Ray 1.x Architecture

Ray Team, September 2020

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 1.0+). 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

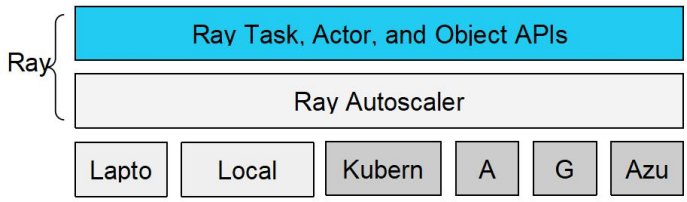
This document is not meant as an introduction to Ray. For that and any further questions that arise from this document, please refer to A Gentle Introduction to Ray, the Ray GitHub, and the Ray Slack. You may also want to check out the list of common Ray Design Patterns. This document supersedes previous papers describing Ray; in particular, the underlying architecture has changed considerably from versions 0.7 to 0.8.

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 provides a universal distributed API that is suitable both for building entirely new apps and scaling existing ones.

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 Tune, RLlib, RaySGD), online serving (e.g., Ray Serve, online learning use cases), data processing (e.g., Modin, Dask-on-Ray, MARS-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. 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 and distributed memory sharing (via a shared memory object store).

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 spillback 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

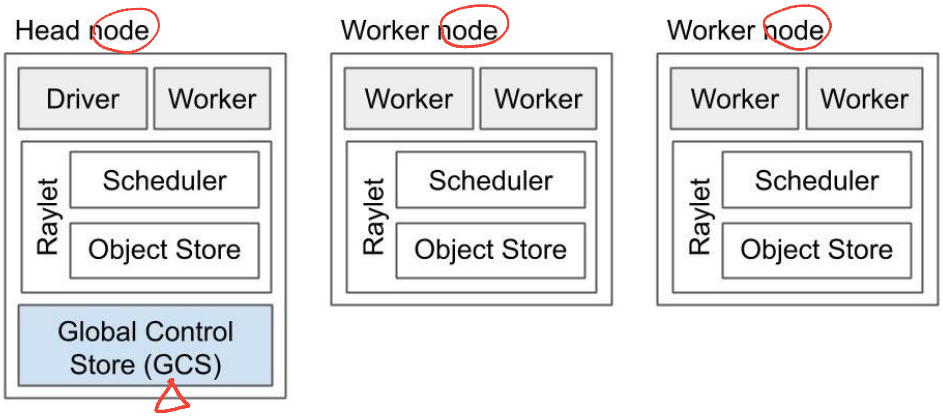
	through libraries only (e.g., Modin, Dask-on-Ray, MARS-on-Ray).
Actor Frameworks	Unlike specialized actor frameworks such as Erlang and Akka, Ray integrates with existing programming languages, enabling cross language operation and the use of language native libraries. The Ray system also transparently 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).

Architecture Overview

Application concepts

- *Task* - A single function invocation that executes on a process different from the caller. 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 an `ObjectRef` that can be used to retrieve the return value.
- *Object* - An application value. This may be 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.
- *Driver* - The program root. This is the code that runs `ray.init()`.
- *Job* - The collection of tasks, objects, and actors originating (recursively) from the same driver.

Design

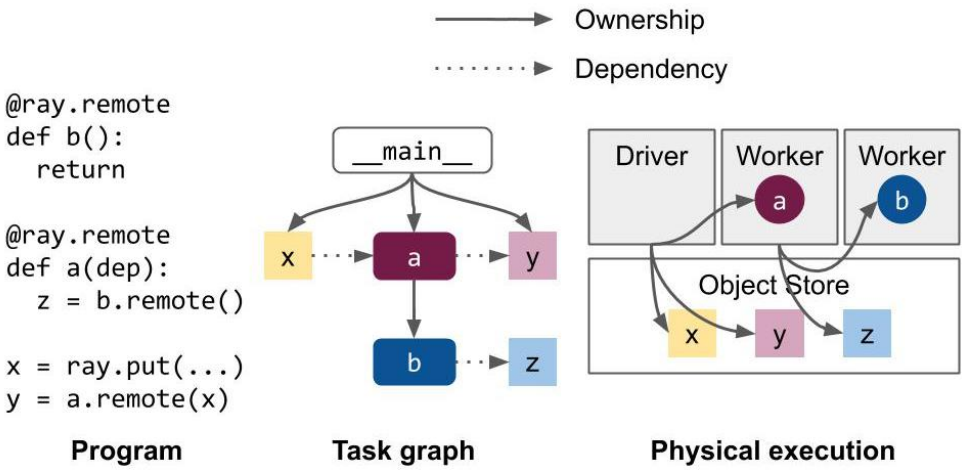


A Ray cluster.

A Ray cluster comprises a set of worker nodes and a centralized Global Control Store (GCS) instance.

Some system metadata is managed by the GCS, a server backed by a pluggable data store. This metadata is also cached locally by the workers, e.g., the address of an actor. The GCS manages metadata that is less frequently accessed but likely to be used by most or all workers in the cluster, e.g., the current node membership of the cluster. This is to ensure that GCS performance is not critical to application performance.

Ownership



Most of the system metadata is managed according to a decentralized concept called *ownership*: Each worker process manages and *owns* the tasks that it submits and the `ObjectRef`s returned by those tasks. The owner is responsible for ensuring execution of the

task and facilitating the resolution of an `ObjectRef` to its underlying value. Similarly, a worker owns any objects that it created through a `ray.put` call.

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.

Components

A Ray instance 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 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 is shared among all jobs on the same cluster. The raylet has two main components, run on separate threads:
    a. A **scheduler**. Responsible for resource management 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 and transferring large objects. The individual object stores in a cluster comprise the Ray **distributed object store**.

Each worker process and raylet is assigned a unique 20-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 Store (GCS)**. The GCS is a key-value server that contains system-level metadata, such as the locations of objects and actors. There is an ongoing effort to support high availability for the GCS, so that it may 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. Driver processes can run on any node.

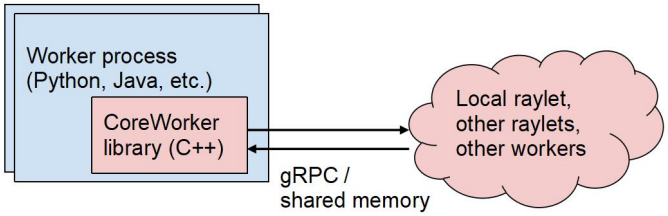
Connecting to Ray

An application driver can connect to Ray in one of the following ways:

1. Calling `ray.init()` with no arguments. This launches an embedded, single-node Ray instance that is immediately available to the application.
2. Connecting to an existing Ray cluster by specifying `ray.init(address=<GCS addr>)`. Under the hood, the driver will connect to the GCS at the specified address and lookup the addresses of other components of the cluster, e.g., its local raylet address. The driver must be co-located with one of the existing nodes of the Ray cluster. This is required due to the shared-memory capabilities of Ray.
3. Using the Ray client `ray.util.connect()` to connect from a remote machine (e.g., laptop). By default, each Ray cluster launches with a Ray client server running on the head node that can receive remote client connections. Note however that when the client is located remotely, some operations run directly from the client may be slower due to WAN latencies.

Language Runtime

All Ray core components are implemented in C++. Ray supports both Python and Java 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.



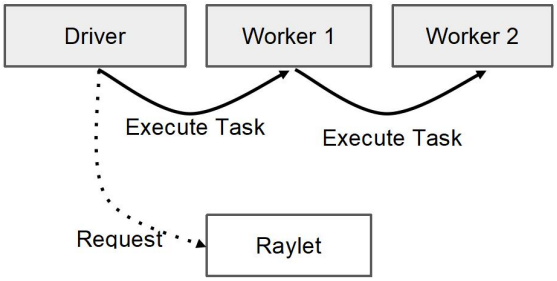
Ray workers interact with other Ray processes through the CoreWorker library.

