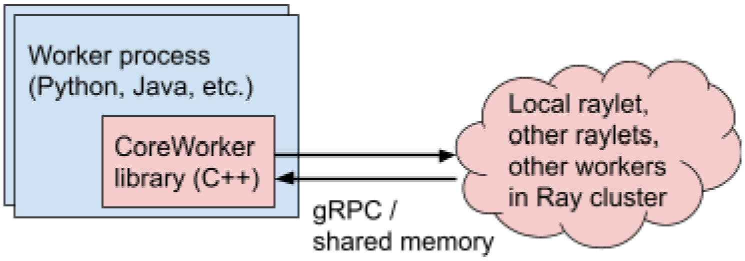
Types of memory used for a typical Ray node. The GCS (not shown) contains cluster-level metadata such as for nodes and actors.

Ray may use memory in the following ways:

1. **Heap memory used by Ray workers during task or actor execution.** Ray workers execute user-defined code when executing tasks or actors. Because Ray tasks and actors usually run in parallel, up to the number of cores, the application developer should be cognizant of each task's individual heap memory usage. If heap memory pressure is too high, Ray will attempt to kill a memory-hungry worker first, to protect the system-level state in the object store and other system-level processes.
2. **Shared memory used by large Ray objects** (values created by `ray.put()` or returned by a Ray task). When a worker calls `ray.put()` or returns from a task, it copies the provided values into Ray's shared memory object store. Ray will then make these objects available throughout the cluster, attempt to recover them in case of a failure, spill them if the object store exceeds its configured capacity, and garbage-collect them once all ObjectRefs have gone out of scope. For values that can be zero-copy deserialized, passing the ObjectRef to `ray.get` or as a task argument will return a direct pointer to the shared memory buffer to the worker. All other values will be deserialized onto the receiver worker's heap memory.
3. **Heap memory used by small Ray objects** (returned by a Ray task). If an object is small enough (default 100KB), Ray will store the values directly in the owner's "in-memory" object store instead of the Raylet shared memory object store. Any workers that read the object (e.g., through `ray.get`) will copy the value directly into their own heap memory. Ray also automatically garbage-collects these objects through the same protocol as for large objects.
4. **Heap memory used by Ray metadata.** This is memory allocated by Ray to manage metadata for the application. The majority of the metadata is in the form of a task specification or metadata about an object (such as the reference count). As of Ray v2.0, the total metadata overhead is expected to be a few KB per ObjectRef that is still in scope. Here is a brief summary of the system-level processes and their expected memory footprint scale:
    a. GCS: # total actors, # total nodes, # total placement groups
    b. Raylet: # tasks queued locally, # object arguments of these tasks, # objects stored in local shared memory or local disk
    c. Worker: # tasks submitted that are still pending or that may get re-executed through lineage reconstruction, # owned objects, # objects in scope at the language frontend

Language Runtime

All Ray core components are implemented in C++. Ray supports Python, Java, and (experimental) C++ frontends via a common embedded C++ library called the "core worker." This library implements the ownership table, in-process store, and manages gRPC communication with other workers and raylets. Since the library is implemented in C++, all language runtimes share a common high-performance implementation of the Ray worker protocol.



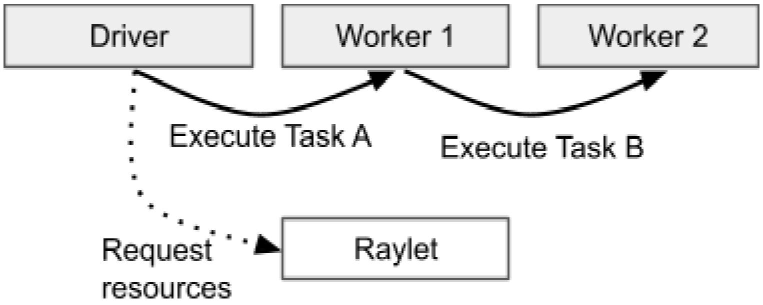
Ray workers interact with other Ray processes through the CoreWorker library.

Code references:

- Core worker source code: src/ray/core_worker/core_worker.h. This code is the backbone for the various protocols involved in task dispatch, actor task dispatch, the in-process store, and memory management.
- Language bindings for Python: python/ray/includes/libcoreworker.pxd
- Language bindings for Java: src/ray/core_worker/lib/java

Lifetime of a Task

The owner is responsible for ensuring execution of a submitted task and facilitating the resolution of the returned `ObjectRef` to its underlying value.



The process that submits a task is considered to be the owner of the result and is responsible for acquiring resources from a raylet to execute the task. Here, the driver owns the result of `A`, and `Worker 1` owns the result of `B`.

The owner can pass normal Python objects as task arguments. If a task argument's value is small, it is copied directly from the owner's in-process object store into the task specification, where it can be referenced by the executing worker.

If a task's argument is large, the owner first calls `ray.put()` on the object under the hood, then passes the ObjectRef as the task argument. Note that Ray objects are **not** automatically memoized or deduplicated; if the same large Python object is passed to two different tasks, it will result in two separate `ray.put()` calls and two separate objects. This is why it is recommended to call `ray.put()` explicitly if the same object needs to be passed to multiple tasks.

The owner can also pass other ObjectRefs as task arguments. When the task is submitted, the owner waits for any ObjectRef arguments to become available. Note that the dependencies need not be local; the owner considers the dependencies to be ready as soon as they are available anywhere in the cluster. If the ObjectRef's physical value is small, then the owner copies the value directly into the task specification, similar to small Python values. Otherwise, the owner attaches the ObjectRef metadata to the task specification, and the task executor must resolve the ObjectRef to the physical value before executing the task. This is to avoid having to transfer large arguments to the task caller.

Once all task dependencies are ready, the owner requests resources from the distributed scheduler to execute the task. The distributed scheduler attempts to acquire the resources and fetch any ObjectRef arguments in the task specification to the local node, through distributed memory. Once both the resources and arguments are available, the scheduler grants the request and responds with the address of a worker that is now *leased* to the owner.

The owner schedules the task by sending the task specification over gRPC to the leased worker. After executing the task, the worker must store the return values. If the return values are small[1], the worker returns the values inline directly to the owner, which copies them to its in-process object store. If the return values are large, the worker stores the objects in its local shared memory store and replies to the owner indicating that the objects are now in distributed memory. Similar to passing ObjectRefs as task arguments, this allows the owner to refer to the return values without having to fetch them to its local node.

When a Ray task is first called, its definition is pickled and then stored in the GCS. Later on the leased worker will fetch the pickled function definition and unpickle it to run the task.

Tasks can end in an error. Ray distinguishes between two types of task errors:

1. Application-level. This is any scenario where the worker process is alive, but the task ends in an error. For example, a task that throws an `IndexError` in Python.
2. System-level. This is any scenario where the worker process dies unexpectedly. For example, a process that segfaults, or if the worker's local raylet dies.

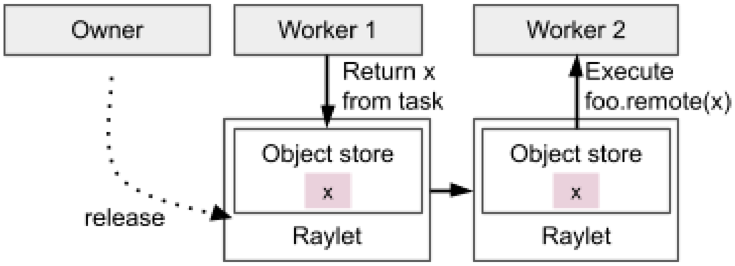
Tasks that fail due to application-level errors by default are not automatically retried. The exception is caught and stored as the return value of the task. In 2.0, users can pass a whitelist of application-level exceptions that may be automatically retried by Ray. Tasks that fail due to system-level errors may be automatically retried up to a specified number of attempts.

[1] Less than 100KiB by default.

Code references:

- Backbone for each worker: src/ray/core_worker/core_worker.cc
- Task spec definition: src/ray/common/task/task_spec.h
- Task spec protobuf definition: src/ray/protobuf/common.proto
- Caller code for requesting a worker lease and sending the task to the leased worker: src/ray/core_worker/transport/direct_task_transport.cc
- Local dependency resolution, before a worker lease is requested: src/ray/core_worker/transport/dependency_resolver.cc
- Manager for all called tasks that are still pending: src/ray/core_worker/task_manager.cc -
- Runs on Python workers to fetch function definitions: python/ray/_private/function_manager.py

Lifetime of an Object



Distributed memory management in Ray. Workers can create and get objects. The owner is responsible for determining when the object is safe to release.

An object is an immutable value that can be stored and referred to from anywhere in the Ray cluster. The owner of an object is the worker that created the initial `ObjectRef`, by submitting the creating task or calling `ray.put`. The owner manages the lifetime of the object. Ray guarantees that if the owner is alive, the object may eventually be resolved to its value (or an error is thrown in the case of worker failure). If the owner is dead, an attempt to get the object's value will throw an exception, even if there are still physical copies of the object.

Each worker stores a ref count for the objects that it owns. See Reference Counting for more information on how references are tracked. References are only counted during these operations:

1. Passing an `ObjectRef` or an object that contains an `ObjectRef` as an argument to a task.
2. Returning an `ObjectRef` or an object that contains an `ObjectRef` from a task.

Objects can be stored in the owner's in-process memory store or in the distributed object store. The in-process memory store is allocated on the owner's heap and does not enforce a capacity limit. This is because Ray only stores small objects in this store; an excessive number of small objects in scope may cause the owner process to be killed from out-of-memory. Objects stored in the distributed object store are first stored in shared memory. The shared memory object store enforces a user-configurable capacity limit (default 30% of machine RAM) and spills objects to local disk on reaching capacity. This decision is meant to reduce the memory footprint and resolution time for each object.

When there are no failures, the owner guarantees that at least one copy of an object will eventually become available as long as the object is still in scope (nonzero ref count). See Memory Management for more details.

There are two ways to resolve an `ObjectRef` to its value:

1. Calling `ray.get` on an `ObjectRef(s)`.
2. Passing an `ObjectRef` as an argument[2] to a task. The executing worker will resolve the `ObjectRef`s and replace the task arguments with the resolved values.

When an object is small, it can be resolved by retrieving it directly from the owner's in-process store. Large objects are stored in the distributed object store and must be resolved with a distributed protocol. See Object Resolution for more details.

When there are no system-level failures, resolution is guaranteed to eventually succeed but may throw an application-level exception. If there are failures, resolution may throw a system-level exception (e.g., ray.exceptions.WorkerCrashedError) but will never hang. An object can fail if it is stored in distributed memory and all copies of the object are lost through raylet failure(s). Ray attempts to automatically recover such lost objects through reconstruction. An object also fails if its owner process dies.

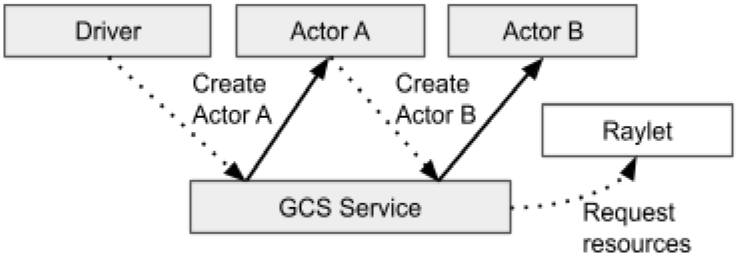
Code references:

- In-process object store, for small objects: src/ray/core_worker/store_provider/memory_store/memory_store.cc
- Worker client to shared-memory object store, for accessing large objects: src/ray/core_worker/store_provider/plasma_store_provider.cc
- Reference counting class on each worker: src/ray/core_worker/reference_count.cc
- Manager for distributed object transfers: src/ray/object_manager/object_manager.cc
- Manager for local objects that are "primary" copies or that need to be spilled: src/ray/raylet/local_object_manager.cc

[2] Note that if the `ObjectRef` is contained within a data structure (e.g., Python list), or otherwise serialized within an argument, it will not be resolved. This allows passing references through tasks without blocking on their resolution.

Lifetime of an Actor

Actor lifetimes and metadata (e.g., IP address and port) are managed by the GCS. Each client of the actor may cache this metadata locally and use it to send tasks to the actor directly over gRPC.



Unlike task submission, which is fully decentralized and managed by the owner of the task, actor lifetimes are managed centrally by the GCS service.

When an actor is created in Python, the creating worker builds a special task known as an actor creation task that runs the actor's Python constructor. The creating worker waits for any dependencies of the creation task to become ready, similar to non-actor tasks. Once this completes, the creating worker asynchronously registers the actor with the GCS. The GCS then creates the actor by scheduling the actor creation task. This is similar to scheduling for non-actor tasks, except that its specified resources are acquired for the lifetime of the actor process.

Meanwhile, the Python call to create the actor immediately returns an "actor handle" that can be used even if the actor creation task has not yet been scheduled. Subsequent tasks that take the actor handle as an argument will not be scheduled until the actor creation task has completed. See Actor Creation for more details.

Task execution for actors is similar to that of normal tasks: they return futures, are submitted directly to the actor process via gRPC, and will not run until all `ObjectRef` dependencies have been resolved. There are two main differences:

1. By default, resources do not need to be acquired from the scheduler to execute an actor task. This is because the actor has already been granted resources for its lifetime, when its creation task was scheduled.
2. For each caller of an actor, the tasks are executed in the same order[3] that they are submitted. This is because the tasks are assumed to modify the actor state.

An actor will be automatically cleaned up when either its creator exits, or there are no more pending tasks or handles in scope in the cluster (see Reference Counting for details on how this is determined). Note that this is not true for detached actors, which are designed to be long-lived

[3] Unless using async actors or threaded actors.

actors that can be referenced by name and must be explicitly cleaned up using `ray.kill(no_restart=True)`. See Actor Death for more information on actor failures.

In some cases, it may be desirable to break the requirement that actor tasks run sequentially, in the order that they are submitted. To support such use cases, Ray also provides an option for actor concurrency, via async actors that can concurrently run tasks using an asyncio event loop, or threaded actors that run multiple tasks in parallel using threads. Submitting tasks to these actors is the same from the caller's perspective as submitting tasks to a regular actor. The only difference is that when the task is run on the actor, it is posted to a background thread or thread pool instead of running directly on the main thread. Ray APIs such as task submission and ray.get are thread-safe, but the user is responsible for all other thread safety within the actor code.

Code references:

- Backbone for each worker: src/ray/core_worker/core_worker.cc
- Sending and executing actor tasks: src/ray/core_worker/transport/direct_actor_transport.cc
- GCS manager for actor lifetimes: src/ray/gcs/gcs_server/gcs_actor_manager.cc
- GCS scheduler for creating actors: src/ray/gcs/gcs_server/gcs_actor_scheduler.cc
- src/ray/protobuf/core_worker.proto

Failure Model

System Model

Ray worker nodes are designed to be homogeneous, so that any single node may be lost without bringing down the entire cluster. The current exception to this is the head node, since it hosts the GCS. In 2.0, we have added experimental support for GCS fault tolerance, which allows the GCS to be restarted while minimizing disturbance to the rest of the cluster.

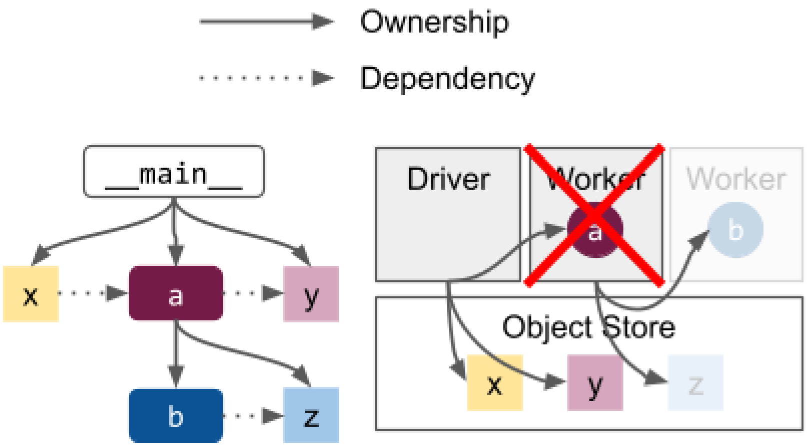
All nodes are assigned a unique identifier and communicate with each other through heartbeats. The GCS is responsible for deciding the membership of a cluster, i.e. which nodes are currently alive. The GCS tombstones any node ID that times out, meaning that a new raylet must be started on that node with a different node ID in order to reuse the physical resources. A raylet that is still alive exits if it hears that it has been timed out. Failure detection of a node currently does not handle network partitions: if a worker node is partitioned from the GCS, it will be timed out and marked as dead.

Each raylet reports the death of any **local** worker process to the GCS. The GCS broadcasts these failure events and uses them to handle actor death. All worker processes fate-share with the raylet on their node.

The raylets are responsible for preventing leaks in cluster resources and system state after individual worker process failures. For a worker process (local or remote) that has failed, each raylet is responsible for:

- Freeing cluster resources, such as CPUs, needed for task execution. This is done by killing any workers that were leased to the failed worker (see Resource Fulfillment). Any outstanding resource requests made by the failed worker are also canceled.
- Freeing any distributed object store memory used for objects owned by that worker (see Memory Management). This also cleans up the associated entries in the object directory.