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 the raylet to execute the task. Here, the driver owns the result of `A`, and `Worker 1` owns the result of `B`.

When a task is submitted, the owner waits for any dependencies, i.e. `ObjectRef`s that were passed as an argument to the task (see Lifetime of an Object), 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. When the dependencies are ready, the owner requests resources from the distributed scheduler to execute the task. Once resources 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. This allows the owner to refer to the objects without having to fetch the objects to its local node.

When a task is submitted with an `ObjectRef` as its argument, the object value must be resolved before the worker can begin execution. If the value is small, it is copied directly from the owner's in-process object store into the task description, where it can be referenced by the

[1] Less than 100KiB by default.

executing worker. If the value is large, the object must be fetched from distributed memory, so that the worker has a copy in its local shared memory store. The scheduler coordinates this object transfer by looking up the object's locations and requesting a copy from a different node.

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

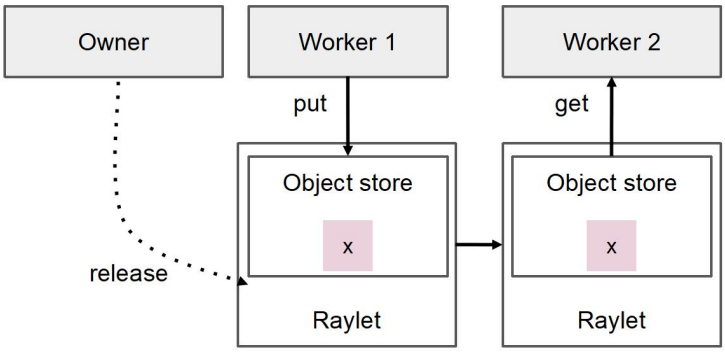
1. Application-level. This is any scenario where the worker process is alive, but the task ends in an error. For example, a task that throws an `IndexError` in Python.
2. System-level. This is any scenario where the worker process dies unexpectedly. For example, a process that segfaults, or if the worker's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

Tasks that fail due to application-level errors are never retried. The exception is caught and stored as the return value of the task. Tasks that fail due to system-level errors may be automatically retried up to a specified number of attempts.

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

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.

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 never hang but may 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. 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`.
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 failures, resolution is guaranteed to eventually succeed (but may throw an application-level exception, e.g., if the worker segfaults). If there are failures, resolution may throw a system-level exception 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 also provides an option to automatically recover such lost objects through reconstruction. An object can also fail if its owner process dies.

Code references:

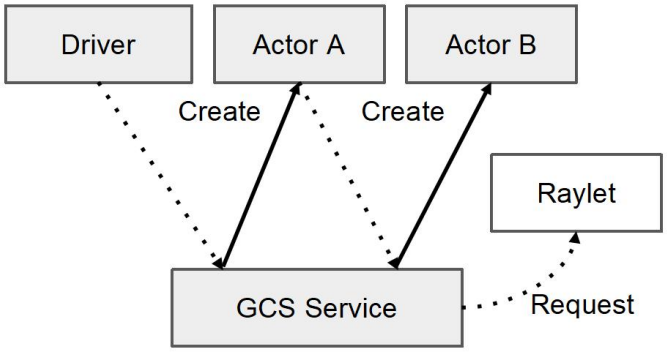
- src/ray/core_worker/store_provider/memory_store/memory_store.cc
- src/ray/core_worker/store_provider/plasma_store_provider.cc
- src/ray/core_worker/reference_count.cc
- src/ray/object_manager/object_manager.cc

Lifetime of an Actor

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

---

[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.



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 first synchronously registers the actor with the GCS. This ensures correctness in case the creating worker fails before the actor can be created. Once the GCS responds, the remainder of the actor creation process is asynchronous. The creating worker process queues locally a special task known as the actor creation task. This is similar to a normal non-actor task, except that its specified resources are acquired for the lifetime of the actor process. The creator asynchronously resolves the dependencies for the actor creation task, then sends it to the GCS service to be scheduled. 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. 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. 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.

An actor will be 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 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.

Ray also supports async actors that can concurrently run tasks using an asyncio event loop. 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 an asyncio event loop running in a background thread or thread pool instead of run directly on the main thread.

---

[3] Unless using async actors.

Code references:

- src/ray/core_worker/core_worker.cc
- 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

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. There is an ongoing effort to support high availability for the GCS, so that it may run on any and multiple nodes.

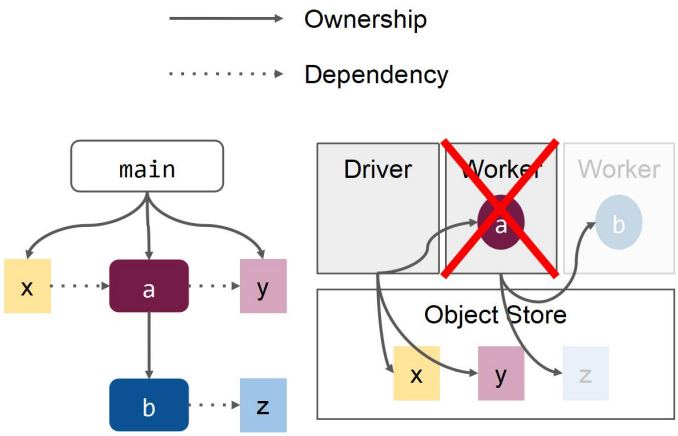
All nodes are assigned a unique identifier and communicate with each other through heartbeats. The GCS is responsible for deciding the membership of a cluster, i.e. which nodes are currently alive. The GCS tombstones any node ID that times out, meaning that a new raylet must be started on that node with 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 cancelled.
- 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:

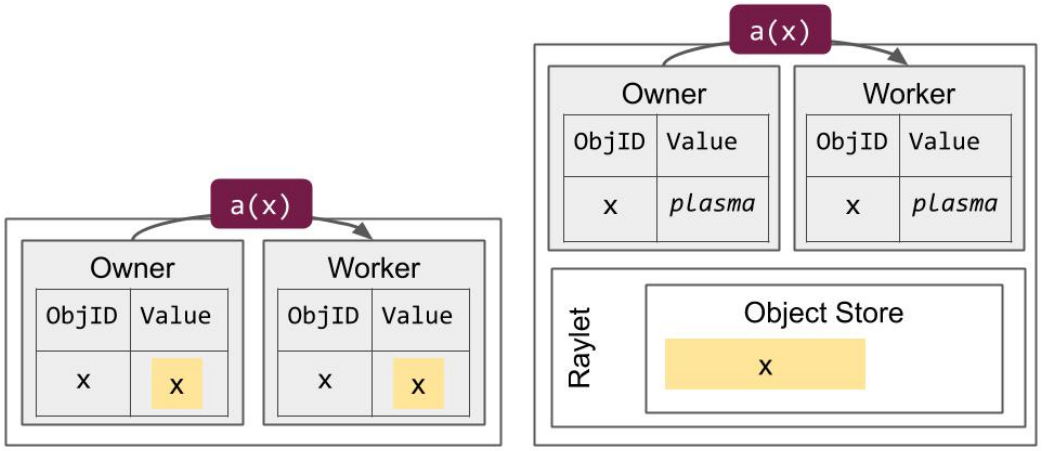
- Any other live process will receive an application-level exception if trying to get the value of such an object. For example, if the `z` ObjectRef had been passed back to the driver in the above scenario, the driver would receive an error on `ray.get(z)`.
- Failures can be isolated from one another 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.

As of v1.3, object spilling can also be used to allow objects to persist past the lifetime of their owner.

Finally, Ray provides some options to aid in transparent recovery, including automatic task retries, actor restart, and object reconstruction.