Application Model



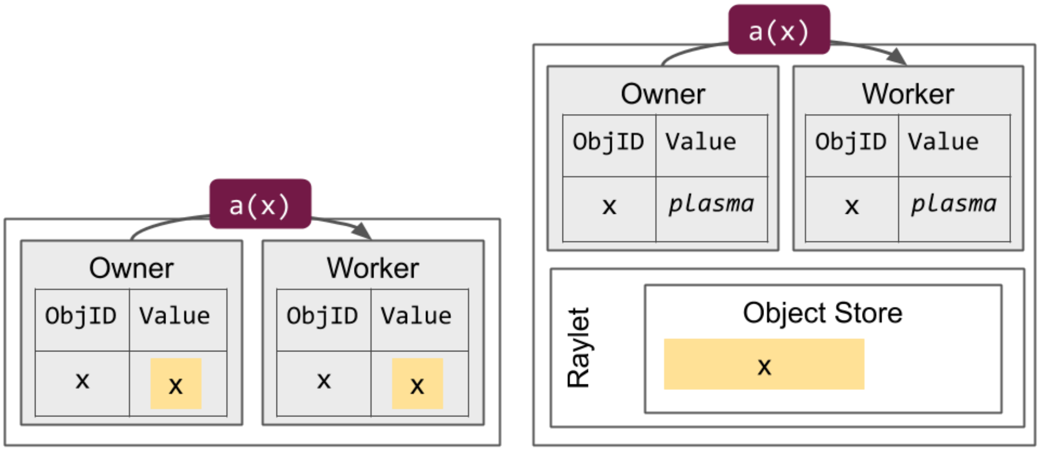
The system failure model implies that tasks and objects in a Ray graph will *fate-share* with their owner. For example, if the worker running `a` fails in this scenario, then any objects and tasks that were created in its subtree (the grayed out `b` and `z`) will be collected. The same applies if `b` were an actor created in `a`'s subtree (see Actor Death). This has a few implications:

- Any other live process will receive an application-level exception if trying to get the value of such an object. For example, if the `z` ObjectRef had been passed back to the driver in the above scenario, the driver would receive an error on `ray.get(z)`.
- Failures can be isolated from one another by modifying the program to place different tasks in different subtrees (i.e. through nested function calls).
- The application will fate-share with the driver, which is the root of the ownership tree.

The main application option to avoid fate-sharing behavior is to use a detached actor, which may live past the lifetime of its original driver and can only be destroyed through an explicit call from the program. The detached actor itself can own any other tasks and objects, which in turn will fate-share with the actor once destroyed.

Ray provides some options to aid in transparent recovery, including automatic task retries and actor restart. As of v1.3, object spilling can also be used to allow objects to persist past the lifetime of their owner. As of v2.0, Ray enables object reconstruction by default for non-actor tasks.

Object Management



(a) Small objects are stored in the in-process store. They are copied directly to the worker through the task description.

(b) Large objects are stored in the distributed object store. The worker retrieves the value through a protocol with the plasma store.

In-process store vs the distributed object store. This shows the differences in how memory is allocated when submitting a task (`a`) that depends on an object (`x`).

In general, small objects are stored in their owner's **in-process store** while large objects are stored in the **distributed object store**. This decision is meant to reduce the memory footprint and resolution time for each object. Note that in the latter case, a placeholder object is stored in the in-process store to indicate the object is actually stored in the distributed object store.

Objects in the in-process store can be resolved quickly through a direct memory copy but may have a higher memory footprint when referenced by many processes due to the additional copies. The capacity of a single worker's in-process store is also limited to the memory capacity of that machine, limiting the total number of such objects that can be in reference at any given time. For objects that are referenced many times, throughput may also be limited by the processing capacity of the owner process.

In contrast, resolution of an object in the distributed object store requires at least one RPC from the worker to the worker's local shared memory store. Additional RPCs may be required if the worker's local shared memory store does not yet contain a copy of the object. On the other hand, because the shared memory store is implemented with shared memory, multiple workers on the same node can reference the same copy of an object. This can reduce the overall memory footprint if an object can be deserialized with zero copies. The use of distributed memory also allows a process to reference an object without having the object local, meaning that a process can reference objects whose total size exceeds the memory capacity of a single machine. Finally, throughput can scale with the number of nodes in the distributed object store, as multiple copies of an object may be stored at different nodes.

Code references:

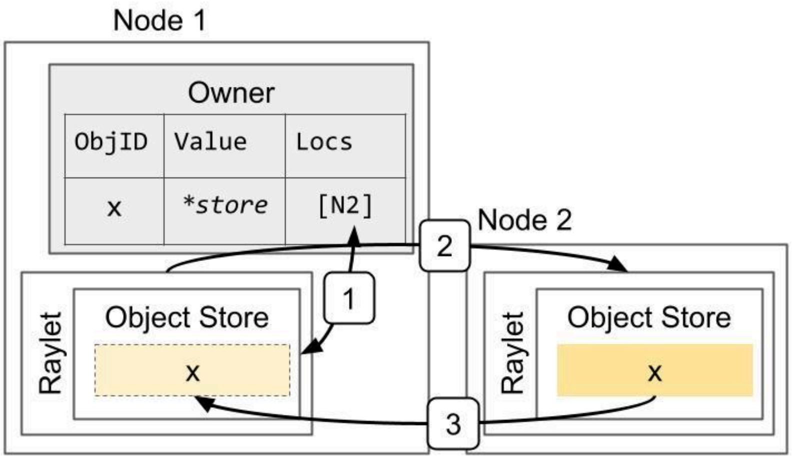
- src/ray/core_worker/store_provider/memory_store/memory_store.cc
- src/ray/core_worker/store_provider/plasma_store_provider.cc
- src/ray/common/buffer.h
- src/ray/protobuf/object_manager.proto

Object resolution

Resolution is the process by which an `ObjectRef` is converted to the underlying physical value, i.e. when calling `ray.get` or passing as a task argument. The `ObjectRef` comprises two fields:

- A unique 28-byte identifier. This is a concatenation of the ID of the task[4] that produced the object and the integer number of objects created by that task so far.
- The address of the object's owner (a worker process). This consists of the worker process's unique ID, IP address and port, and local raylet's unique ID.

Small objects are resolved by copying them directly from the owner's in-process store. For example, if the owner calls `ray.get`, the system looks up and deserializes the value from the local in-process store. If the owner submits a dependent task, it *inlines* the object by copying the value directly into the task specification. Similarly, if a borrower attempts to resolve the value, the object value is copied directly from the owner, bypassing the large object resolution protocol described in the next section.



[4] Task identifiers are computed as a hash of their parent task ID and the number of tasks invoked by that parent before. The root driver task's ID is a monotonically increasing integer, based on the number of jobs that have executed on that cluster before.

Resolving a large object. The object x is initially created on Node 2, e.g., because the task that returned the value ran on that node. This shows the steps when the owner (the caller of the task) calls `ray.get`: 1) Lookup object's locations at the owner. 2) Select a location and send a request for a copy of the object. 3) Receive the object.

Large objects are stored in the distributed object store and must be resolved with a distributed protocol. If the object is already stored in the reference holder's local shared memory store, the reference holder can retrieve the object over IPC. This returns a pointer to shared memory that may be simultaneously referenced by other workers on the same node.

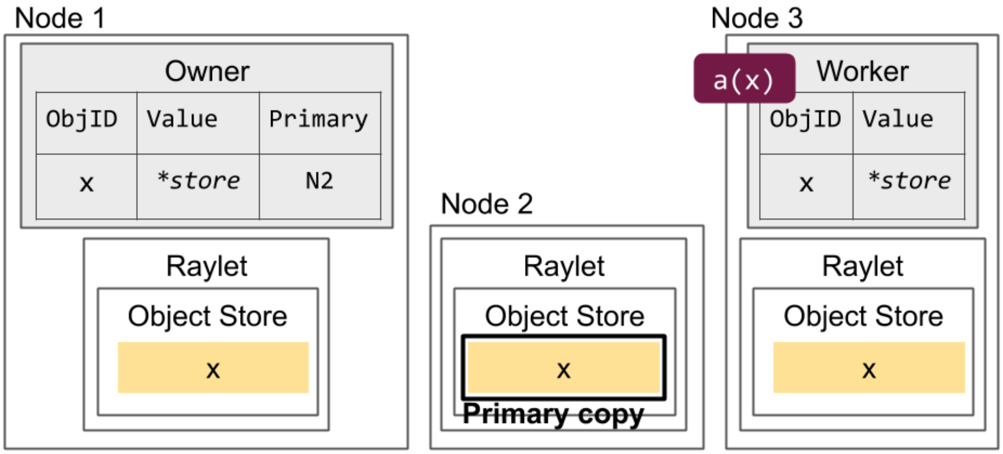
If the object is not available in the local shared memory store, the reference holder notifies its local raylet, which then attempts to fetch a copy from a remote raylet. The raylet looks up the locations from the object directory and requests a transfer from one of these raylets. The object directory is stored at the owners as of Ray v1.3+ (previously it was stored in the GCS).

Code references:

- src/ray/common/id.h
- src/ray/object_manager/ownership_based_object_directory.h

Memory management

For remote tasks, the object value is computed by the executing worker. If the value is small, the worker replies directly to the owner with the value, which is copied into the owner's in-process store. This value is deleted once all references go out of scope.



ObjID	Value	Primary
x	*store	N2

ObjID	Value
x	*store

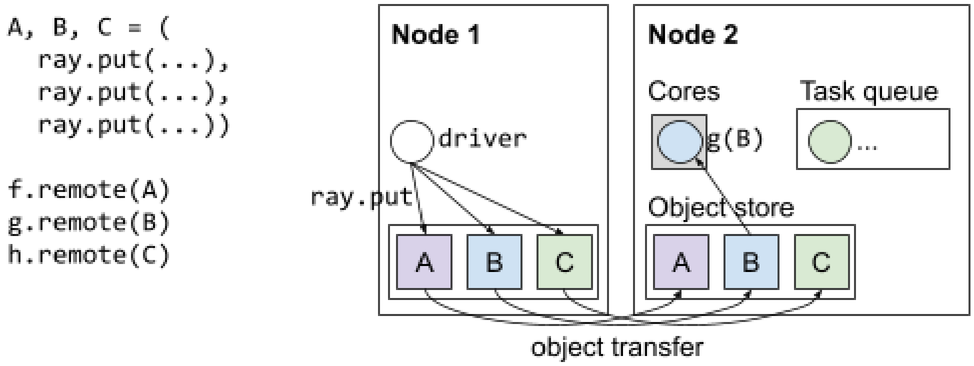
Primary copy versus evictable copies. The primary copy (Node 2) is ineligible for eviction. However, the copies on Nodes 1 (created through `ray.get`) and 3 (created through task submission) can be evicted under memory pressure.

If the value is large, the executing worker stores the value in its local shared memory store. This initial copy of a shared memory object is known as the *primary copy*. The primary copy is unique in that it will not be evicted as long as there is a reference in scope. The raylet "pins" the primary copy by holding a reference to the physical shared memory buffer where the object is stored, which prevents the object store from evicting it. In contrast, other copies of the object may get evicted by LRU if under local memory pressure, unless a Python worker is actively using the object.

In most cases, the primary copy is the first copy of the object to be created. If the initial copy is lost through a failure, the owner will attempt to designate a new primary copy based on the object's available locations.

Once the object ref count goes to 0, all copies of the object are eventually and automatically garbage-collected. Small objects are erased immediately from the in-process store by the owner. Large objects are asynchronously erased from the distributed object store by the raylets.

The raylets also manage distributed object transfer, which creates additional copies of an object based on where the object is currently needed, e.g., if a task that depends on the object is scheduled to a remote node.



```
A, B, C = (
  ray.put(...),
  ray.put(...),
  ray.put(...))

f.remote(A)
g.remote(B)
h.remote(C)
```

The types of objects that can be stored on a node. Objects are either created by a worker (such as A, B, and C on node 1), or a copy is transferred from a different node because it is needed by a local worker (such as A, B, and C on node 2).

Thus, an object may be stored in a node's shared-memory object store due to any of the following reasons:

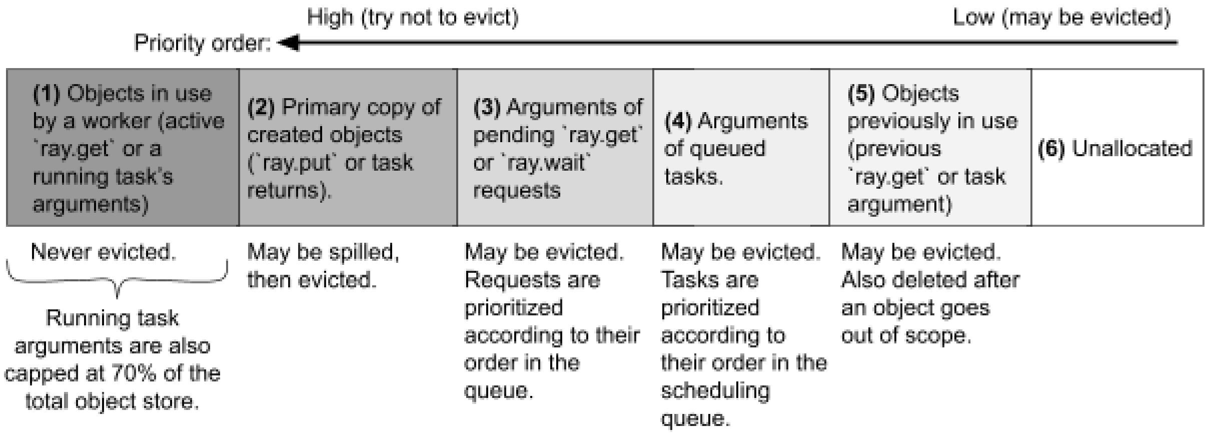
1. It was requested by a local worker process through `ray.get` or `ray.wait`. These can be freed once the worker finishes the `ray.get` request. Note that for objects that can be zero-copy deserialized, the Python value returned from `ray.get` refers directly to the shared-memory buffer, so the object will be "pinned" until this Python value goes out of scope.
2. It was returned by a previous task that executed on that node. These can be freed once there are no more references to the object OR once the object has been spilled.

3. It was created through `ray.put` by a local worker process on that node. These can be freed once there are no more references to the object (objects A, B, and C on node 1 in the above diagram).
4. It is the argument of a task queued or executing on that node. These can be freed once the task completes or is no longer queued. Objects B and C on node 2 are both examples of this, since their downstream tasks g and h have not yet finished.
5. It was previously needed on this node, e.g., by a completed task. Object A on node 2 is an example of this, since f has already finished executing. These objects may be evicted based on local LRU if under memory pressure. They are also eagerly evicted when the ObjectRef goes out of scope (e.g., A is deleted from node 2 after f finishes and after calling `del A`).

Handling out-of-memory cases

For small objects, Ray currently does not impose a memory limit on each worker's in-process store. Thus, an excessive number of small objects in scope may cause the owner process to be killed from out-of-memory.

Ray imposes a hard limit on shared-memory objects The raylet is responsible for enforcing this limit. Below is a visualization of the different types of shared-memory objects that may be stored on a node, with a rough priority.



Priority order: High (try not to evict) ← Low (may be evicted)

(1) Objects in use by a worker (active `ray.get` or a running task's arguments)	**(2)** Primary copy of created objects (`ray.put` or task returns).	**(3)** Arguments of pending `ray.get` or `ray.wait` requests	**(4)** Arguments of queued tasks.	**(5)** Objects previously in use (previous `ray.get` or task argument)	**(6)** Unallocated
Never evicted. Running task arguments are also capped at 70% of the total object store.	May be spilled, then evicted.	May be evicted. Requests are prioritized according to their order in the queue.	May be evicted. Tasks are prioritized according to their order in the scheduling queue.	May be evicted. Also deleted after an object goes out of scope.	

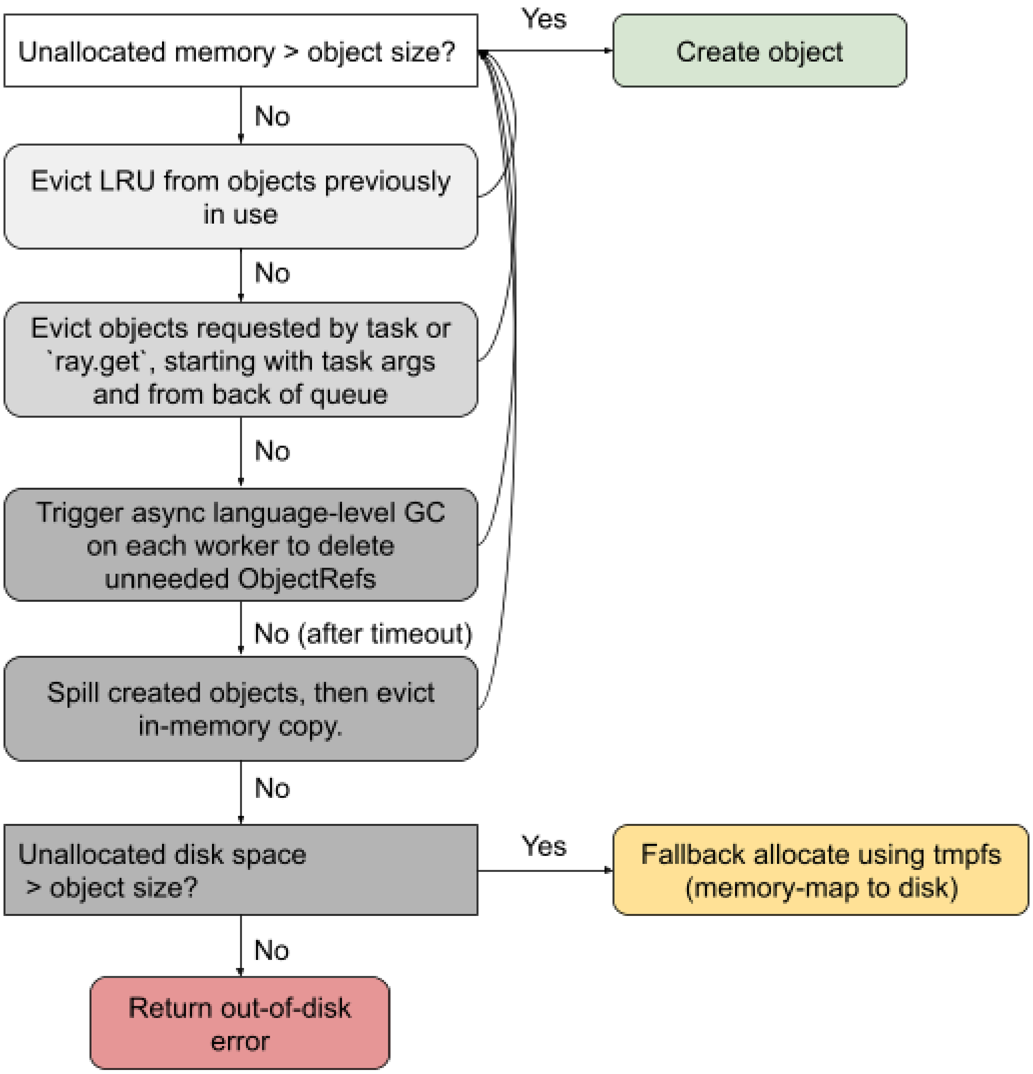
Object creation requests are queued by the raylet and served once enough memory is available in (6) to create the object. If more memory is needed, the raylet will choose objects to evict from (3)-(5) to make space. Even after all of these objects are evicted, the raylet may not have space for the new object. This can happen if the total memory needed by the application is greater than the cluster's memory capacity.