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 has been *promoted to shared memory*.

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 IPC 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:

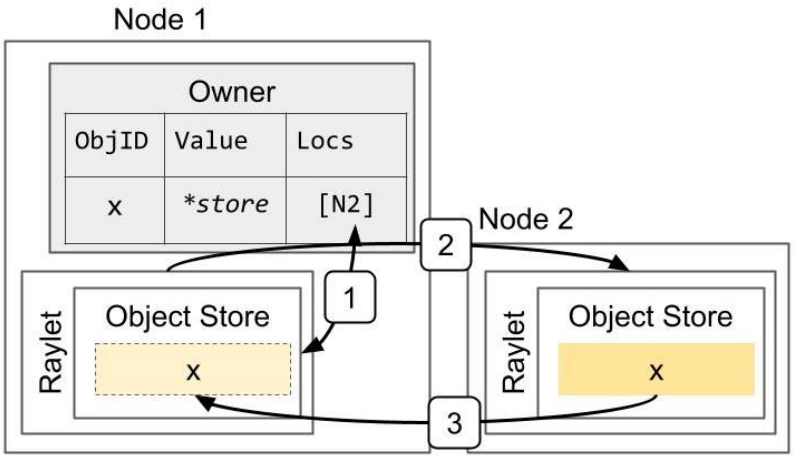
- src/ray/core_worker/store_provider/memory_store/memory_store.cc
- src/ray/core_worker/store_provider/plasma_store_provider.cc
- src/ray/common/buffer.h
- src/ray/protobuf/object_manager.proto

Object resolution

The value of an object can be resolved using an `ObjectRef`. The `ObjectRef` comprises two fields:

- A unique 20-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 description. Note that these objects are local to the owner process: if a borrower attempts to resolve the value, the object is *promoted* to shared memory, where it can be retrieved through the distributed object resolution protocol described next.



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

[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.

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.

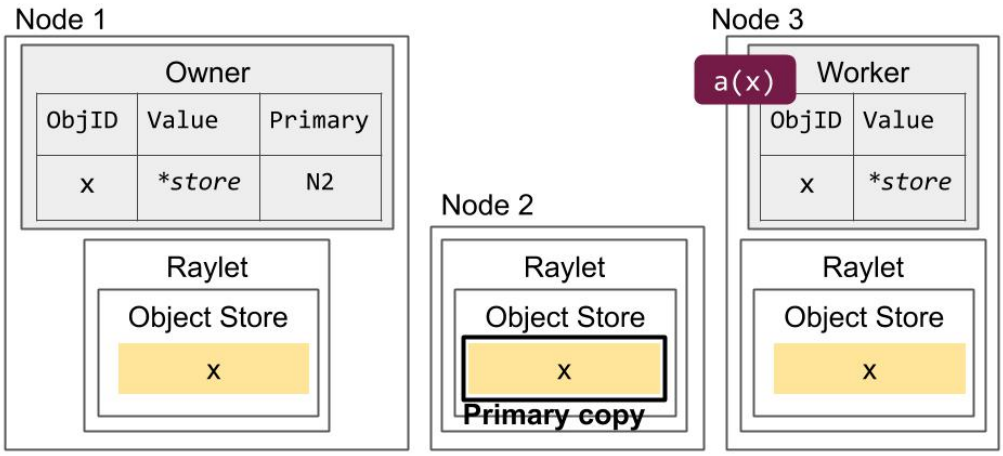
If the object is not available in the local shared memory store, the reference holder notifies its local raylet, which then attempts to fetch a copy from a remote raylet. The raylet looks up the locations from the object directory and requests a transfer from one of these raylets. The object directory 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. 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*.



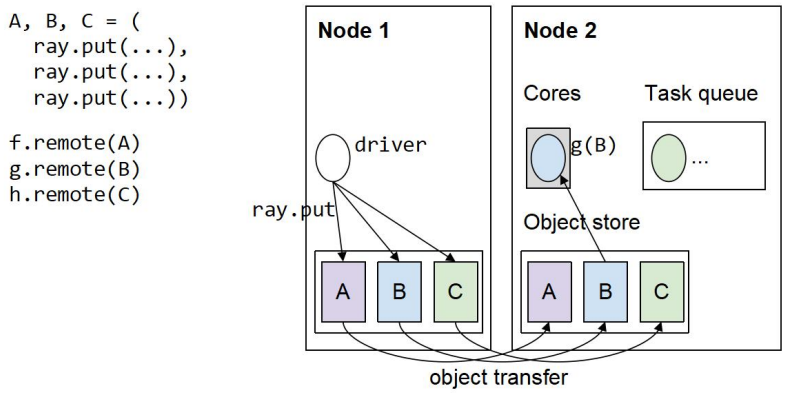
Primary copy versus evictable copies. The primary copy (Node 2) is ineligible for eviction. However, the copies on Nodes 1 (created through `ray.get`) and 3 (created through task submission) can be evicted under memory pressure.

The primary copy is unique in that it will not be evicted as long as the owner's ref count for the object is greater than 0. This is in contrast to other copies of the object, which may get evicted by LRU under local memory pressure.

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.



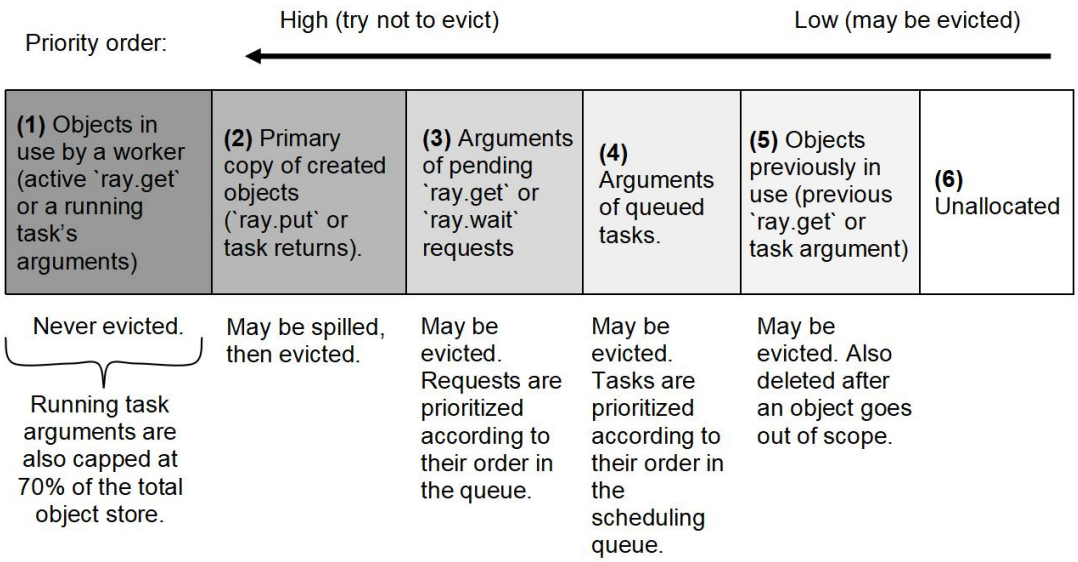
The types of objects that can be stored on a node. Objects are either created by a worker (such as A, B, and C on node 1), or a copy is transferred from a different node because it is needed by a local worker (such as A, B, and C on node 2).

Thus, an object may be stored on a node due to any of the following reasons:

1. It was requested by a worker process through `ray.get` or `ray.wait`. These can be freed once the worker finishes the `ray.get` request and the Python reference to the object's value goes out of scope.
2. It was returned by a 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 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 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.

Handling out-of-memory cases

The raylet is responsible for managing all of these objects while staying under that node's memory capacity. Below is a visualization of the different types of objects that may be stored on a node, with a rough priority.

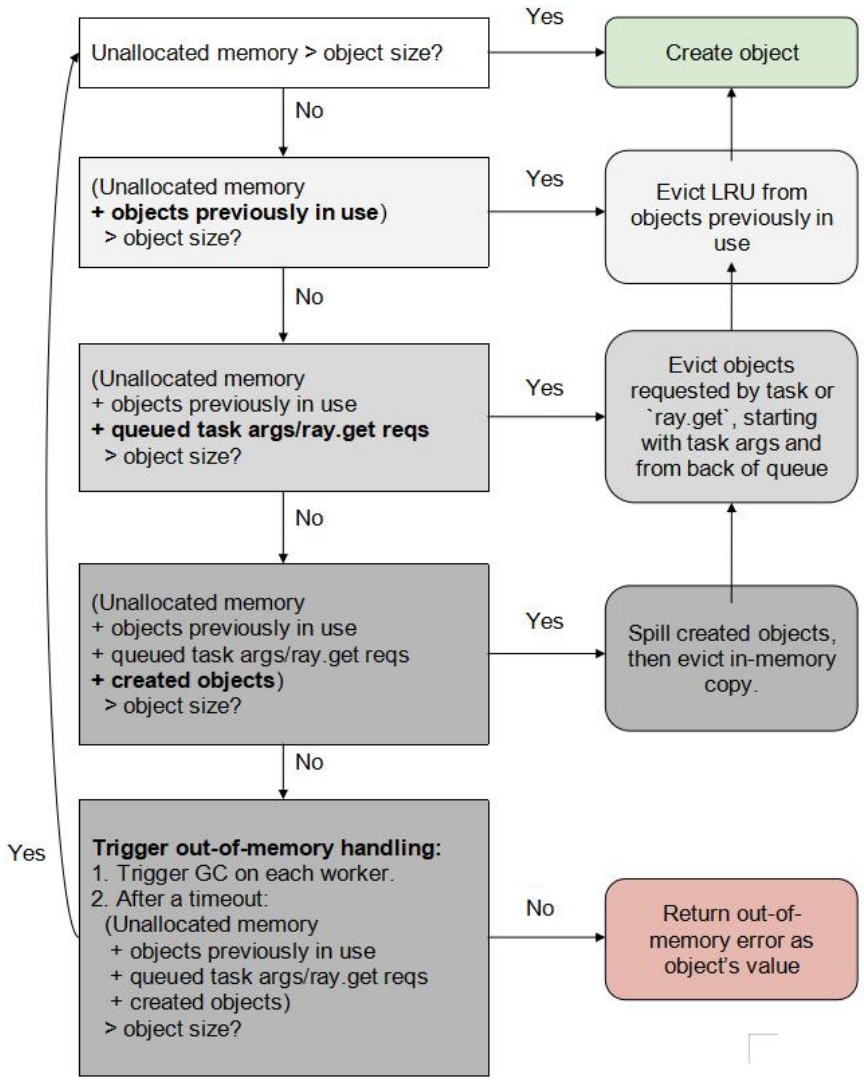


Object creation requests are queued by the raylet and served once enough memory is available in (6) to create the object. If more memory is needed, the raylet will choose objects to evict from (3)-(5) to make space. Even after all of these objects are evicted, the raylet may not have space for the new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

Ray v1.2+ features spilling to external storage to better support these cases. If more space is needed after eviction, the raylet also triggers spilling. 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.

Note that an ObjectStoreFullError can still be thrown 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 a worker's `ray.get` request. If there are excessive concurrent `ray.get` requests of large objects, the application may receive an ObjectStoreFullError. **The memory management system is under active development, so if you encounter problems with a memory-intensive application, please file an issue on Github.**



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 to free memory by evicting local objects according to the objects' priority, described above.

Object spilling and persistence

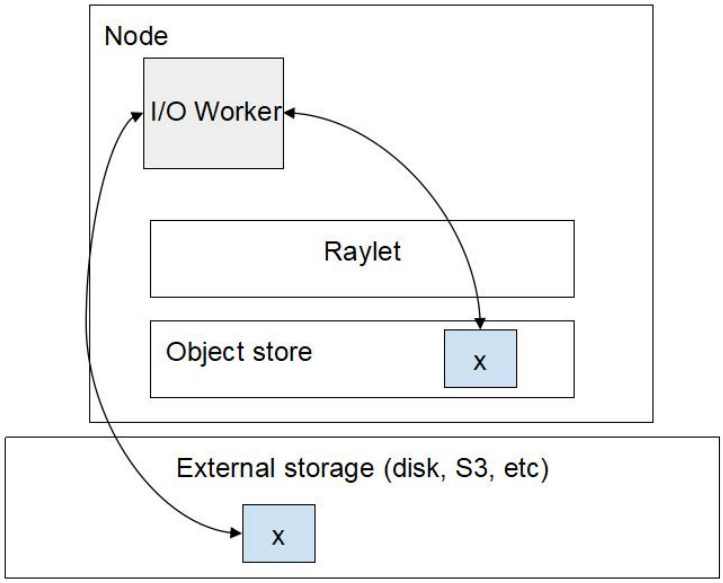
Ray 1.3.0+ has default support for spilling objects to external storage once the capacity of the object store is used up. This enables out-of-core data processing and memory-intensive distributed applications.

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

- Local storage. Local disk is selected by default so that Ray users can use the object spilling feature without any additional configuration.
- Distributed storage (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.

- Raylet: Keeps track of object metadata, such as the location in external storage, and coordinates IO workers and communication with other raylets.
- Object store
- IO workers: Python processes that are in charge of spilling and restoring 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.

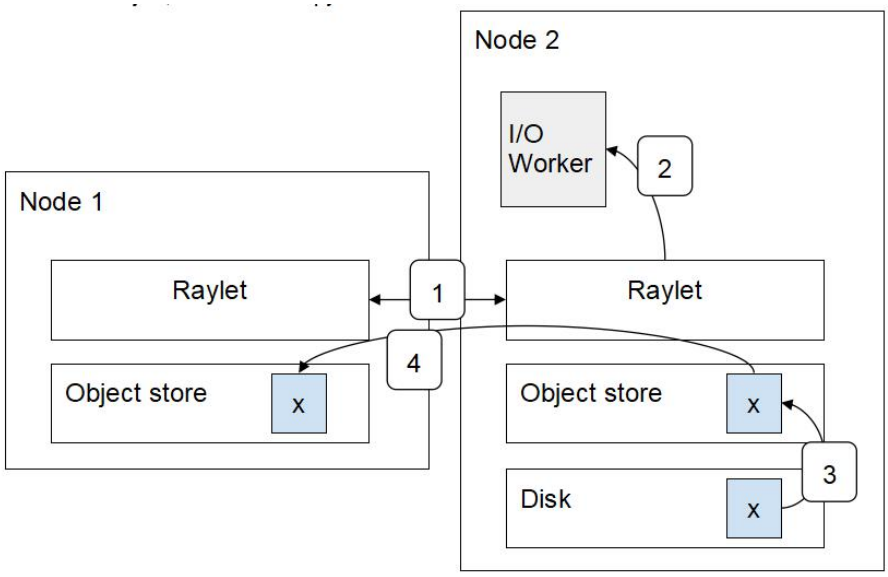
When Ray does not have enough memory capacity to create objects, it initiates object spilling. Note that Ray only spills the primary copy of an object: this is the initial copy of the object, created by executing a task or through `ray.put`. Primary copies are only evictable after object spilling, or 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 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 obtains the object with `ray.get` and copies the object 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.

There are two types of external storage: local storage and distributed storage. When local storage (e.g., local SSD) is used, there is one additional step to restore an object because the object may be stored on the disk of a remote node (the node where it was originally spilled). This is shown in the below diagram, where the raylet on node 1 requests the raylet on node 2 to restore the object, then send a copy.