If more space is needed after eviction, the raylet first triggers language-specific garbage collection at each of the workers in the entire cluster. The ObjectRefs seen in the language frontend appear to be very small, and thus are unlikely to trigger the usual language-specific garbage collection mechanisms (e.g., Python's gc.collect()). However, the actual memory footprint of an ObjectRef can be very large, since the physical value is stored elsewhere in Ray's object store, and potentially on a different node(s) from the language-level ObjectRef. Thus, when any Ray object store reaches capacity, we trigger the language-level garbage collection at *all* workers, which cleans up any unneeded ObjectRefs and allows the physical values to be freed from the object store.

The raylet starts a timeout to give workers time to asynchronously garbage-collect ObjectRefs, before triggering spilling to external storage. Spilling allows primary copies in (2) to be freed from the object store even though the objects may still be referenced. If spilling is disabled, the application will instead receive an ObjectStoreFullError after a configurable timeout. Spilling can be expensive and add long delays to task execution; thus Ray also eagerly spills objects once the object store reaches a configurable threshold (80% by default) to try to ensure available space.

Note that the object store can still run out of memory even with object spilling enabled. This can occur if there are too many objects in use (1) at the same time. To mitigate this, the raylet limits the total size of the executing tasks' arguments, since an argument cannot be released until the task completes. The default cap is 70% of the object store memory. This ensures that as long as there are no other objects actively pinned due to a `ray.get` request, it should be possible for a task to create an object that is 30% of the object store's capacity.

Currently, the raylet does not implement a similar cap for objects pinned by workers' `ray.get` request, as doing so naively could introduce deadlock between tasks. Thus, if there are excessive concurrent `ray.get` requests of large objects, the raylet could still run out of shared memory. When this happens, the raylet *fallback-allocates* objects as memory-mapped files on the local disk (/tmp by default). The fallback-allocated objects are less performant due to I/O overhead, but it allows the application to continue running even if the object store is full. The fallback allocation will fail if the local disk is full, after which the application will receive an OutOfDiskError.


Unallocated memory > object size?
Yes
Create object
No
Evict LRU from objects previously in use
No
Evict objects requested by task or `ray.get`, starting with task args and from back of queue
No
Trigger async language-level GC on each worker to delete unneeded ObjectRefs
No (after timeout)
Spill created objects, then evict in-memory copy.
No
Unallocated disk space > object size?
Yes
Fallback allocate using tmpfs (memory-map to disk)
No
Return out-of-disk error

Raylet flowchart for handling an object creation request. If there is not enough available memory in the local object store to serve the request, the raylet attempts a series of steps to make memory available.

Code references:

- src/ray/object_manager/plasma/store.cc
- src/ray/object_manager/plasma/create_request_queue.cc
- src/ray/object_manager/plasma/object_lifecycle_manager.cc

Object spilling

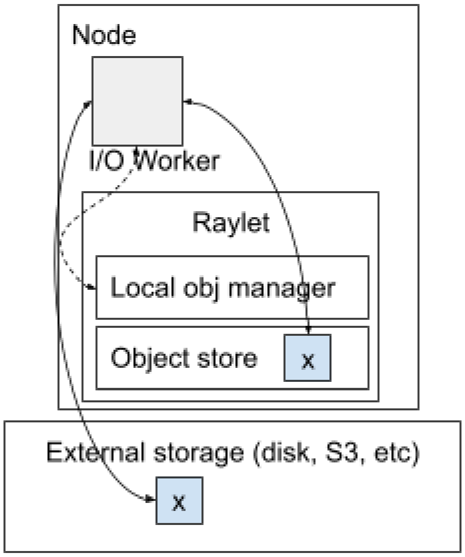
Ray has default support for spilling objects to external storage once the capacity of the object store is used up. This enables out-of-core data processing and memory-intensive distributed applications.

External storage is implemented with a pluggable interface. There are two types of external storage supported by default:

- Local storage (stable). Local disk is selected by default so that Ray users can use the object spilling feature without any additional configuration.
- Distributed storage (experimental, currently offering Amazon S3). Speed of access may be slower but this can provide better fault tolerance, since data will survive worker node failures.

Four components are involved in the object spilling protocol.

- Within the raylet:
  - Local object manager: Keeps track of object metadata, such as the location in external storage, and coordinates IO workers and communication with other raylets.
  - Shared-memory object store
- IO workers: Python processes that spill and restore objects.
- External storage: Stores Ray objects that cannot fit into the object store memory.



An overview of the design for spilling or restoring an object. The raylet manages a pool of I/O workers. I/O workers read/write from the local shared-memory object store and external storage.

When Ray does not have enough memory capacity to create objects, it initiates object spilling. Note that Ray only spills the primary copy of an object: this is the initial copy of the object, created by executing a task or through `ray.put`. Non-primary copies can be evicted immediately, and this design ensures that we have at most one spilled copy of each object in the cluster. Primary copies are only evictable after object spilling, or if there are no more references in the application.

The protocol is as follows, repeating until enough space is made to create any pending objects:

1. Raylet (local object manager) finds all primary copies in the local object store.

2. Raylet sends spill requests for these objects to an IO worker.
3. An IO worker writes the object value along with its metadata to external storage.
4. Once the primary copies are spilled to external storage, the raylet updates the object directory with the location of the spilled objects.
5. The object store evicts the primary copies.
6. Once an object's reference count goes to 0, the owner notifies the raylet that the object can be removed. The raylet sends a request to an IO worker to remove the object from external storage.

Spilled objects are restored as they are needed. When an object is requested, the Raylet either restores the object from external storage by sending a restore request to a local IO worker, or it fetches a copy from a raylet on a different node. The remote raylet might have the object spilled on local storage (e.g., local SSD). In this case, the remote raylet directly reads the object from local storage and sends it to the network.

Spilling many small objects with one object per file is inefficient due to IO overhead. For local storage, the OS would run out of inodes very quickly. If objects are smaller than 100MB, Ray fuses objects into a single file to avoid this problem.

Ray also supports multi-directory spilling, meaning it utilizes multiple file systems mounted at different locations. This helps to improve spilling bandwidth and maximum external storage capacity when there are multiple local disks attached to the same machine.

Known limitations:

- When using local file storage, spilled objects are lost if the node where the object is stored is lost. In this case, Ray will attempt to recover the object as if it was lost from shared memory.
- A spilled object is not reachable if the owner is lost, since the owner stores the object's locations.
- Objects that are currently in use by the application are "pinned". For example, if the Python driver has a raw pointer to an object that was obtained by ray.get, (e.g., a numpy array view over shared memory), then the object is pinned. These objects are not spillable until the application releases them. Arguments of a running task are also pinned for the task's duration.

Code references:

- src/ray/raylet/local_object_manager.cc
- python/ray/_private/external_storage.py

Reference Counting

Each worker stores a ref count for each object that it owns. The owner's local ref count includes the local Python ref count and the number of pending tasks submitted by the owner that depend on the object. The former is decremented when a Python `ObjectRef` is deallocated. The latter is decremented when a task that depends on the object successfully finishes (note that a task that ends in an application-level exception counts as a success).

`ObjectRef`s can also be copied to another process by storing them inside another object. The process that receives the copy of the `ObjectRef` is known as a *borrower*. For example:

```
@ray.remote
def temp_borrow(obj_refs):
  # Can use obj_refs temporarily as if I am the owner.
  x = ray.get(obj_refs[0])

@ray.remote
class Borrower:
  def borrow(self, obj_refs):
    # We save the ObjectRef in local state, so we are still borrowing the
object once this task finishes.
    self.x = obj_refs[0]

x_ref = foo.remote()
temp_borrow.remote([x_ref])  # Passing x_ref in a list will allow `borrow`
to run before the value is ready.
b = Borrower.remote()
b.borrow.remote([x_ref])  # x_ref can also be borrowed permanently by an
actor.
```

These references are tracked through a distributed reference counting protocol. Briefly, the owner adds to the local ref count whenever a reference "escapes" the local scope. For example, in the above code, the owner would increment the pending task count for x_ref when calling `temp_borrow.remote` and `b.borrow.remote`. Once the task finishes, it replies to its owner with a list of the references that are still being borrowed. For example, in the above code, `temp_borrow`'s worker would reply saying that it is no longer borrowing `x_ref`, while the `Borrower` actor would reply saying that it is still borrowing `x_ref`.

If the worker is still borrowing any references, the owner adds the worker's ID to a local list of borrowers. The borrower keeps a second local ref count, similar to the owner, and the owner asks the borrower to reply once the borrower's local ref count has gone to 0. At this point, the owner may remove the worker from the list of borrowers and collect the object. In the above example, the `Borrower` actor is borrowing the reference permanently, so the owner would not free the object until the `Borrower` actor itself goes out of scope or dies.

Borrowers can also be added recursively to the owner's list. This happens if the borrower itself passes the `ObjectRef` to another process. In this case, when the borrower responds to the owner that its local ref count is 0, it also includes any new borrowers that it has created. The owner in turn contacts these new borrowers using the same protocol.

A similar protocol is used to track `ObjectRef`s that are *returned* by their owner. For example:

```
@ray.remote
def parent():
  y_ref = child.remote()
  x_ref = ray.get(y_ref)
  x = ray.get(x_ref)

@ray.remote
def child():
  x_ref = foo.remote()
  return x_ref
```

When the `child` function returns, the owner of `x_ref` (the worker that executes `child`) would mark that `x_ref` is contained in `y_ref`. The owner would then add the `parent` worker to the list of borrowers for `x_ref`. From here, the protocol is similar to the above: the owner sends a message to the `parent` worker asking the borrower to reply once its references to both `y_ref` and `x_ref` have gone out of scope.

Reference type	**Description**	**When is it updated?**
Local Python ref count	Number of local `ObjectRef` instances. This is equal to the worker's process-local Python ref count.	Incremented/decremented when a new Python `ObjectRef` is allocated/deallocated.
Submitted task count	Number of tasks that depend on the object that have not yet completed execution.	Incremented when the worker submits a task (e.g., `foo.remote(x_ref)`). Decremented when the task completes. If the object is small enough to be stored in the in-process store, this count is decremented early, when the object is copied into the task specification.
Borrowers	A set of worker IDs for the processes that are currently borrowing the `ObjectRef`. A borrower is any worker that is not the owner and that has a local instance of the Python	When a task is passed an ObjectRef and continues to use it past the end of the task (e.g., saving the ObjectRef in an actor's local state), the task notifies its caller that it is borrowing the object. Then, the calling worker adds the task worker's ID to this set.

	`ObjectRef`. Each borrower also maintains a local set of borrowers, forming a tree of borrower sets rooted at the owner. This allows a borrower to send the `ObjectRef` to another borrower without having to contact the owner.	Removal if an owner: The owner sends an async RPC to each of the borrower workers. A borrower responds once its ref count for the `ObjectRef` goes to 0. The owner removes a worker when it receives this reply or if the borrower dies first. Removal if a borrower: The worker waits for RPC from the owner. Once the worker's local ref count (local Python count + submitted task count) is 0, the worker pops its local set of borrowers into the reply to the owner. In this way, the owner learns of and can track recursive borrowers.
Nested count	Number of `ObjectRef`s that are in scope and whose values contain the `ObjectRef` in question.	Incremented when the `ObjectRef` is stored inside another object (e.g., `ray.put([x_ref])` or `return x_ref`). Decremented when the outer `ObjectRef` goes out of scope.
Lineage count	Maintained when reconstruction is enabled. Number of tasks that depend on this `ObjectRef` whose values are stored in the distributed object store (and therefore may be lost upon a failure).	Incremented when a task is submitted that depends on the object. Decremented if the task's returned `ObjectRef` goes out of scope, or if the task completes and returns a value in the in-process store.

Summary of the different types of references and how they are updated.

Corner cases

References that are captured in a remote function or class definition will be pinned permanently. For example:

```
x_ref = foo.remote()
@ray.remote
def capture():
  ray.get(x_ref)  # x_ref is captured. It will be pinned as long as the
driver lives.
```

The conventional method for creating references is to pass ObjectRefs to other workers as task arguments, either directly or inside a data structure like a list. References can also be created "out-of-band" by pickling an `ObjectRef` with `ray.cloudpickle`. In this case, Ray cannot track the serialized copy of the object or determine when the ObjectRef has been deserialized (e.g., if the

ObjectRef is deserialized by a non-Ray process). Thus, a permanent reference will be added to the object's count to prevent the object from going out of scope.

Other methods of out-of-band serialization include using `pickle` or custom serialization methods. Similar to above, Ray cannot track these references. Accessing the deserialized ObjectRef, i.e. by calling `ray.get` or passing as a task argument, may result in a reference counting exception.

Code references:

- src/ray/core_worker/reference_count.cc
- python/ray/includes/object_ref.pxi
- java/runtime/src/main/java/io/ray/runtime/object/ObjectRefImpl.java

Actor handles

The same reference counting protocol described above is used to track the lifetime of an (non-detached) actor. A *dummy object* is used to represent the actor. This object's ID is computed from the ID of the actor creation task. The creator of the actor owns the dummy object.

When the Python actor handle is deallocated, this decrements the local ref count for the dummy object. When a task is submitted on an actor handle, this increments the submitted task count for the dummy object. When an actor handle is passed to another process, the receiving process is counted as a borrower of the dummy object. Once the ref count reaches 0, the owner notifies the GCS service that it is safe to destroy the actor.

Note that detached actors are not automatically garbage-collected by Ray. They must be explicitly deleted by the application.

Code references:

- src/ray/core_worker/actor_handle.cc
- python/ray/actor.py
- java/api/src/main/java/io/ray/api/ActorCall.java

Interaction with Python GC

When objects are part of reference cycles in Python, the Python garbage collector does not guarantee these objects will be garbage collected in a timely fashion. Since uncollected Python `ObjectRef`s can spuriously keep Ray objects alive in the distributed object store, Ray triggers `gc.collect()` in all Python workers periodically and when the object store is near capacity. This ensures that Python reference cycles never lead to a spurious object store full condition.

Object Failure

In the event of a system failure, Ray will attempt to recover any lost objects and, if recovery is not possible, will instead throw an application-level exception if a worker tries to get the value of the object.

At a high level, Ray guarantees that if the owner is still alive, recovery of the object will be attempted. If recovery fails, the owner will populate the exception with the cause. Otherwise, if the owner of an object has died, any worker that tries to get the value will receive a generic error about the owner's death, even if the object copies still exist in the cluster.

Small objects

Small objects: Small objects are stored in the owner's in-process object store and thus will be lost if the owner dies. Any workers that try to get the value of the object in the future will learn that the owner has died and store the error in their local in-process object store. If the worker tries to access the object, e.g., via ray.get(), it will receive this error.

Large objects and lineage reconstruction

If the object is lost from distributed memory: Non-primary copies of an object can be lost without consequences. If the primary copy of an object is lost, the owner will attempt to designate a new primary copy by looking up the remaining locations in the object directory.

If no other copies exist, Ray will then attempt to recover the object through *object reconstruction*. This refers to the recovery of a lost object through re-execution of the task that created the object. If dependencies of the task are also lost, or were evicted previously due to garbage collection, then these objects are recursively reconstructed.

Lineage reconstruction works by keeping an additional "lineage ref count" alongside each object. This refers to the number of tasks that depend on the object that may themselves be re-executed. A task can be re-executed if any objects that it or a downstream task returns are still in scope. Once the lineage ref count reaches 0, Ray will garbage-collect the specification of the task that creates the object. Note that this is a separate garbage collection mechanism from the object *value*: If an object's direct reference count reaches 0, its value will be garbage-collected from Ray's object store even if its lineage stays in scope.

Note that lineage reconstruction can cause higher than usual driver memory usage because of the cached lineage. Each Ray worker will attempt to evict their locally cached lineage if the total size exceeds a system-wide threshold (default 1GB).

Lineage reconstruction currently has the following limitations. If the application does not meet these requirements, then it will instead receive an error with the reason reconstruction failed:

- The object, and any of its transitive dependencies, must have been generated by a task (actor or non-actor). This means that **objects created by ray.put are not recoverable**. Note that objects created by `ray.put` are always stored on the same node as their owner and the owner will fate-share with this node; thus in the case that the primary copy of ` ray.put` object is lost, the application will receive a generic `OwnerDiedError`.
- Tasks are assumed to be deterministic and idempotent. Thus, by default, **objects created by actor tasks are not reconstructable**. Actor tasks may be re-executed as part of the lineage if the user sets the actor's `max_task_retries` and `max_restarts` to a nonzero value.
- Tasks will only be re-executed up to their maximum number of retries. By default, a non-actor task can be retried up to 3 times and an actor task cannot be retried. This can be overridden with the `max_retries` parameter for non-actor tasks and the `max_task_retries` parameter for actors.
- The owner of the object must still be alive (see below).