Restoring an object from local storage. (1) Raylet that needs the object sends a restore request to the raylet that has the object in its local disk. (2) Remote raylet dispatches a restoration request. (3) Object is restored at the remote raylet. (4) Object is transferred to the requesting raylet.

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.
- 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.

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):
    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 `borrower.remote` and `b.borrower.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 description.
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 `ObjectRef`. Non-owning workers also maintain this set, in	The worker adds another worker's ID to this set when it discovers that the `ObjectRef` is being borrowed by that worker. For example, the caller would add an actor's worker ID when an actor task that saves the `ObjectRef` in the local state finishes. Removal if an owner: The owner sends a

	case the worker sends the `ObjectRef` to another borrower.	long-running 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. Removal if a borrower: The worker waits for RPC from the owner. Once the worker's 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	Only 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.

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.
```

References can also be created "out-of-band" by pickling an `ObjectRef` with `ray.cloudpickle`. In this case, 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 (e.g., passing the binary string that uniquely identifies an `ObjectRef`) are not guaranteed to work because they do not contain the owner's address and the reference is not tracked by the owner.

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.

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 Loss

Small objects: Small objects that are stored in the in-process object store fate-share with their owner. Since borrowed objects are promoted to shared memory, any borrowers will detect the failure through the distributed protocol described below.

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 none exist, the owner stores a system-level error that will be thrown during object resolution.

Ray also supports *object reconstruction*, or recovery of a lost object through re-execution of the task that created the object. When this feature is enabled, the owner caches the object *lineage*: the descriptions of the tasks needed to recreate an object in memory. Then, if all object copies are lost due to a failure, the owner resubmits the task that returned the object. Any objects that the task depends on are recursively reconstructed.

Object reconstruction is not supported for objects created with `ray.put`: the primary copy for these objects is always the owner's local shared memory store. Thus, the primary copy cannot be lost independently of the owner process.

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.

Resource Management and Scheduling

A resource in Ray is any "key" -> float quantity. For convenience, the Ray scheduler has native support for CPU, GPU, and memory resource types, meaning that Ray automatically detects physical availability of these resources on each node. However, the user may also define custom resource requirements using any valid string, e.g., specifying a resource requirement of {"something": 1}.

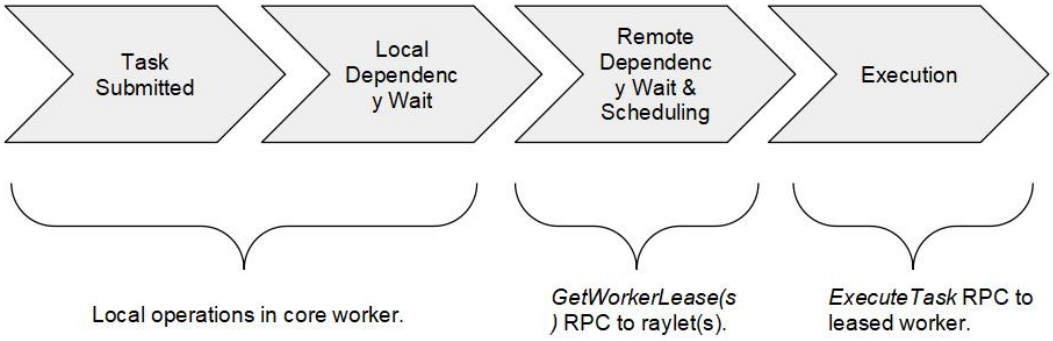
The purpose of the distributed scheduler is to match resource requests from the owners to resource availability in the cluster. Resource requests are hard scheduling constraints. For example, {"CPU": 1.0, "GPU": 1.0} represents a request for 1 CPU and 1 GPU. This task can only be scheduled on a node that has >= 1 CPU and >= 1 GPU. Each `@ray.remote` function requires one CPU for execution ({"CPU": 1}). An actor, i.e. a `@ray.remote` class, will request {"CPU": 0} for placement by default.

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 resource limits are not enforced by Ray (except for actor memory: if specified, an actor's memory limit is checked at the end of each task). 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.

Task scheduling (owner-raylet protocol)



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 when 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.

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_id)
- Non-inlined object: f.remote(large_or_pending_obj_id)

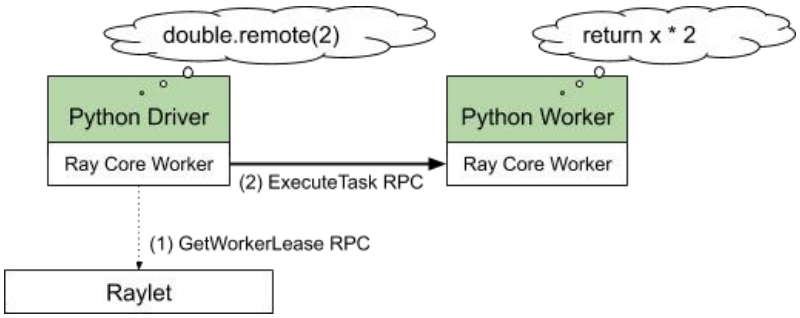
Plain values don't require dependency resolution.

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

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 owner 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.

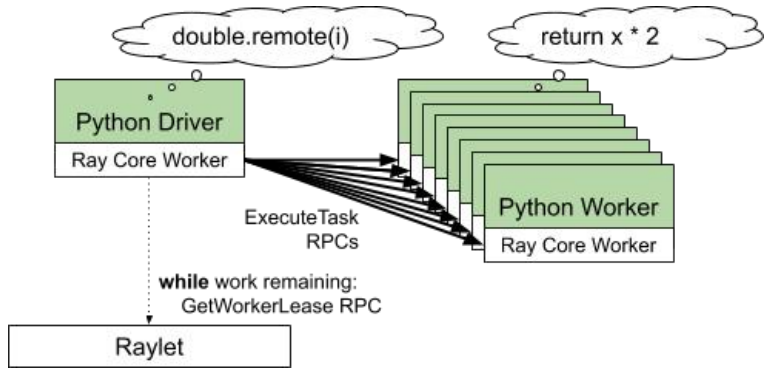
Resource fulfillment

An owner schedules a task by first sending a resource request to its local raylet. The raylet queues the request and if it chooses to grant the resources, responds to the owner with the address of a local worker that is now *leased* to the owner. The lease remains active as long as the owner 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, an owner returns the worker if no work remains or if enough time has passed (e.g., a few hundred milliseconds).



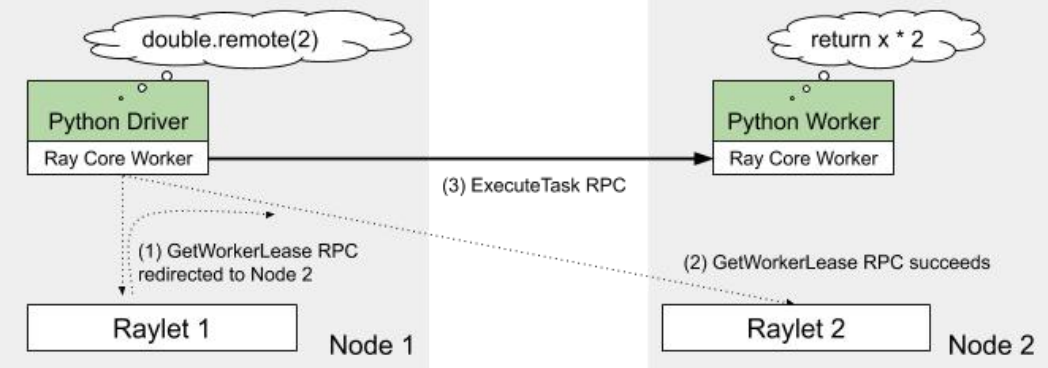
Resource fulfillment and execution of the `double(2)` task in a Ray cluster.

The owner 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.



Owner 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 raylet chooses not to grant the resources locally, it may also respond to the owner with the address of a remote raylet at which the owner should retry the resource request. This is known as *spillback scheduling*. Spillback scheduling is iterative: instead of forwarding the resource request directly, each raylet responds to the owner with the address of the next raylet to try. This ensures that the owner's metadata about the location of a task is always consistent.



During spillback scheduling, the raylet redirects the owner's request to a remote raylet that has 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

Distributed scheduler (raylet-raylet protocol)

Resource accounting

Each raylet tracks the resources local to its node. When a resource request is granted, the raylet decreases the local resource availability 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. Every heartbeat interval (100ms by default), each raylet sends its current resource availability to the GCS service. The GCS aggregates these heartbeats and rebroadcasts them to each raylet.

Scheduling policy

A raylet always attempts to grant a resource request using local resources first. When there are no local resources available, there are three other possibilities:

1. Another node has enough resources, according to the possibly stale information published by the GCS. The raylet will spillback the request to the other raylet.
2. No node currently has enough resources. The task is queued locally until resources on the local or remote node become available again.

3. No node in the cluster has the requested resources (e.g., a {“GPU”: 1} request in a CPU-only cluster). The task is considered *infeasible*. The raylet emits a warning message to the corresponding driver. The raylet queues the task until resources become available, e.g., a node with 1 GPU is added to the cluster.

In the future, the raylet may also make locality-based scheduling decisions (i.e.,spillback a task to a node that already has a task argument local). Object locality is not currently implemented.

Code references:

- src/ray/raylet/node_manager.cc
- src/ray/protobuf/node_manager.proto

Autoscaler

The Ray Autoscaler (also known as the Cluster Launcher), is responsible for bringing up an initial set of cluster nodes and adding additional nodes as required based on resource demands.

In Ray versions **<= 1.0.1**, the Autoscaler implements the following control loop:

1. It calculates the estimated utilization of the cluster based on the *most-currently-assigned resource*. For example, suppose a cluster has 100/200 CPUs assigned, but 15/25 GPUs assigned, then the utilization will be considered to be `max(100/200, 15/25) = 60%`.
2. If the estimated utilization is greater than the target (80% by default), then the autoscaler will attempt to add nodes to the cluster.
3. If a node is idle for a timeout (5 minutes by default), it is removed from the cluster.

In Ray versions **1.1+**, the control loop is implemented as follows:

1. It calculates the number of nodes required to satisfy all currently pending tasks, actor, and placement group requests.
2. Launch new nodes.
    a. If the number of nodes required total divided by the number of current nodes exceeds `1 + upscaling_speed`, then the number of nodes launched will be limited by that threshold.
    b. When nodes are launched via request_resources(), the upscaling_speed limit is bypassed.
3. If a node is idle for a timeout (5 minutes by default), it is removed from the cluster.

The advantage of the new algorithm in 1.1 is that it upscales to exactly the number of nodes needed to meet resource demands. Since the previous algorithm only used aggregate utilization data, it could not know exactly how many nodes will be needed.

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

Custom Resources

Beyond native system resources such as CPU, GPU, and memory, Ray supports the definition and usage of custom resources. These are generally added to a node on startup, for example, a node may advertise that it has a particular hardware feature and dataset with custom resources {"HasHardwareFeature": 1, "HasDatasetA": 1}. Tasks and actors can require some quantity (e.g., 0.01) of this resource for scheduling, which effectively constrains them to running on that particular node. Custom resources can also be added to a node dynamically by tasks.

Placement Groups

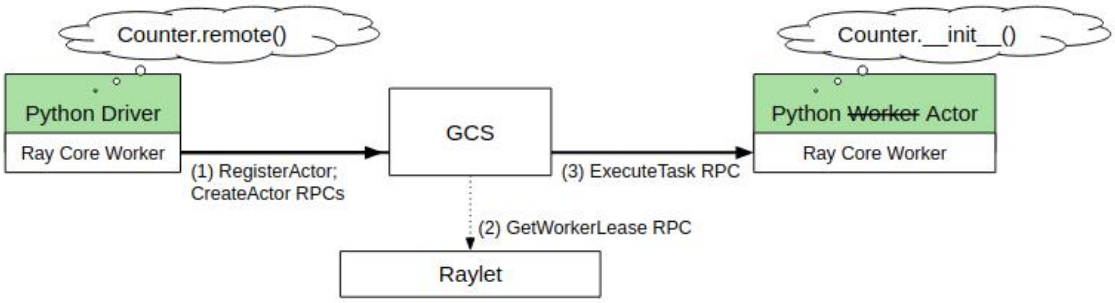
In 1.0, Ray supports Placement Groups. Placement groups allow users to atomically reserve groups of resources across multiple nodes (i.e., gang scheduling). They can request the resource bundles that make up the group to be packed as close as possible for locality (PACK), or spread apart (SPREAD). Groups 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.

Multi-tenancy

In 1.0, Ray supports Multi-tenancy. The basic level of multi-tenancy that will be supported is setting different environment variables for workers of different jobs. This allows multiple soft-isolated environments (e.g., different PYTHONPATH, Java CLASSPATH) to exist within one Ray cluster. To ensure isolation, worker processes are not reused across different jobs when multi-tenancy is enabled.

Actor management

Actor creation



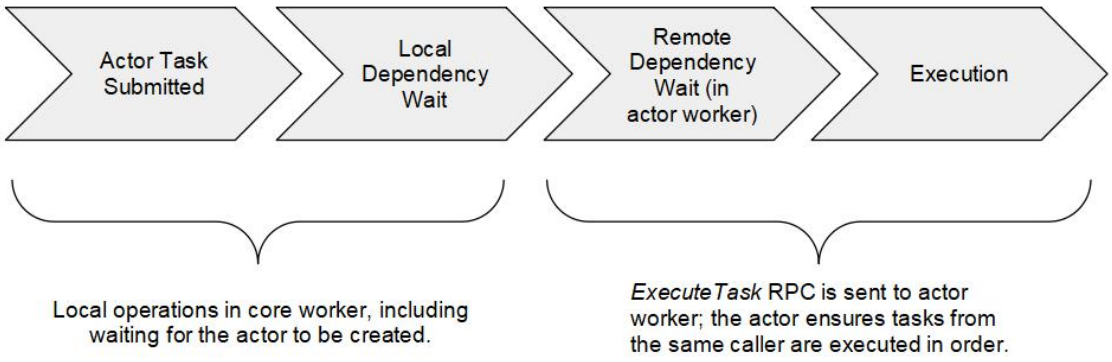
Actor creation tasks are scheduled through the centralized GCS service.