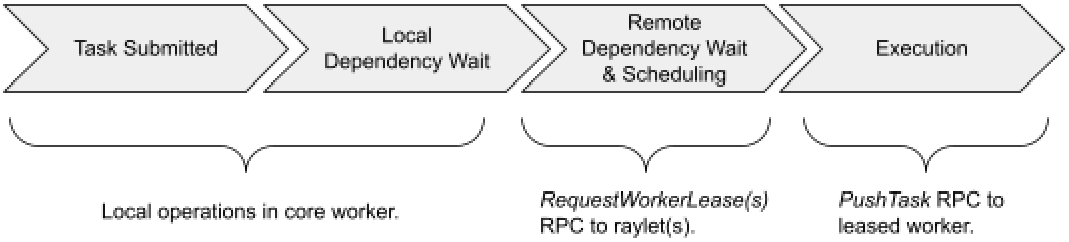
If the owner of an object stored in distributed memory is lost: During object resolution, a raylet will attempt to locate a copy of the object. At the same time, the raylet will periodically contact the owner to check that the owner is still alive. If the owner has died, the raylet will store a system-level error that will be thrown to the reference holder during object resolution.

Code references:

- src/ray/core_worker/object_recovery_manager.h - Recovery protocol
- src/ray/core_worker/reference_count.h - Lineage ref counting
- src/ray/core_worker/task_manager.h - Lineage cache

Task Management

Task execution


Task Submitted
Local Dependency Wait
Remote Dependency Wait & Scheduling
Execution
Local operations in core worker.
RequestWorkerLease(s) RPC to raylet(s).
PushTask RPC to leased worker.

The scheduling workflow of a normal Ray task.

Dependency resolution

The task caller waits for all task arguments to be created before requesting resources from the distributed scheduler. In many cases, the caller of a task is also the owner of the task arguments. For example, for a program like `foo.remote(bar.remote())`, the caller owns both tasks and will not schedule `foo` until `bar` has completed. This can be executed locally because the caller will store the result of `bar` in its in-process store.

The caller of a task may be *borrowing* a task argument, i.e., it received a deserialized copy of the argument's `ObjectRef` from the owner. In this case, the task caller must determine whether the argument has been created by executing a protocol with the owner of the argument. A borrower process will contact the owner upon deserializing an `ObjectRef`. The owner responds once the object has been created, and the borrower marks the object as ready. If the owner fails, the borrower also marks the object as ready, since objects fate-share with their owner.

```python
@ray.remote
def caller(refs: List[ObjectRef]):
  foo.remote(refs[0])  # caller borrows refs[0] and it will only schedule foo after the object referenced by refs[0] is created.
```

Tasks can have three types of arguments: plain values, inlined objects, and non-inlined objects.

- Plain values: f.remote(2)
- Inlined object: f.remote(small_obj_ref)
- Non-inlined object: f.remote(large_or_pending_obj_ref)

Plain values don't require dependency resolution.

Inlined objects are objects small enough to be stored in the in-process store (default threshold is 100KB). The caller can copy these directly into the task specification.

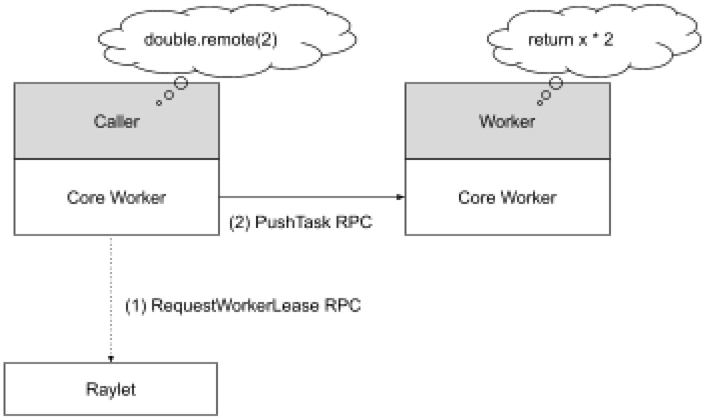
Non-inlined objects are those stored in the distributed object store. These include large objects and objects that have been borrowed by a process other than the owner. In this case, the caller will ask the raylet to account for these dependencies during the scheduling decision. The raylet will wait for those objects to become local to its node before granting a worker lease for the dependent task. This ensures that the executing worker will not block upon receiving the task, waiting for the objects to become local.

Resource fulfillment

A task caller schedules a task by first sending a resource request to the preferred raylet for that request. This is chosen either:

1. By data locality: If the task has object arguments stored in shared memory, then the caller chooses the node that has the most number of object argument bytes already local. This information is retrieved through the caller's local object directory and may be stale (e.g., if an object transfer or eviction occurs concurrently).
2. By node affinity: If a target raylet is specified by the NodeAffinitySchedulingStrategy.
3. By default, the local raylet.

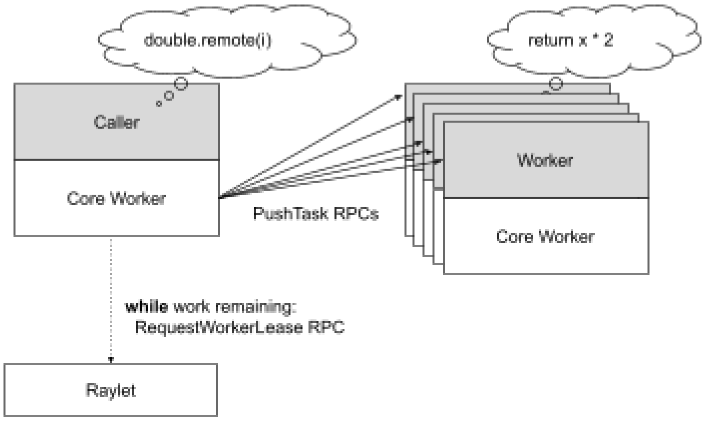
The preferred raylet queues the request and if it chooses to grant the resources, responds to the caller with the address of a local worker that is now *leased* to the caller. The lease remains active as long as the caller and leased worker are alive, and the raylet ensures that no other client may use the worker while the lease is active. To ensure fairness, a caller returns the idle worker if no more task remains or if enough time has passed (e.g., a few hundred milliseconds).



Resource fulfillment and execution of the `double(2)` task in a Ray cluster.

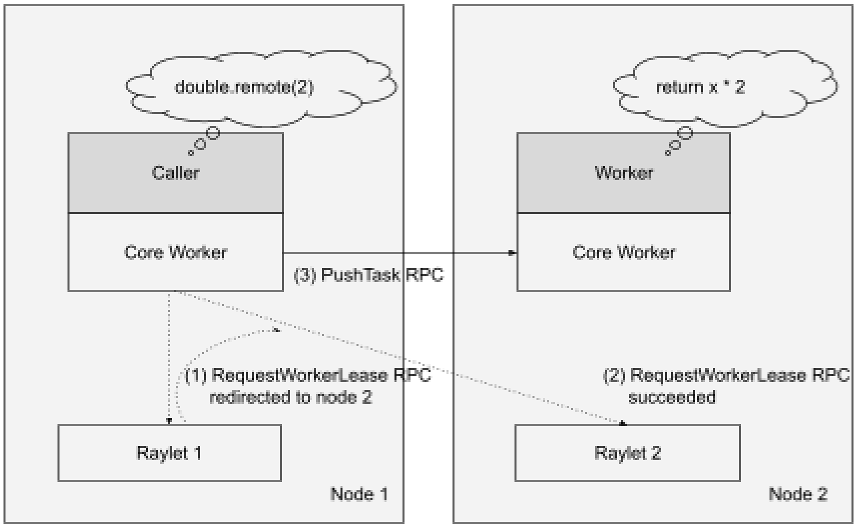
The caller may schedule any number of tasks onto the leased worker, as long as the tasks are compatible with the granted resource request. Hence, leases can be thought of as an optimization to avoid communication with the scheduler for similar scheduling requests. A scheduling request can reuse the leased worker if it has the same:

1. Resource shape (e.g., {"CPU": 1}), as these must be acquired from the node during task execution.
2. Shared-memory task arguments, as these must be made local on the node before task execution. Note that small task arguments do not need to match since these are inlined into the task argument. Also, ObjectRefs that are passed inside a data structure do not need to match, since Ray does not make these local before the task begins.
3. Runtime environment, as the leased worker is started inside this environment.



Caller can hold multiple worker leases to increase parallelism. Worker leases are cached across multiple tasks as an optimization to reduce the load on the scheduler.

If the preferred raylet chooses not to grant the resources locally, it may also respond to the caller with the address of a remote raylet at which the caller should retry the resource request. This is known as *spillback scheduling*. The remote raylet may grant or reject the resource request depending on the current availability of its local resources. Should the resource request be rejected, the caller will request again from the preferred raylet and the same process repeats until the resource request is granted by some raylet.



During spillback scheduling, the local raylet redirects the caller's request to a remote raylet that might have resources available.

Code references:

- src/ray/core_worker/core_worker.cc
- src/ray/common/task/task_spec.h
- src/ray/core_worker/transport/direct_task_transport.cc
- src/ray/core_worker/transport/dependency_resolver.cc

- src/ray/core_worker/task_manager.cc
- src/ray/protobuf/common.proto
- Backbone class for raylet: src/ray/raylet/node_manager.cc
- Distributed scheduler on each raylet: src/ray/raylet/scheduling/cluster_task_manager.cc
- Local task queue on each raylet: src/ray/raylet/local_task_manager.cc

Resource Management and Scheduling

A resource in Ray is a key-value pair where the key denotes a resource name, and the value is a float quantity. For convenience, the Ray scheduler has native support for CPU, GPU, and memory resource types. Ray resources are *logical* and don't need to have 1-to-1 mapping with physical resources (e.g. you can start a Ray node with 3 GPUs even if it physically has zero). By default, Ray sets the quantities of logical resources on each node to the physical quantities auto detected by Ray.

The user may also define custom resource requirements using any valid string, e.g., specifying a resource requirement of {“custom_resource”: 0.01}. Custom resources can be added to a node on startup, for example, to advertise that a node has a particular hardware feature. By requesting that custom resource, tasks or actors can effectively be constrained to running on that particular node.

The purpose of the distributed scheduler is to match resource requests from the callers to resource availability in the cluster. Resource requests are hard scheduling constraints. For example, {"CPU": 1.0, "GPU": 1.0} represents a request for 1 CPU and 1 GPU. This task can only be scheduled on a node that has >= 1 CPU and >= 1 GPU. Each `@ray.remote` function requires 1 CPU for execution ({“CPU”: 1}) by default. An actor, i.e. a `@ray.remote` class, will request 0 CPU for execution by default. This is so that a single node can host more actors than it has cores, leaving CPU multiplexing to the OS. Actors also request 1 CPUs for *placement*, meaning that the chosen node must have at least 1 CPU in its total resources. This is to enable applications to prevent actors from being scheduled to particular nodes, i.e. by starting the node with `--num-cpus=0`.

There are a few resources with special handling:

- The quantity of "CPU", "GPU", and "memory" are autodetected during Ray startup.
- Assigning "GPU" resources to a task will automatically set the CUDA_VISIBLE_DEVICES env var within the worker to limit it to specific GPU ids.

Note that because resource requests are logical, physical resource limits are not enforced by Ray. It is up to the user to specify accurate resource requirements, e.g., specifying `num_cpus=n` for a task with n threads. The main purposes of Ray’s resource requirements are admission control and intelligent autoscaling.

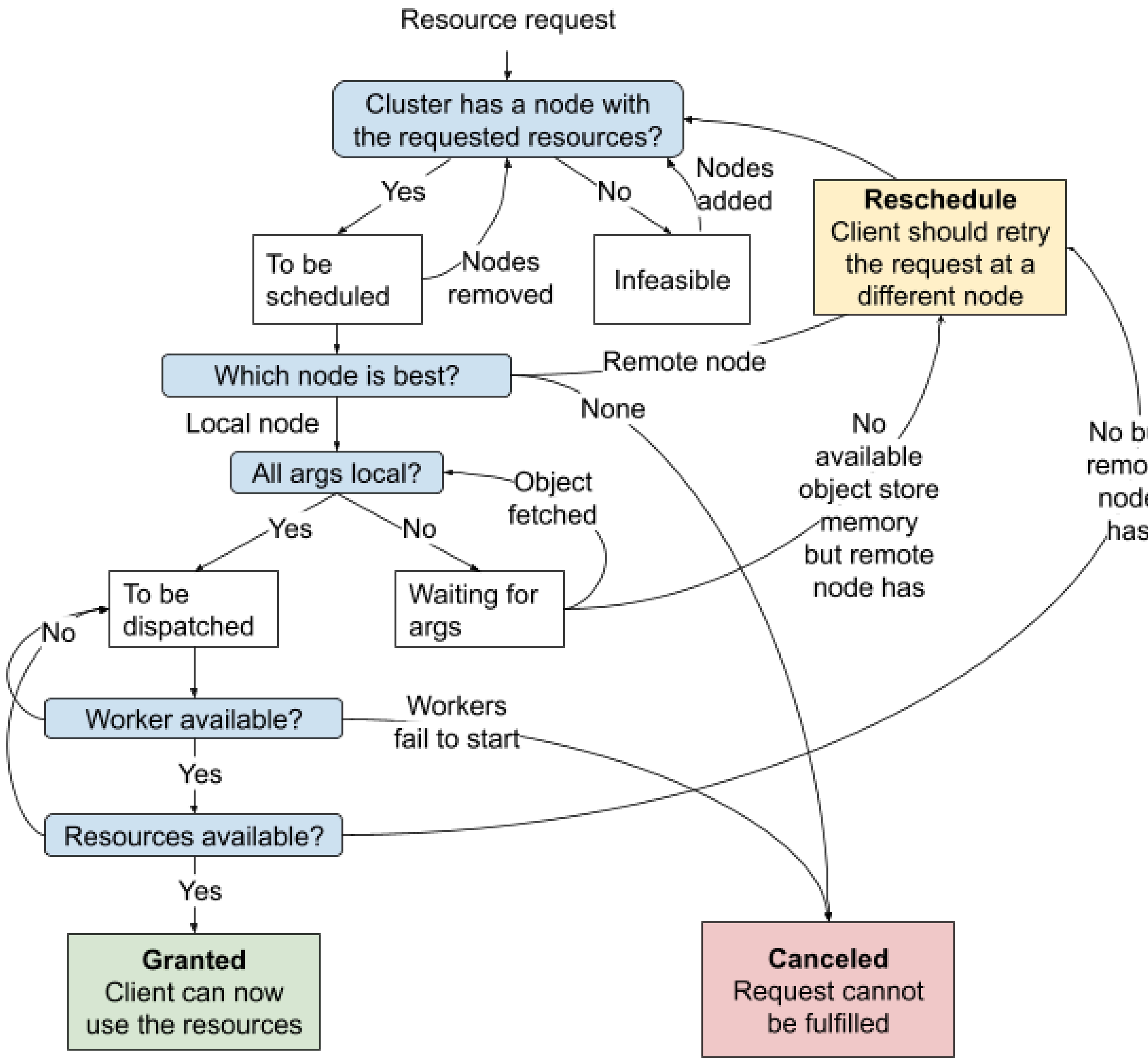
Distributed scheduler

Resource accounting

Each raylet tracks the resources local to its node. When a resource request is granted, the raylet decreases the available local resources accordingly. Once the resources are returned (or the requester dies), the raylet increases the local resource availability accordingly. Thus, the raylet always has a strongly consistent view of the local resource availability.

Each raylet also receives information from the GCS about resource availability on other nodes in the cluster. This is used for distributed scheduling, e.g., to load-balance across nodes in the cluster. To reduce overheads of collection and dissemination, this information is only eventually consistent; it may be stale. The information is sent through a periodic broadcast. GCS pulls resource availability from each raylet periodically (100ms by default) and then aggregates and rebroadcasts them back to each raylet.

Scheduling state machine



State machine of raylet distributed scheduling

When a resource request (i.e RequestWorkerLease PRC) is received by the raylet, it will go through the above state machine and end with one of the three states:

- **Granted**: The client may now use the granted resources and worker to execute a task or actor.
- **Reschedule**: There was a better node than the current one, according to the current node's view of the cluster. The client should reschedule the request. There are two possibilities:
  - If the current node is the client's preferred raylet (i.e. the first raylet that the client contacted), then this is a *spillback request*. The client should retry the request at the raylet specified by the first raylet.

    - Else, the current node was the one chosen by the client's preferred raylet. The client should retry the request at the preferred raylet again.
- **Canceled:** The resource request could not be run. This can happen if the requested scheduling policy is not feasible. For example:
    - The client requested a hard node affinity policy but the node is dead.
    - The requested runtime env for a task failed to be created, so the raylet could not start workers to fulfill the request.

The actual logic of deciding "Which node is best" to fulfill the request is controlled by the scheduling policies, described below.

Code references:
- src/ray/raylet/node_manager.cc
- src/ray/raylet/local_task_manager.cc
- src/ray/raylet/scheduling
- src/ray/protobuf/node_manager.proto

Scheduling policies

Ray has several scheduling policies that control where to run the task or actor. When a task or actor is submitted, the user can optionally specify a scheduling strategy/policy to use (e.g. `task.options(scheduling_strategy=MySchedulingPolicy).remote()`).

Default Hybrid Policy

This is the default policy when nothing else is specified. This policy first tries to pack tasks onto the local node until the node's *critical resource utilization* exceeds a configured threshold (50% by default). The critical resource utilization is the max utilization of any resource on that node, e.g., if a node is using 8/10 CPUs and 70/100GB RAM, then its critical resource utilization is 80%.

After the threshold is exceeded on the local node, the policy packs tasks onto the first remote node (sorted by the node id), then the second remote node and so on and so forth until the critical resource utilization on all nodes exceeds the threshold. After that it will pick the node with the least critical resource utilization.

The purpose of this policy is to achieve a medium between bin-packing and load-balancing. When nodes are under the critical resource utilization, the policy favors bin-packing. Sorting by node ID ensures that all nodes use the same order when bin-packing. When nodes are over the critical resource utilization, the policy favors load-balancing, picking the least-loaded node.

Spread Policy

This policy spreads tasks among nodes with available resources using round-robin. Note that the round-robin order is local to each node's distributed scheduler; there is no global round-robin. If no node has available resources, tasks are spread round-robin among feasible nodes.

Node Affinity Policy

With this policy, a user can explicitly specify a target node where the task or actor should run. If the target node is alive, the task or actor only runs there. If the target node is dead, then depending on whether the affinity is soft or not, the task or actor may be scheduled to other nodes or fail to be scheduled.

Data Locality Policy

Ray supports data locality by having each task caller choose a preferred raylet based on the caller's local information about task argument locations. The scheduling policies implemented by the raylets do not consider data locality. This is to avoid adding extra RPCs and complexity to raylets to discover which task arguments are stored on which other nodes.

Placement Group Policy

This policy runs the task or actor where the given placement group is located.

Code references:

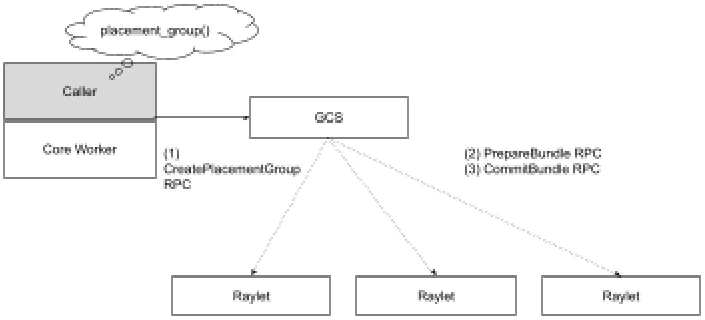
- src/ray/raylet/scheduling/policy
- python/ray/util/scheduling_strategies.py

Placement Groups

Since 1.0, Ray supports Placement Groups. Placement groups allow users to atomically reserve groups of resources across multiple nodes (i.e., for gang scheduling). Each placement group consists of *resource bundles*, i.e. {CPU: 2}. Bundles in the group can be packed as close as possible for locality (PACK), or spread apart (SPREAD). Groups can be destroyed to release all resources associated with the group. The Ray Autoscaler is aware of placement groups, and auto-scales the cluster to ensure pending groups can be placed as needed.

Placement Group Creation

When the application requests a placement group creation, the worker sends a synchronous RPC to the GCS. The GCS flushes the creation request to its backing storage and queues the creation request.



Since resource groups may involve resources across multiple nodes, Ray uses a two phase commit protocol across the raylets to ensure atomicity. The protocol is coordinated by the GCS. If any raylet dies in the middle of the protocol, the placement group creation is rolled back and the GCS queues the request again. If the GCS dies and GCS fault tolerance is enabled, it pings all participants upon restart to reinitiate the protocol.

Placement Group Lifetime

Unlike other Ray primitives (tasks, actors, objects), placement groups are not reference-counted. They are owned by a job or a detached actor that creates them and are automatically destroyed when the owner is dead. Users are also allowed to destroy the placement group using the API `remove_placement_group`. Like actors, placement groups also support detached placement groups, which live beyond the lifetime of their owners.

When a placement group is destroyed, all actors and tasks that use the reserved resources are killed, and all reserved resources are released.

Fault Tolerance

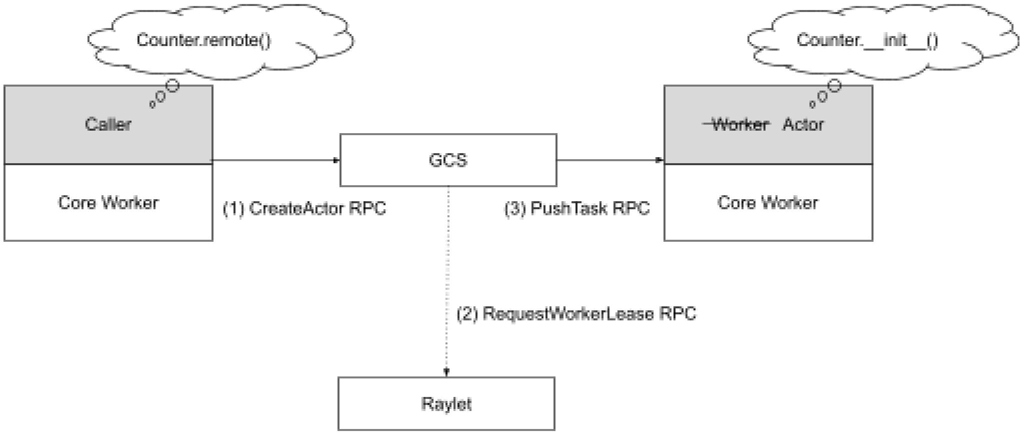
When a placement group is created, it reserves resource bundles across multiple nodes. When one of the nodes is killed, the lost bundles are rescheduled with higher priority than any pending placement groups. Until those bundles are recreated, the placement group remains in a partially allocated state.

Code references:

- src/ray/gcs/gcs_server/gcs_placement_group_manager.h
- src/ray/gcs/gcs_server/gcs_placement_group_scheduler.h

Actor management

Actor creation



Actor creation tasks are scheduled through the centralized GCS service.

When an actor is created in Python, the creating worker first registers the actor with the GCS. For detached actors, the registration is done in a synchronized way to avoid race conditions between actors with the same name. For non-detached actors (the default), registration is asynchronous for performance.

After the registration, once all of the input dependencies for an actor creation task are resolved, the creator then sends the task specification to the GCS service. The GCS service then schedules the actor creation task through the same distributed scheduling protocol that is used for normal tasks, as if the GCS were the actor creation task's caller.

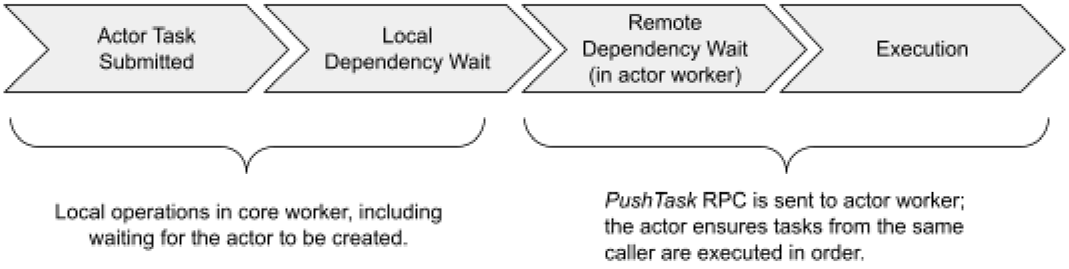
The original creator of the handle can begin to submit tasks on the actor handle or even pass it as an argument to other tasks/actors before the GCS has scheduled the actor creation task. Note that for asynchronous registration, the creator does not pass the actor handle to other tasks/actors until the actor has been registered with the GCS. This is in case the creator dies before registration finishes; by blocking task submission, we can ensure that other workers with a reference to the actor can discover the failure. In this case, task submission is still asynchronous, as the creator simply buffers the remote task until the actor registration is complete.

Once the actor has been created, the GCS notifies any worker that has a handle to the actor via pub-sub. Each handle caches the newly created actor's run-time metadata (e.g., RPC address).

Any pending tasks that were submitted on the actor handle can then be sent to the actor for execution.

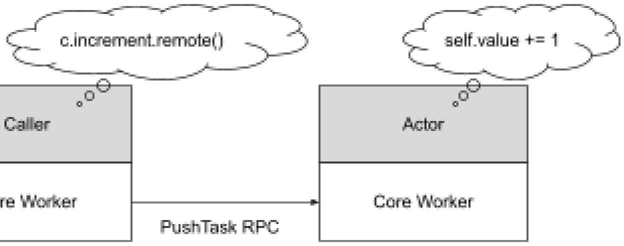
Similar to task definitions, actor definitions are downloaded onto workers via the GCS.

Actor task execution


Actor Task Submitted
Local Dependency Wait
Remote Dependency Wait (in actor worker)
Execution
Local operations in core worker, including waiting for the actor to be created.
PushTask RPC is sent to actor worker; the actor ensures tasks from the same caller are executed in order.

The scheduling workflow of a Ray actor task.

An actor can have an unlimited number of callers. An actor handle represents a single caller: it contains the RPC address of the actor to which it refers. The calling worker connects and submits tasks to this address.



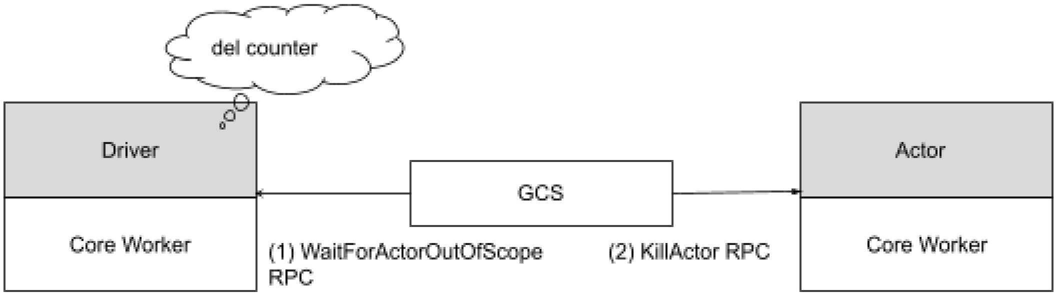
Once created, actor tasks translate into direct gRPC calls to the actor process. An actor can handle many concurrent calls, though here we only show one.

The execution order of submitted tasks is described here.

Actor death

Actors may be detached or non-detached. Non-detached actors are the default and are recommended for most use cases. They are automatically garbage-collected by Ray when all handles go out of scope or the job exits. Detached actors' lifetimes are not tied to their original creator and must be deleted manually by the application once they are no longer needed.

For a non-detached actor, when all pending tasks for the actor have finished and all handles to the actor have gone out of scope (tracked through reference counting), the original creator of the actor notifies the GCS service. The GCS service then sends a KillActor RPC to the actor that will cause the actor to exit its process. The GCS also terminates the actor if it detects that the creator has exited (published through the heartbeat table). All pending and subsequent tasks submitted on this actor will then fail with a RayActorError.



Actor termination is also through the GCS.

Actors may also unexpectedly crash during their runtime (e.g., from a segfault or calling sys.exit). By default, any task submitted to a crashed actor will fail with a RayActorError, as if the actor exited normally.

Ray also provides an option (`max_restarts`) to automatically restart actors, up to a specified number of times. If this option is enabled and the actor's owner is still alive, the GCS service will attempt to restart a crashed actor by resubmitting its creation task. All clients with handles to the actor will cache any pending tasks to the actor until the actor has been restarted. If the actor is not restartable or has reached the maximum number of restarts, the client will fail all pending tasks.

A second option (`max_task_retries`) can be used to enable automatic retry of failed actor tasks after the actor has restarted. This can be useful for idempotent tasks and cases where the user does not require custom handling of a RayActorError.

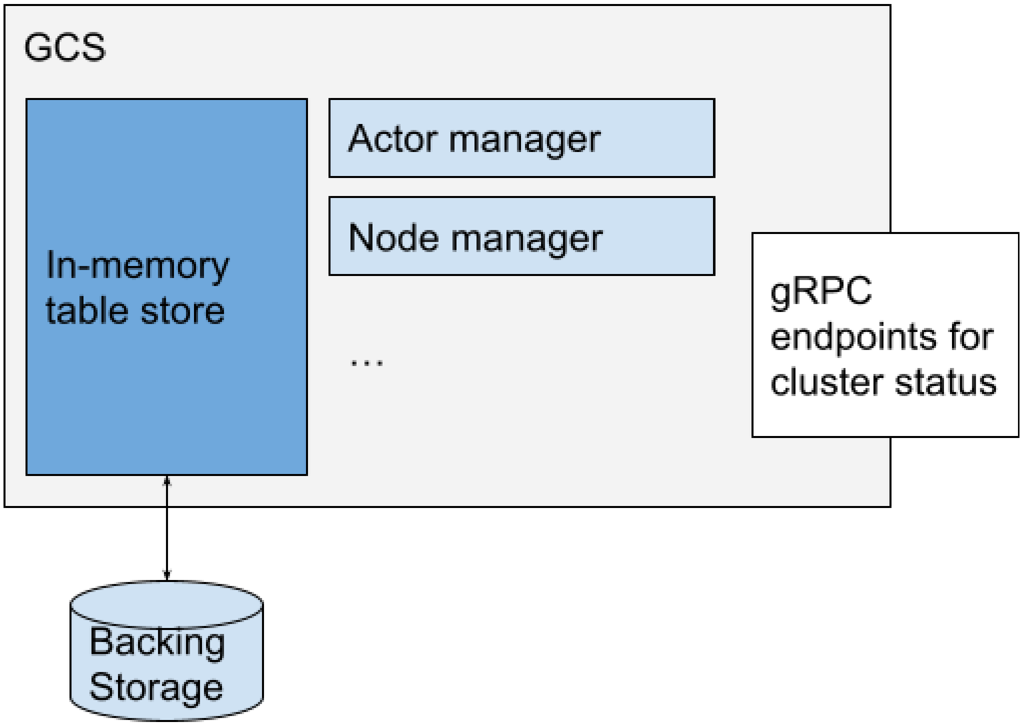
Code references:

- src/ray/core_worker/core_worker.cc
- src/ray/common/task/task_spec.h
- src/ray/core_worker/transport/direct_actor_transport.cc
- src/ray/gcs/gcs_server/gcs_actor_manager.cc
- src/ray/gcs/gcs_server/gcs_actor_scheduler.cc
- src/ray/protobuf/core_worker.proto

Global Control Service

Overview

The Global Control Service, also known as the GCS, is Ray's cluster control plane. It manages the Ray cluster and serves as a centralized place to coordinate raylets and discover other cluster processes. The GCS also serves as an entry point for external services like the autoscaler and dashboard to communicate with the Ray cluster. The GCS is currently single-threaded except for heartbeat checks and resource polling; there are ongoing efforts to scale other operations such as actor management via multithreading.



If the GCS fails, features involving the GCS won't work. These are:

- **Node management:** Manage addition and deletion of nodes to the cluster. It also broadcasts this information to all raylets so the raylets will be aware of the node changes.
- **Resource management:** Broadcast the resource availability of each raylet to the whole cluster to make sure each raylet's view of the resource usage is updated.
- **Actor management:** Manage actor creation and deletion requests. It also monitors the actor liveness and triggers recreation (if configured) in case of an actor failure.
- **Placement group management:** Coordinate placement group creation and deletion in the Ray cluster.

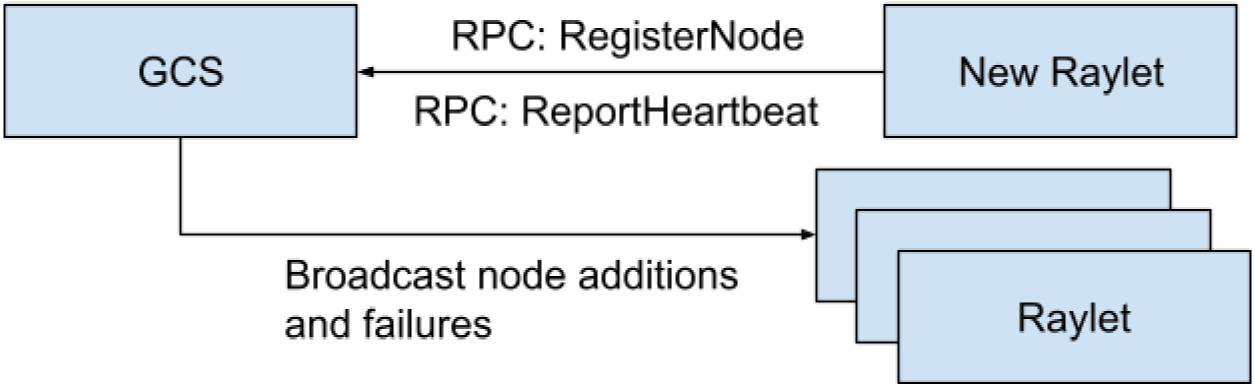
- **Metadata store:** Provide a key-value store which can be accessed by any worker. Note that this is meant for small metadata only and is not meant to be a scalable storage system. Task and object metadata, for example, are stored directly in the workers.
- **Worker manager:** Handle worker failure reported by raylet.
- **Runtime env**: GCS is the place managing the runtime env package, including counting the number of usage of the plackages and the garbage collection.

GCS also provides several gRPC endpoints to retrieve the current status of the ray cluster, like actor, worker and node information.

GCS can be optionally backed by an external storage system. By default, a simple in-memory hash map is used, but it can be configured to write-through to a Redis store.

Node management

When a raylet starts, it registers with the GCS. The GCS writes the raylet's information to storage. Once the raylet is registered to the GCS, the event is broadcasted to all other raylets.



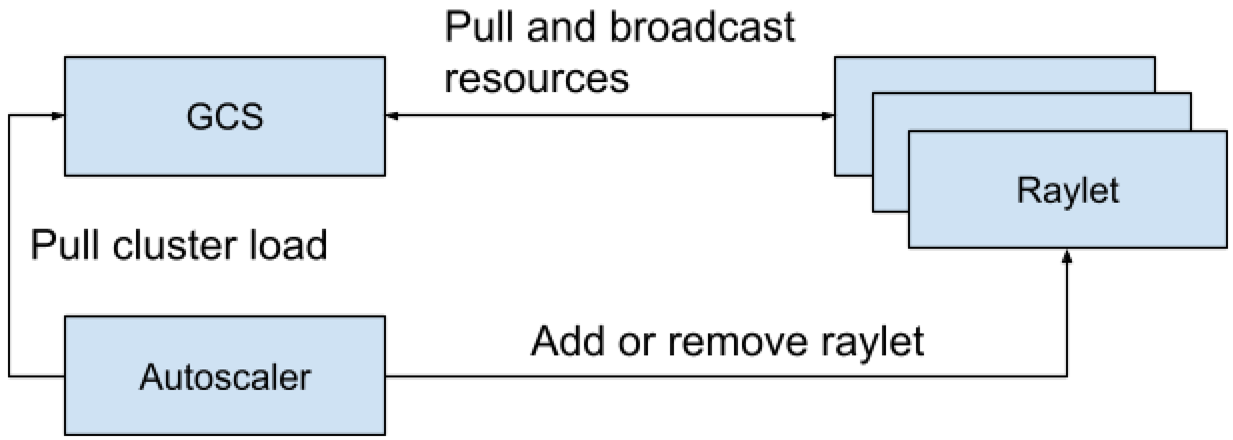
After the node is registered, the GCS monitors the liveness of the raylet by doing periodic health checks. The GCS also pulls the resource view of the raylet and broadcasts it to the other raylets. If the raylet fails, the GCS also broadcasts the death of the raylet to the cluster. Once raylets receive the information, they clean up any related state as needed.

Raylets also report deaths of any local worker processes to the GCS so it can be broadcasted to the other raylets. This is used to clean up any related system state, e.g., killing tasks that were submitted by that worker.

Code references:

- src/ray/gcs/gcs_server/gcs_node_manager.cc
- src/ray/gcs/gcs_server/gcs_heartbeat_manager.cc

Resource management



The GCS is responsible for making sure raylets have the latest view of the resource usage in the cluster. Distributed scheduling efficiency depends on this: if the view is not fresh enough a raylet might mistakenly schedule a task to another raylet which doesn't have resources to run the task.

The GCS will pull the resource usage from registered raylets every 100ms by default. It also broadcasts the global view to all the raylet every 100ms.

The GCS is also the entrypoint for the autoscaler to get the current cluster load. The autoscaler uses this to allocate or remove nodes from the cluster.

Code references:

- src/ray/gcs/gcs_server/gcs_resource_manager.cc
- src/ray/gcs/gcs_server/gcs_resource_report_poller.cc
- src/ray/gcs/gcs_server/ray_syncer.h

Actor management

The GCS plays an important role in actor management. All actors need to be registered in the GCS first before being scheduled. GCS is also the owner of the detached actors. Refer to actor management for more information.

Code references:

- src/ray/gcs/gcs_server/gcs_actor_manager.cc
- src/ray/gcs/gcs_server/gcs_actor_scheduler.cc

Placement group management

The GCS also manages the lifecycle of the placement groups. The GCS implements a two phase commit protocol to create placement groups. For more details, please check the placement groups section.

Code reference:

- src/ray/gcs/gcs_server/gcs_placement_group_manager.cc
- src/ray/gcs/gcs_server/gcs_placement_group_scheduler.cc

Metadata store

The GCS stores certain cluster-level metadata in an internal key-value store. This includes:

- The cluster's dashboard address.
- Remote function definitions. Whenever a remote function is defined in the language frontend, the Ray worker process will check whether it's stored in the GCS and add it if not. The worker assigned to run a task then loads the function definition from the GCS.
- Runtime environment data. By default, the runtime environment's working dir is stored in the GCS. The GCS garbage collects runtime environments by counting the number of detached actors and jobs using the environment.
- Some Ray libraries also use the GCS to store metadata. For example, Ray Serve stores deployment metadata in the GCS.

Code reference:

- src/ray/gcs/gcs_server/gcs_kv_manager.cc

Fault tolerance

The GCS is a critical component of the Ray cluster, and if it fails, all functions mentioned above will not work.

In Ray v2.0, the GCS can optionally recover from failure. However, during recovery the above functions will be temporarily unavailable.

By default, the GCS stores everything into an in-memory store, which is lost upon failure. To make the GCS fault-tolerant, it has to write the data into a persistent store. Ray supports Redis as an external storage system. To support GCS fault tolerance, the GCS should be backed by an HA Redis instance. Then, when the GCS restarts, it loads the information from the Redis store first, including any Raylets, actors, and placement groups that were running in the cluster at the time of the failure. Then the GCS will resume regular functions like health checks and resource management.

Thus, the following features won't be available while the GCS recovers:

- Actor creation/deletion/reconstruction
- Placement group creation/deletion/reconstruction
- Resource management
- New raylets won't be able to register
- New Ray workers won't be able to start

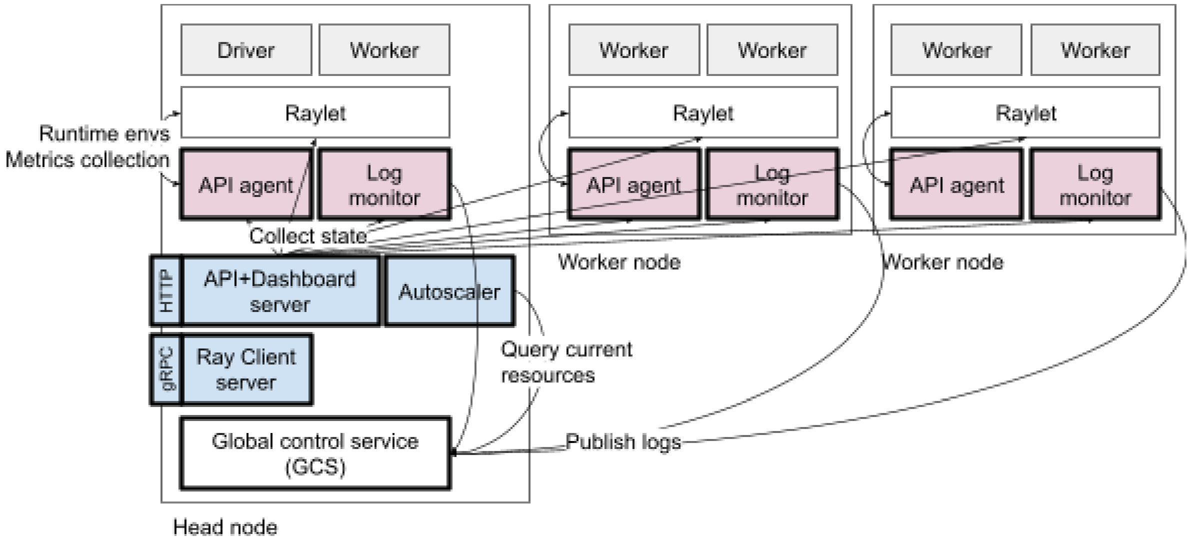
Note that any running Ray tasks and actors will remain alive since these components do not require reading or writing the GCS. Similarly, any existing objects will continue to be available.

Code references:

- src/ray/gcs/gcs_server/gcs_server.cc
- src/ray/protobuf/gcs.proto
- src/ray/protobuf/gcs_service.proto

Cluster Management

The previous sections described the system design for execution of Ray tasks, actors, and objects. However, realistic Ray usage requires additional operations for deploying applications, adding and removing nodes, observing cluster state, etc. Here, we describe these auxiliary operations related to managing and deploying Ray clusters.



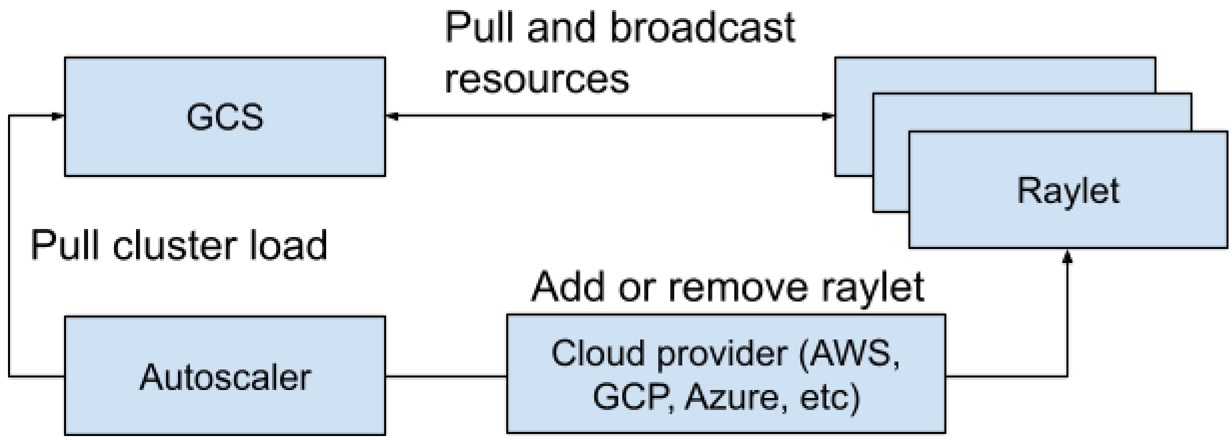
Auxiliary processes involved in cluster management. Blue processes are singletons that live on the head node. Pink processes are launched per-node and manage auxiliary operations for their local node.

Here is a summary of the auxiliary processes that Ray clusters launch and the process name

	Process name	Role
Autoscaler (also known as the cluster launcher)	autoscaler/_private/monitor.py	Adds and removes nodes based on cluster resources and utilization.
Ray Client server	ray.util.client.server	Proxy server for Ray Client, which is used for interactive development on a cluster.
API server (also known as the Dashboard server)	dashboard/dashboard.py	Previously the main role was to host the dashboard server, but today it is also the main endpoint for API calls to the cluster, including job submission and the state API.
API agent (also known as Ray agent)	dashboard/agent.py	Collects metrics on the local node for cluster-level aggregation, installs runtime environments for task and actor execution.
Log monitor	log_monitor.py	Monitors local logs (by default stored in /tmp/ray/session_latest/logs) and publishes errors to the driver.

Autoscaler

The Ray Autoscaler is responsible for adding and removing nodes from the cluster. It looks at the logical resource demands exposed by the distributed scheduler, the nodes currently in the cluster, the node profiles of the cluster, calculates the desired cluster configuration and performs the operations to move the cluster to the desired state.


Pull and broadcast resources
GCS
Raylet
Pull cluster load
Add or remove raylet
Autoscaler
Cloud provider (AWS, GCP, Azure, etc)

Autoscaler pulls current cluster load from the GCS and invokes cloud providers to add or remove machines

The autoscaling loop works as follows:

- The application submits tasks, actors, placement groups, which requests resources such as cpu.
- The scheduler looks at the demand and availability and makes a decision to place the task for execution or puts it into pending if it cannot be satisfied. This information is snapshotted into GCS.
- The autoscaler, running as a separate process, will periodically fetch the snapshot from GCS. It looks at the resources available in the cluster, resources requested, what is pending, the worker node configuration specified for the cluster, and runs a bin-packing algorithm to calculate the number of nodes to satisfy both running and pending tasks, actors, and placement group requests.
- The autoscaler then adds or removes nodes from the cluster via the node provider interface. The node provider interface allows Ray to plug into different cloud providers (e.g. AWS, GCP, Azure), cluster managers (e.g. Kubernetes), or on-premise data centers.
- When the node comes up it registers with the cluster and accepts application workload.

Downscaling

If a node is idle for a timeout (5 minutes by default), it is removed from the cluster. A node is considered idle when there are no active tasks, actors, or primary copies of the objects.

Upscaling speed

There is a limit on the number of pending nodes, determined by the upscaling speed. The speed is defined as the ratio of the number of nodes pending to the current number of nodes. The higher the value, the more aggressive upscaling will be. For example, if this is set to 1.0, the cluster can grow in size by at most 100% at any time, so if the cluster currently has 20 nodes, at most 20 pending launches are allowed. The minimum number of pending nodes is 5 to ensure sufficient upscaling speed even for small clusters.

Heterogeneous node types

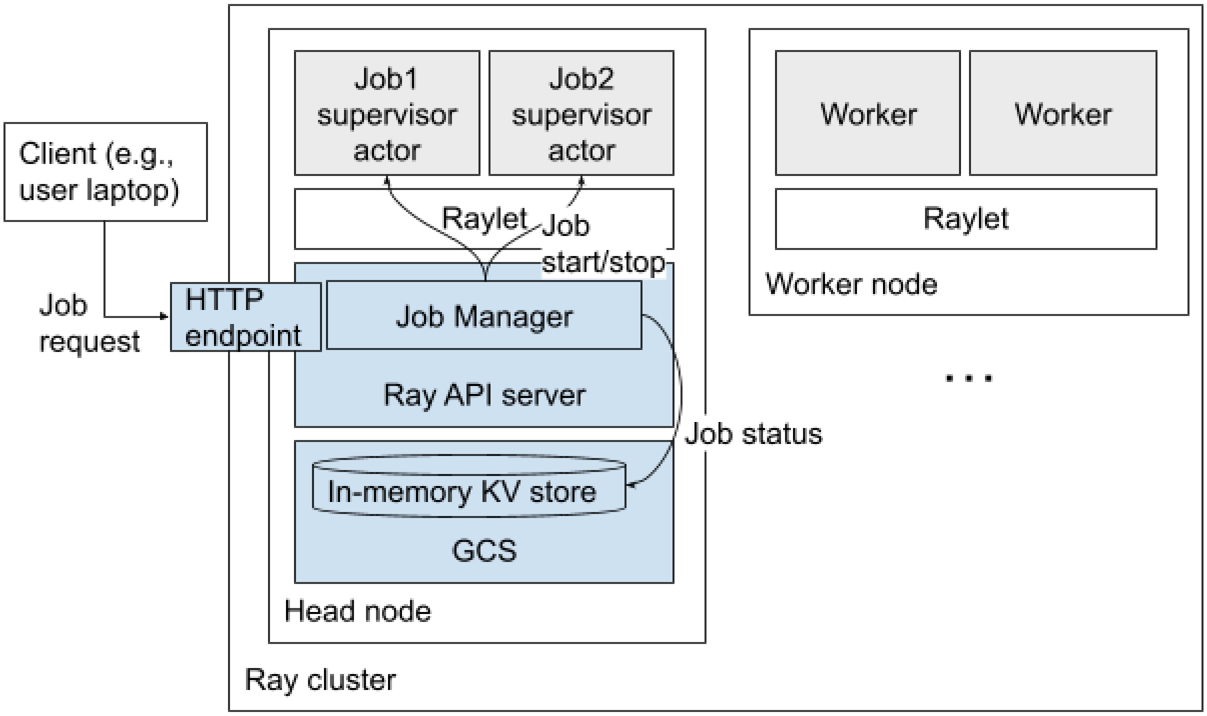
Ray also supports multiple cluster node types. The concept of a cluster node type encompasses both the physical instance type (e.g., AWS p3.8xl GPU nodes vs m4.16xl CPU nodes), as well as other attributes (e.g., IAM role, the machine image, etc). Custom resources can be specified for each node type so that Ray is aware of the demand for specific node types at the application level (e.g., a task may request to be placed on a machine with a specific role or machine image via custom resource).

Code references:

- python/ray/autoscaler/_private/autoscaler.py
- python/ray/autoscaler/_private/resource_demand_scheduler.py
- python/ray/autoscaler/node_provider.py

Job Submission

Jobs can be submitted to the Ray cluster via a command line interface (CLI), a Python SDK, or a REST API. The CLI calls into the Python SDK, which in turn makes HTTP requests to the Jobs REST API server on the Ray cluster. The REST API is currently hosted in the Ray dashboard backend, but may be moved to a standalone API server in the future.



A diagram of the architecture of Ray Job Submission. Blue boxes indicate singleton services that manage job submission, among other cluster-level operations.

Each job is managed by its own dedicated job supervisor actor that runs on the Ray head node. This actor runs in the job's user-specified runtime environment, and the job's user-specified entrypoint command is run in a subprocess that inherits this runtime environment. If this command contains a Ray script, the Ray script will attach to the running Ray cluster.

Jobs report structured statuses (e.g. PENDING, RUNNING, SUCCEEDED) and messages that can be fetched via the API. These are stored in the GCS.

The job supervisor actor and the subprocess it starts fate-share with each other. If the job actor dies, the latest job status remains. The status is updated to FAILED the next time the user requests the job status.

Stopping a job happens asynchronously by setting a "stop" event on the corresponding job supervisor actor. This actor is responsible for terminating the job subprocess, updating its status, then exiting.

The job manager manages job supervisor actors and logs. The output of the entrypoint script is written directly to a file on the head node, and this file can be read or streamed via the HTTP endpoint.

Job Submission is in beta as of Ray 2.0. A key step on the roadmap for exiting beta is enabling job supervisor actors to be scheduled on nodes other than the head node to reduce pressure on the head node in multi-tenant setups.

Runtime Environments and Multitenancy

A runtime environment defines dependencies such as files, packages, environment variables needed for a Python script to run. It is installed dynamically on the cluster at runtime, and can be specified for a Ray job, or for specific actors and tasks.

Installation and deletion of runtime environments are handled by a RuntimeEnvAgent gRPC server that runs on each node. The RuntimeEnvAgent fate-shares with the raylet to simplify the failure model and because it is a core component for scheduling tasks and actors.

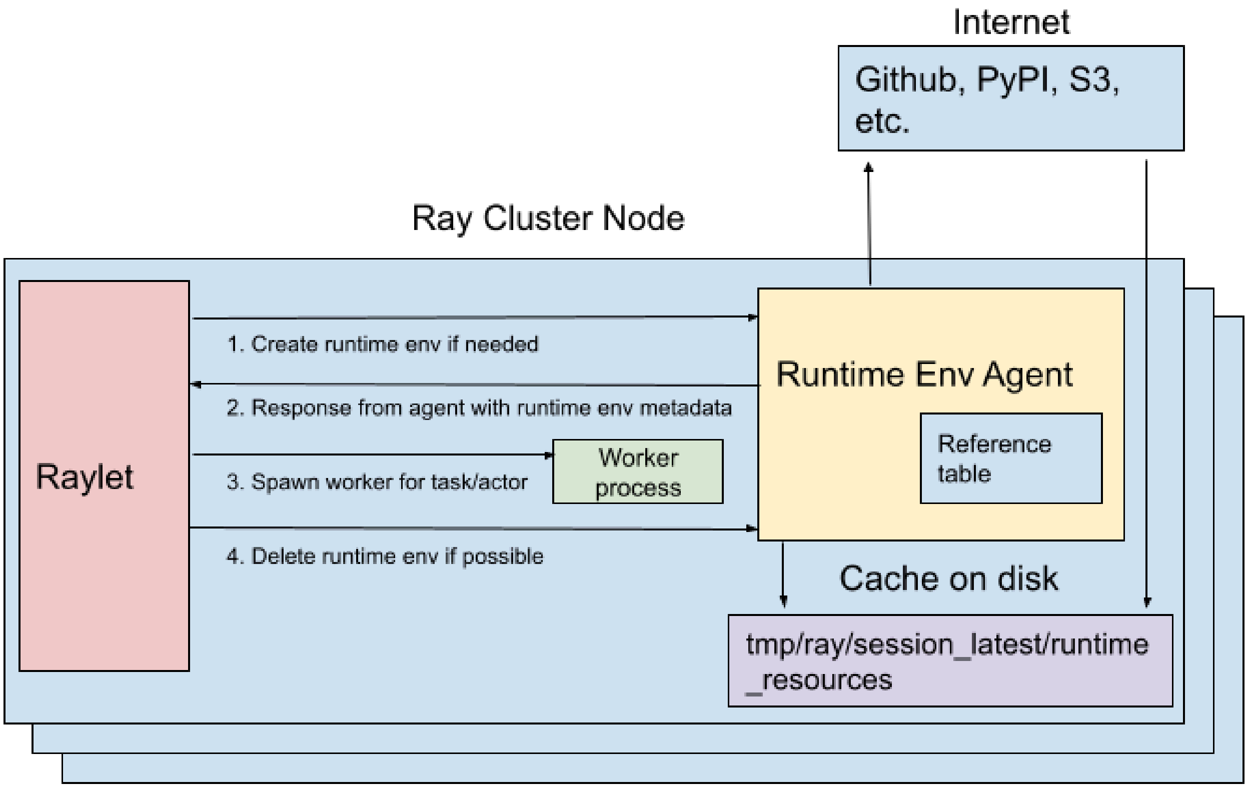
When a task or actor requires a runtime environment, the raylet sends a gRPC request to the RuntimeEnvAgent to create the environment if it does not already exist.

Creating an environment may entail:

- Downloading and installing packages via `pip install`
- Setting environment variables for a Ray worker process
- Calling `conda activate` before starting a Ray worker process
- Downloading files from remote cloud storage

Runtime environment resources such as downloaded files and installed conda environments are cached on each node so that they can be shared between different tasks, actors and jobs. When the cache size limit is exceeded, resources not currently used by any actor, task or job will be deleted.

In ongoing work, support is being added for user-defined third-party plugins for setup and installation of custom resources (such as PEX files, for example).


Internet
Github, PyPI, S3, etc.
Ray Cluster Node
Raylet
1. Create runtime env if needed
2. Response from agent with runtime env metadata
3. Spawn worker for task/actor
Worker process
4. Delete runtime env if possible
Runtime Env Agent
Reference table
Cache on disk
tmp/ray/session_latest/runtime_resources

An overview of the lifecycle of a runtime environment.

KubeRay

The KubeRay operator manages Ray clusters in a Kubernetes setting. Each Ray node runs as a Kubernetes pod. KubeRay follows the Kubernetes Operator Pattern:

- A custom resource, called a *RayCluster*, describes the desired state of a Ray cluster.
- A custom controller, the *KubeRay operator*, manages Ray pods in order to match the RayCluster's specification.

See the docs on KubeRay for more information.

Ray Observability

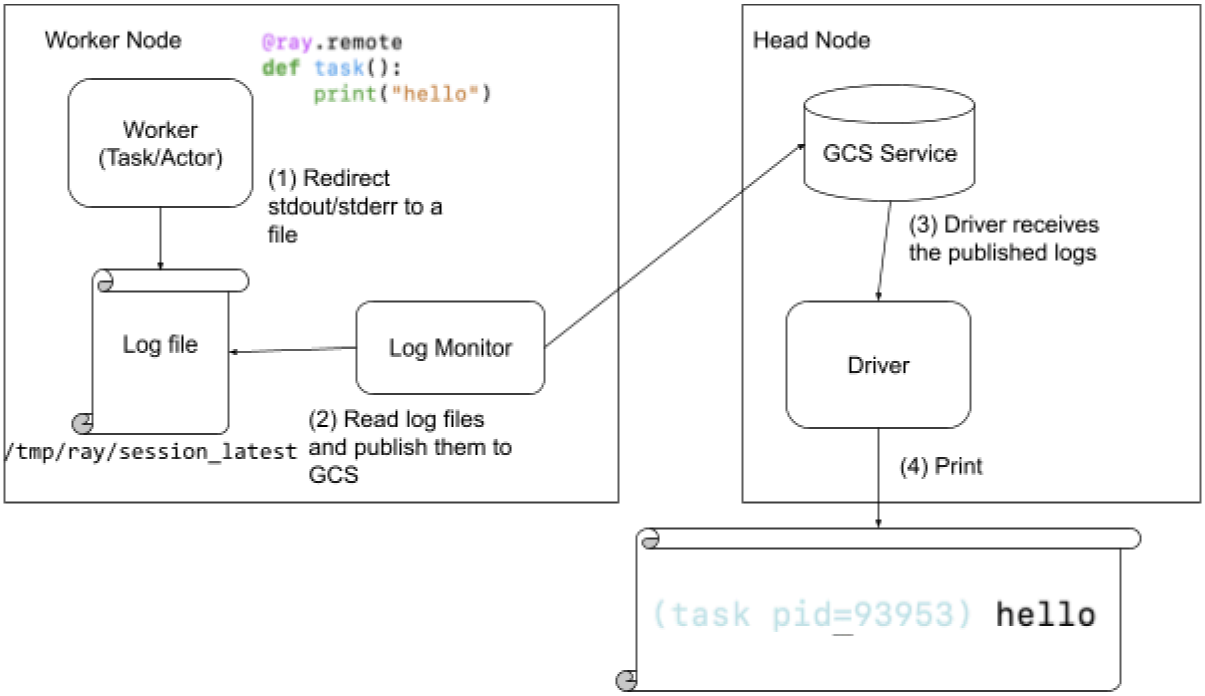
Ray provides various tools and features to give more visibility into the cluster.

Ray Dashboard

Ray provides a built-in dashboard that runs as an HTTP server on the head node. The Ray dashboard periodically aggregates system state from the cluster, organizes and stores data, and provides the web UI to visualize the cluster state.

Log Aggregation

The Ray driver aggregates and prints all log messages printed from actors and tasks. When a task or actor prints logs to its stdout or stderr, they are automatically redirected to the corresponding worker log file. A Python process known as a log monitor runs on each node. The log monitor periodically reads local Ray log files and publishes the log messages to the driver program via GCS pubsub.



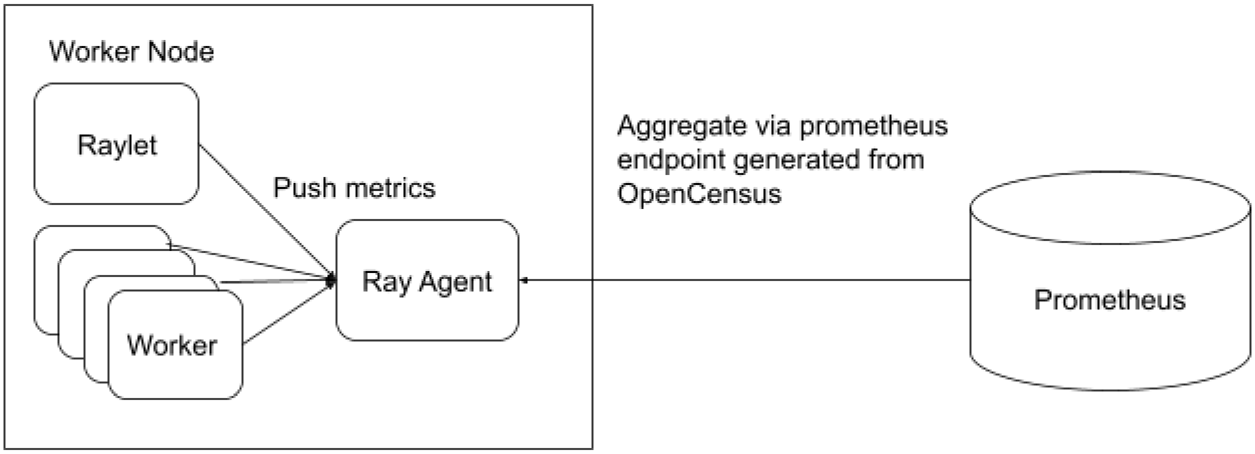
```
@ray.remote
def task():
    print("hello")
```

```
(task pid=93953) hello
```

Worker Node
Worker (Task/Actor)
(1) Redirect stdout/stderr to a file
Log file
/tmp/ray/session_latest
Log Monitor
(2) Read log files and publish them to GCS
Head Node
GCS Service
(3) Driver receives the published logs
Driver
(4) Print

Metrics

Ray has native integration with OpenCensus and supports export to Prometheus by default. All Ray components (GCS Service, Raylet, Workers) emit metrics to their local Ray agent process. Each Ray agent exposes the metrics via OpenCensus (by default as a Prometheus endpoint).



Ray State API

Since Ray 2.0, Ray supports state APIs that allow users to conveniently access snapshots of the current Ray state through a CLI or the Python SDK. State APIs support summaries and queries of specific Ray tasks, actors, etc.

Systems like Kubernetes that support similar APIs often write such execution metadata, e.g., the current pods, to a persistent key-value store or database. In contrast, Ray does not persist execution information such as currently running tasks. This is because Ray tasks, objects, etc. are much lighter-weight, and the cost of writing such metadata to a database would be prohibitively high. Instead, the execution metadata is distributed across workers via the ownership model, and the metadata fate-shares with the workers. Thus, to perform a query such as “list all tasks”, the state APIs must query the data from the data source (e.g., workers) on demand.

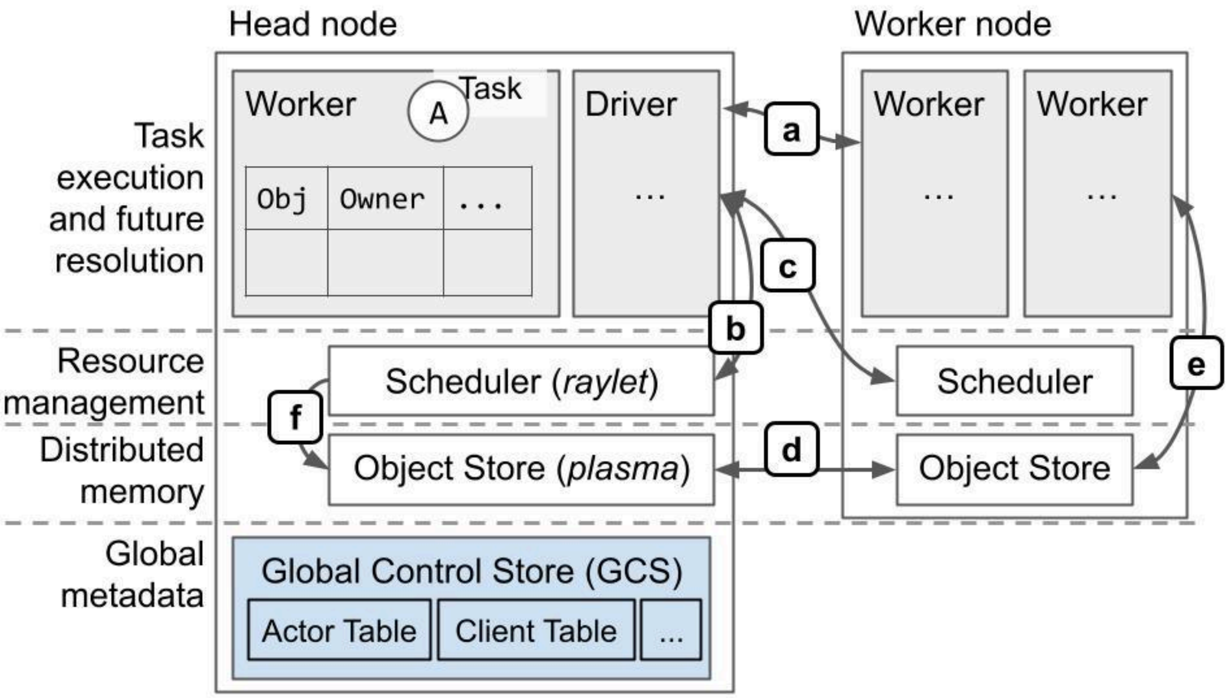
Currently, the API server collects the following information from these data sources:

- Raylet
  - Tasks
  - Objects
- Ray agent
  - Runtime environments
- GCS
  - Actors
  - Nodes
  - Placement groups
  - Worker (processes)

Appendix

Below are more detailed diagrams and examples of the system architecture.

Architecture diagram



Architecture and protocols

Protocol overview (mostly over gRPC):

a. Task execution, object reference counting.
b. Local resource management.
c. Remote/distributed resource management.
d. Distributed object transfer.
e. Storage and retrieval of large objects. Retrieval is via `ray.get` or during task execution, when replacing a task's ObjectID argument with the object's value.
f. Scheduler fetches objects from remote nodes to fulfill dependencies of locally queued tasks.

Example of task scheduling and object storage

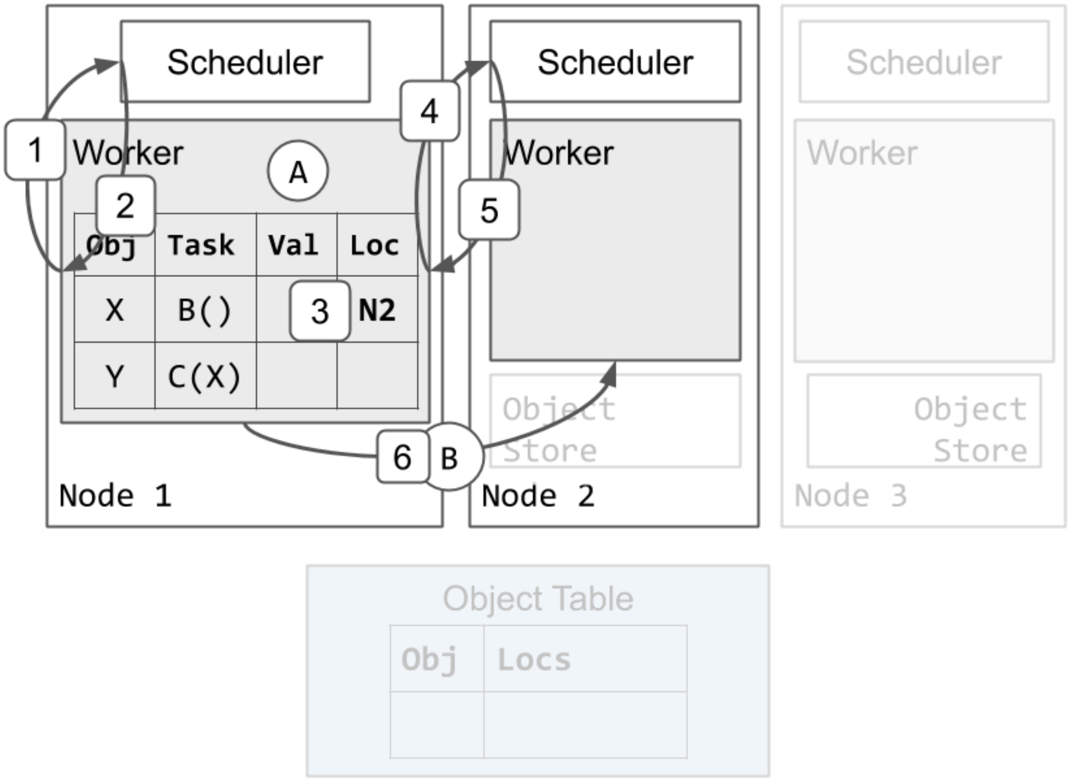
We'll walk through the physical execution of a Ray program that looks like this:

```python
@ray.remote
def A():
    y_id = C.remote(B.remote())
    y = ray.get(y_id)
```

In this example, task A submits tasks B and C, and C depends on the output of B. For illustration purposes, let's suppose that B returns a large object X, and C returns a small object Y. This will allow us to show the difference between the in-process and shared-memory object stores. We'll also show what happens if tasks A, B, and C all execute on different nodes, to show how distributed scheduling works.

Distributed task scheduling

We’ll start off with worker 1 executing A. Tasks B and C have already been submitted to worker 1. Thus, worker 1’s local *ownership table* already includes entries for both X and Y. First, we’ll walk through an example of scheduling B for execution:



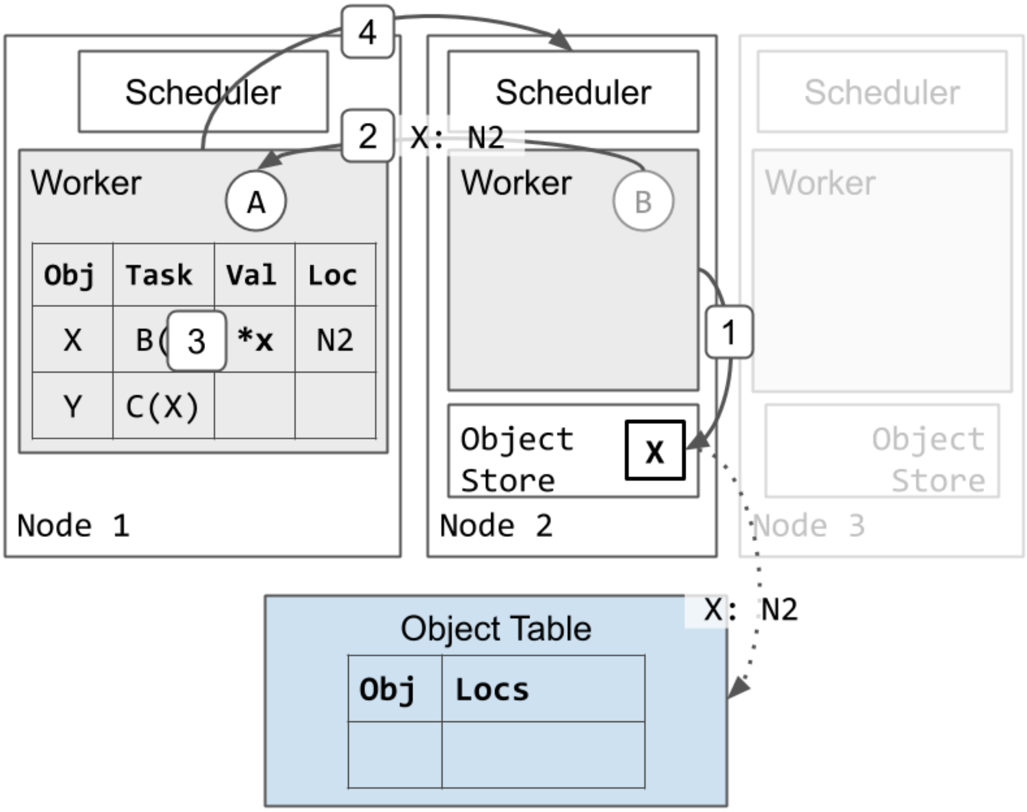
Obj	Task	Val	Loc
X	B()		N2
Y	C(X)		

Obj	Locs

1. Worker 1 asks its local scheduler for resources to execute B.
2. Scheduler 1 responds, telling worker 1 to retry the scheduling request at node 2.
3. Worker 1 updates its local ownership table to indicate that task B is pending on node 2.
4. Worker 1 asks the scheduler on node 2 for resources to execute B.
5. Scheduler 2 grants the resources to worker 1 and responds with the address of worker 2. Scheduler 2 ensures that no other tasks will be assigned to worker 2 while worker 1 still holds the resources.
6. Worker 1 sends task B to worker 2 for execution.

Task execution

Next, we'll show an example of a worker executing a task and storing the return value in the distributed object store:



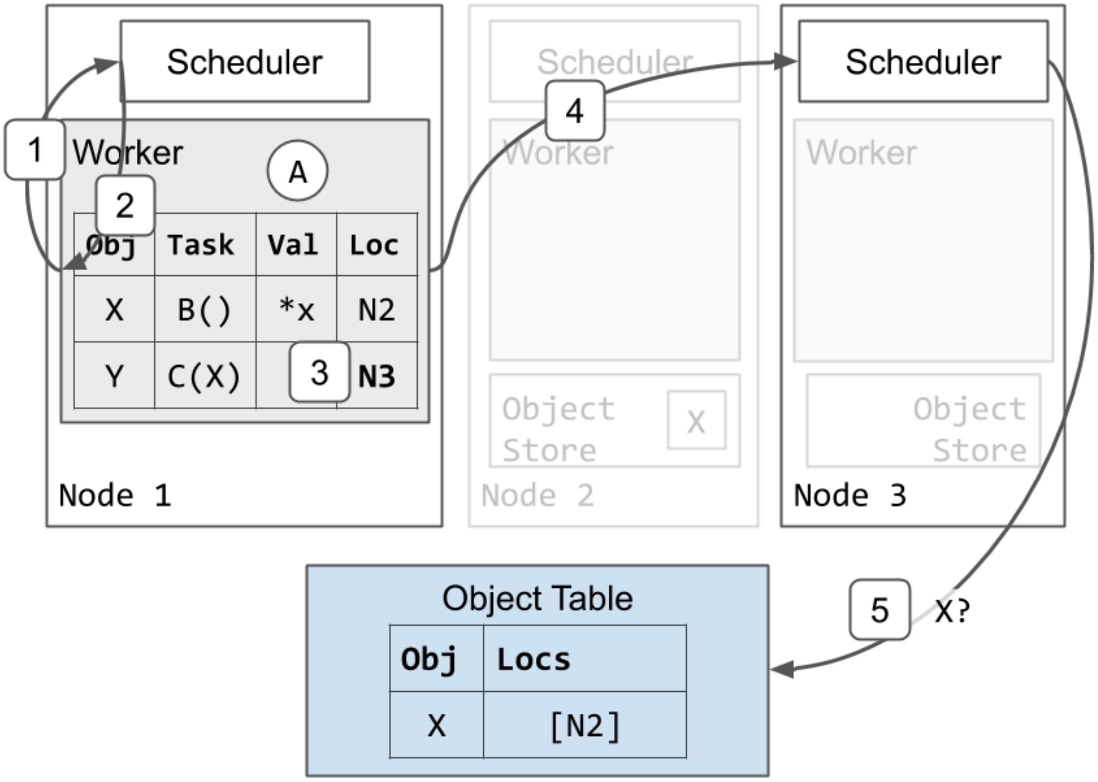
Obj	Task	Val	Loc
X	B(	*x	N2
Y	C(X)		

Obj	Locs

1. Worker 2 finishes executing B and stores the return value X in its local object store.
    a. Node 2 asynchronously updates the object table to indicate that X is now on node 2 (dotted arrow).
    b. Since this is the first copy of X to be created, node 2 also pins its copy of X until worker 1 notifies node 2 that it is okay to release the object (not shown). This ensures that the object value is reachable while it is still in reference.
2. Worker 2 responds to worker 1 indicating that B has finished.
3. Worker 1 updates its local ownership table to indicate that X is stored in distributed memory.
4. Worker 1 returns the resources to scheduler 2. Worker 2 may now be reused to execute other tasks.

Distributed task scheduling and argument resolution

Now that B has finished, task C can start execution. Worker 1 schedules C next, using a similar protocol as for task B:

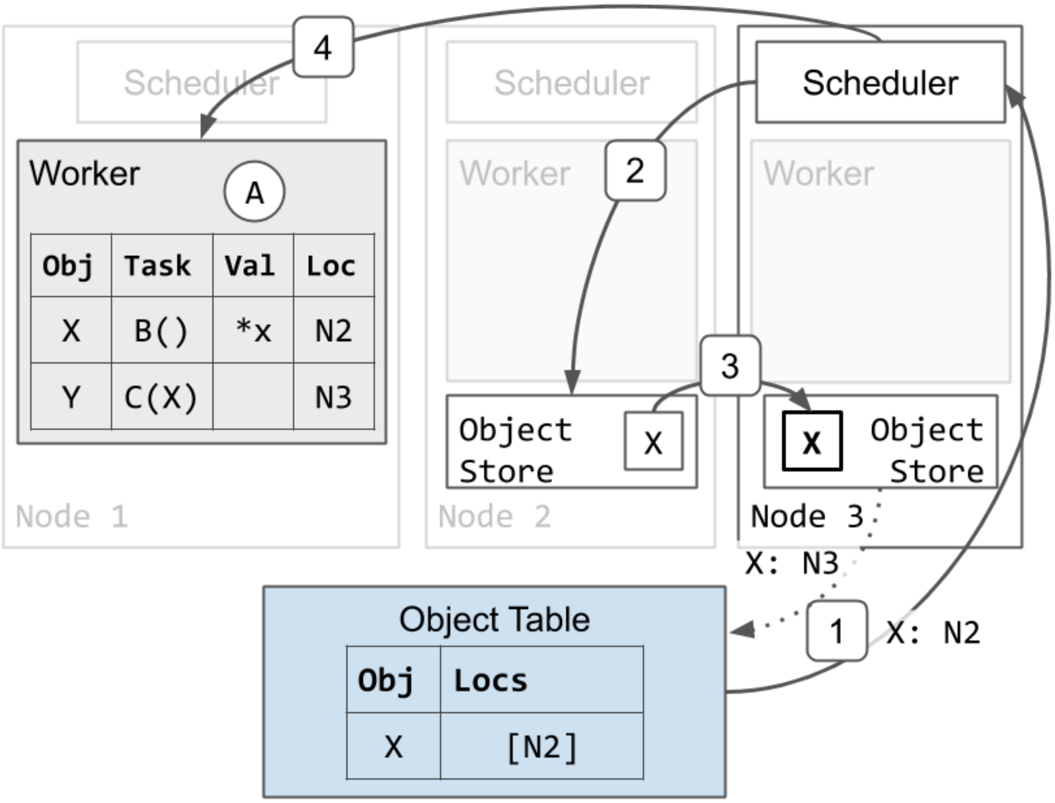


Obj	Task	Val	Loc
X	B()	*x	N2
Y	C(X)		N3

Obj	Locs
X	[N2]

1. Worker 1 asks its local scheduler for resources to execute C.
2. Scheduler 1 responds, telling worker 1 to retry the scheduling request at node 3.
3. Worker 1 updates its local ownership table to indicate that task C is pending on node 3.
4. Worker 1 asks the scheduler on node 3 for resources to execute C.
5. Scheduler 3 sees that C depends on X, but it does not have a copy of X in its local object store. Scheduler 3 queues C and asks the object table for a location for X.

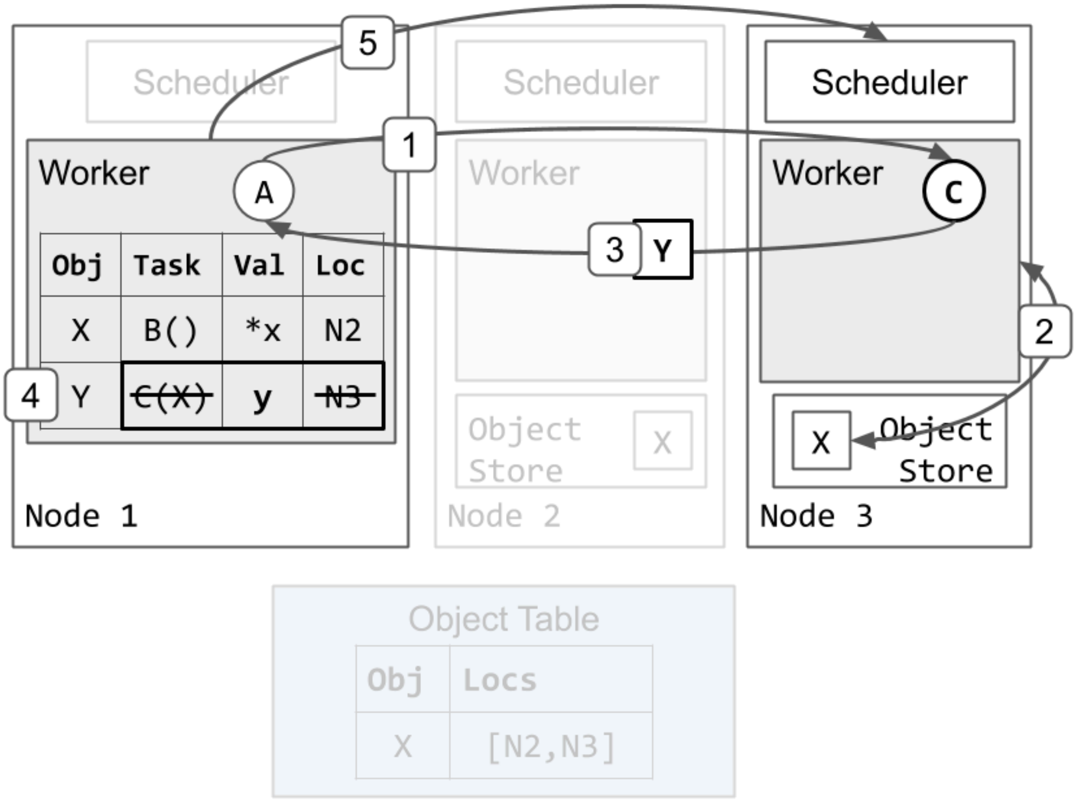
Task C requires a local copy of X to begin execution, so node 3 fetches a copy of X:



1. Object table responds to scheduler 3 indicating that X is located on node 2.
2. Scheduler asks object store on node 2 to send a copy of X.
3. X is copied from node 2 to node 3.
    a. Node 3 also asynchronously updates the object table to indicate that X is also on Node 3 (dotted arrow).
    b. Node 3's copy of X is cached but not pinned. While a local worker is using it, the object will not be evicted. However, unlike the copy of X on node 2, node 3's copy may be evicted according to LRU when object store 3 is under memory pressure. If this occurs and node 3 later needs the object again, it can re-fetch it from node 2 or a different copy using the same protocol shown here.
4. Since node 3 now has a local copy of X, scheduler 3 grants the resources to worker 1 and responds with the address of worker 3.

Task execution and object inlining

Task C executes and returns an object small enough to be stored in the in-process memory store:



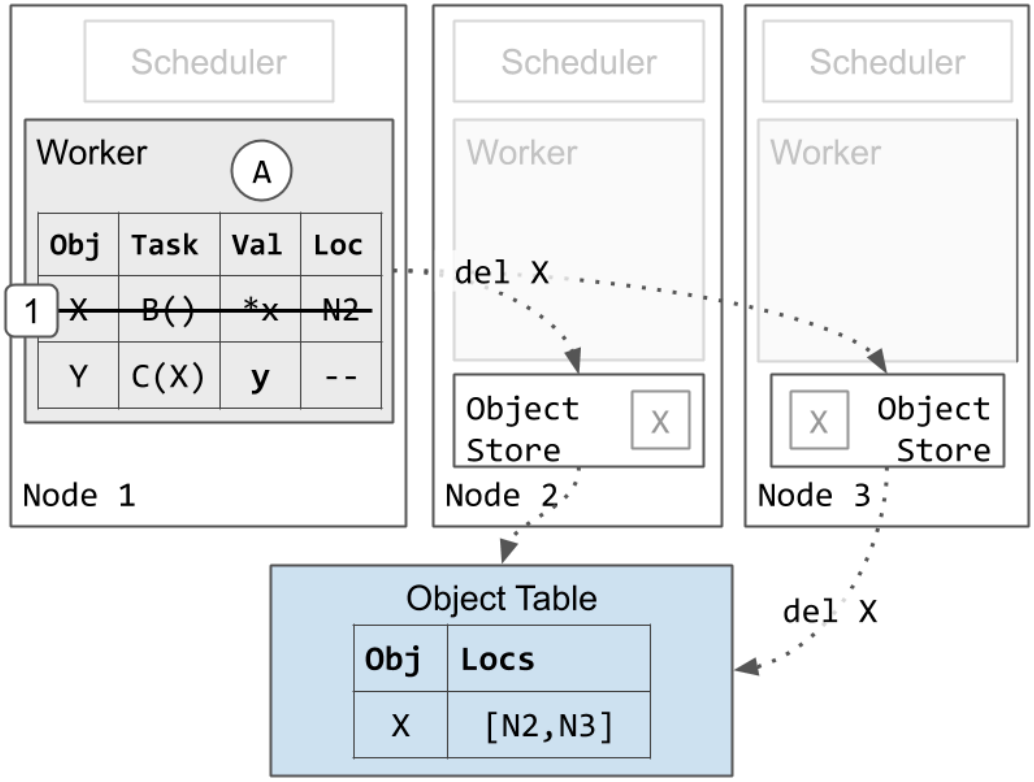
Obj	Task	Val	Loc
X	B()	*x	N2
Y	~~C(X)~~	y	~~N3~~

Obj	Locs
X	[N2,N3]

1. Worker 1 sends task C to worker 3 for execution.
2. Worker 3 gets the value of X from its local object store (similar to a `ray.get`) and runs C(X).
3. Worker 3 finishes C and returns Y, this time by value instead of storing it in its local object store.
4. Worker 1 stores Y in its in-process memory store. It also erases the specification and location of task C, since C has finished execution. At this point, the outstanding `ray.get` call in task A will find and return the value of y from worker 1's in-process store.
5. Worker 1 returns the resources to scheduler 3. Worker 3 may now be reused to execute other tasks. This may be done before step 4.

Garbage collection

Finally, we show how memory is cleaned up by the workers:



Obj	Task	Val	Loc
~~X~~	~~B()~~	~~*x~~	~~N2~~
Y	C(X)	y	--

Obj	Locs
X	[N2,N3]

1. Worker 1 erases its entry for object X. This is safe to do because the pending task C had the only reference to X and C has now finished. Worker 1 keeps its entry for Y because the application still has a reference to y's ObjectID.
    a. Eventually, all copies of X are deleted from the cluster. This can be done at any point after step 1. As noted above, node 3's copy of X may also be deleted before step 1, if node 3's object store is under memory pressure.