When an actor is created in Python, the creating worker first synchronously registers the actor with the GCS. This ensures that in the case that the creating worker dies before the actor can be created, any workers with a reference to the actor will be able to discover the failure.

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 owner. Because the GCS service persists all state to the backing store, once the task specification has successfully been sent to the GCS service, the actor will eventually be created.

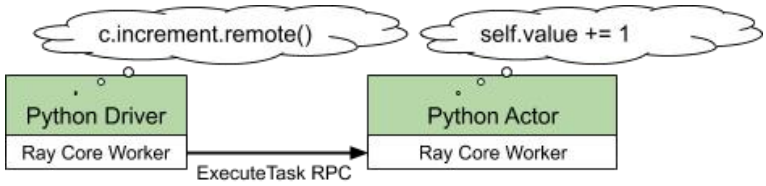
The original creator of the handle can begin to submit tasks on the actor handle or even pass it to other tasks/actors before the GCS has scheduled the actor. Once the actor has been created, the GCS notifies any worker that has a handle to the actor via pub-sub. Each handle caches the newly created actor's runtime metadata (e.g., RPC address and the node it's on). Any pending tasks that had been submitted on the actor handle can then be sent to the actor for execution.

Actor task execution



The scheduling workflow of a Ray actor task.

Each 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.

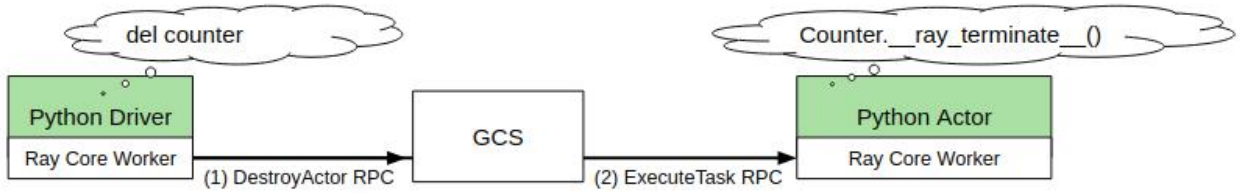
Each submitted task is assigned a sequence number on the caller side, which is used by the receiving actor to ensure that tasks from each caller are executed in the order that they were submitted, even if the messages were reordered in flight. There are no guarantees on task execution ordering across callers. For example, execution ordering between callers may vary depending on message delays and the order in which the tasks' dependencies become available.

Actor death

Actors may be detached or non-detached. Non-detached actors are the default and are recommended for storing transient state whose physical resources should be automatically collected when the actor's parent or job exits.

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 submits a special `__ray_terminate__` task to the actor that will cause the actor to gracefully 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 tasks are also scheduled 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 to automatically restart actors, up to a specified number of times. If this option is enabled, 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 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 Store

The global control store (GCS) holds critical but less frequently accessed cluster metadata such as the addresses of connected clients and nodes. In earlier versions of Ray (<0.8), the GCS also held object lineage and metadata for small objects, meaning that the GCS was on the critical path for most system operations, e.g., task scheduling. In newer versions of Ray (0.8+), object lineage and metadata has moved largely into the worker processes, allowing the GCS to remain off the critical path during most system operations. This has led to overall improved performance and reduced GCS storage requirements.

Storage

The GCS is currently implemented in Redis, and we rely on Redis for pub-sub. However, an effort is underway to remove the redis dependency and enable pluggable persistent storage (e.g., MySQL).

Actor Table

This holds the list of actors and their state. This table is used to recreate actors on failure, and to manage actor lifetime.

Heartbeat Table

This holds the list of clients, workers, and nodes connected to Ray.

Each raylet periodically sends a heartbeat to the GCS to indicate that the node is alive and to report a summary of its scheduler thread's current resource usage and load. The GCS periodically aggregates all heartbeats from the raylets, to reduce network bandwidth usage, and broadcasts the aggregate back out to all nodes. This is used to determine cluster membership and for distributed scheduling. The broadcasted information is also used for autoscaling.

If the GCS does not hear from a raylet for a configurable number of heartbeat intervals, the GCS marks the raylet as dead and broadcasts this message to all nodes. This acts as a tombstone: a raylet will exit if it hears that it has been marked as dead and its physical resources can only be reused by starting a new raylet, which is assigned a different unique ID.

Job Table

This holds the list of jobs running in the cluster. When a job is terminated, Ray will cancel running tasks and actors created by the job to avoid resource leaks.

Object Table

In previous versions of Ray, the GCS stored the node locations for large shared-memory objects. As of v1.3, the object table has been moved from the GCS into the ownership table, to improve scalability and distributed object transfer performance.

Profile Table

Profiling events are stored in this table, and may be LRU evicted.

Persistence

The GCS currently does not provide persistence, though an effort is underway to enable pluggable storage via generic SQL database. A max size is set for Redis, which results in LRU

eviction of non-critical data. However, with recent versions of Ray (0.8+), exceeding this max size is very rare, since most object and task metadata is no longer stored in the GCS.

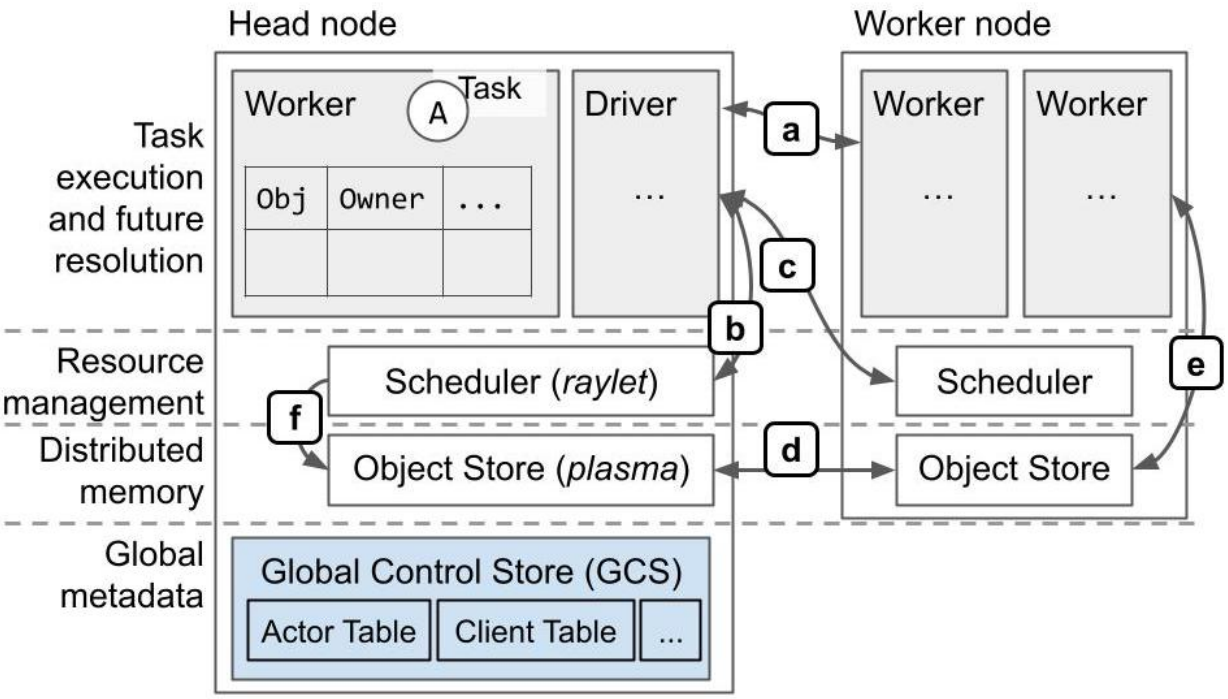
Code references:

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

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

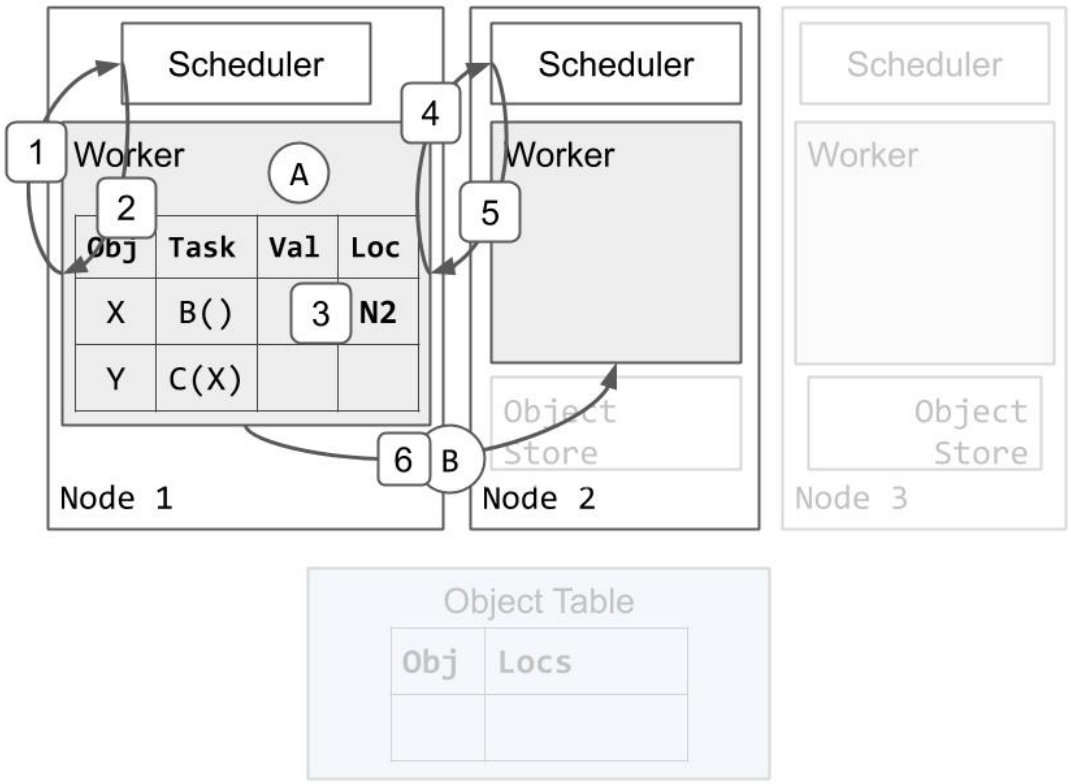
We'll walk through the physical execution of a Ray program that looks like this:

```
@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:

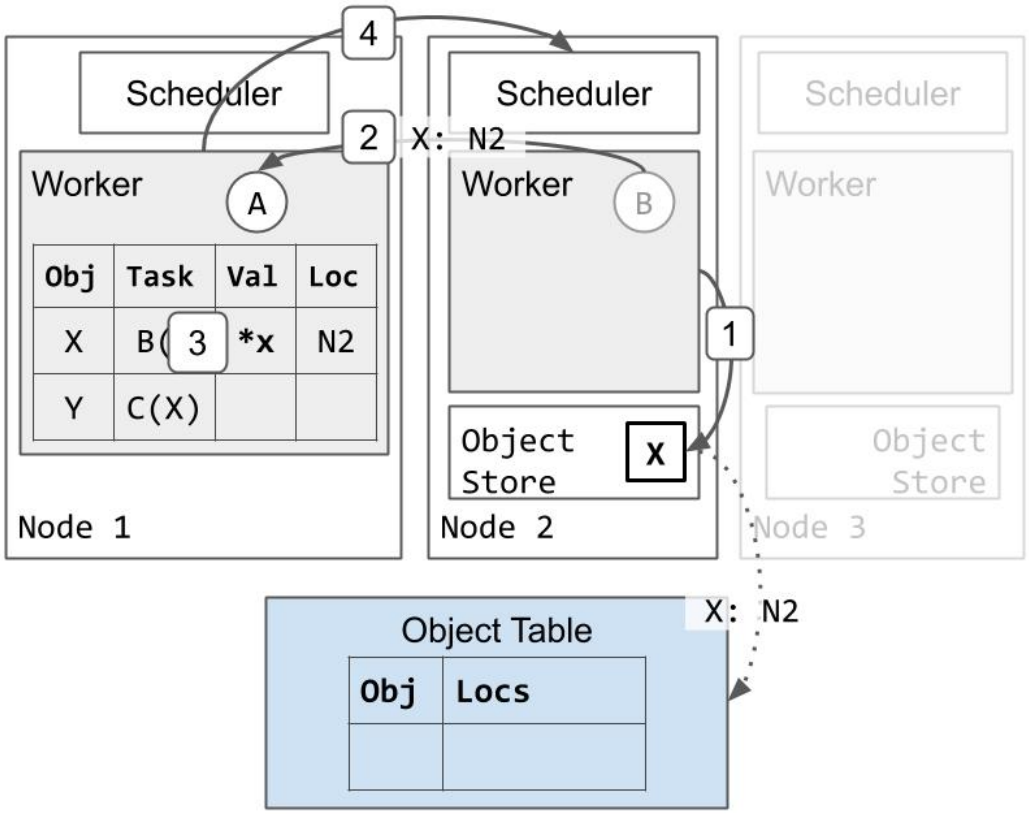


1. Worker 1 asks its local scheduler for resources to execute B.
2. Scheduler 1 responds, telling worker 1 to retry the scheduling request at node 2.
3. Worker 1 updates its local ownership table to indicate that task B is pending on node 2.
4. Worker 1 asks the scheduler on node 2 for resources to execute 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:

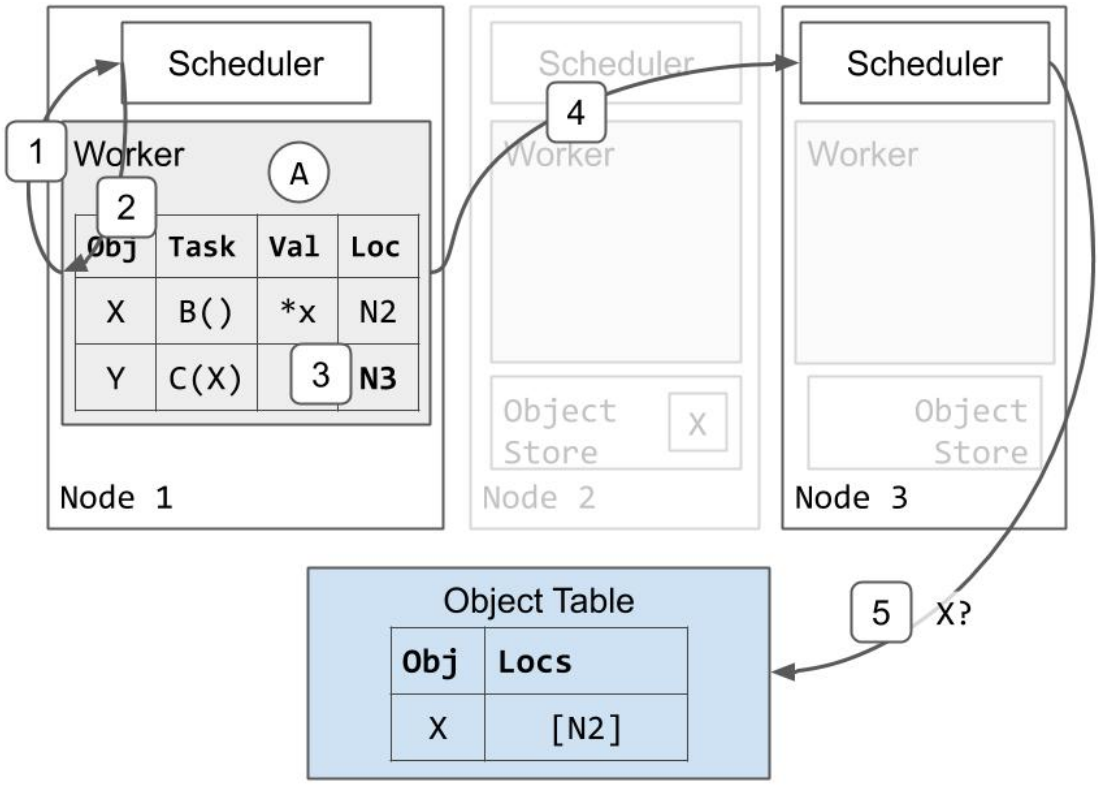


1. Worker 2 finishes executing B and stores the return value X in its local object store.
    a. Node 2 asynchronously updates the object table to indicate that X is now on node 2 (dotted arrow).
    b. Since this is the first copy of X to be created, node 2 also pins its copy of X until worker 1 notifies 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

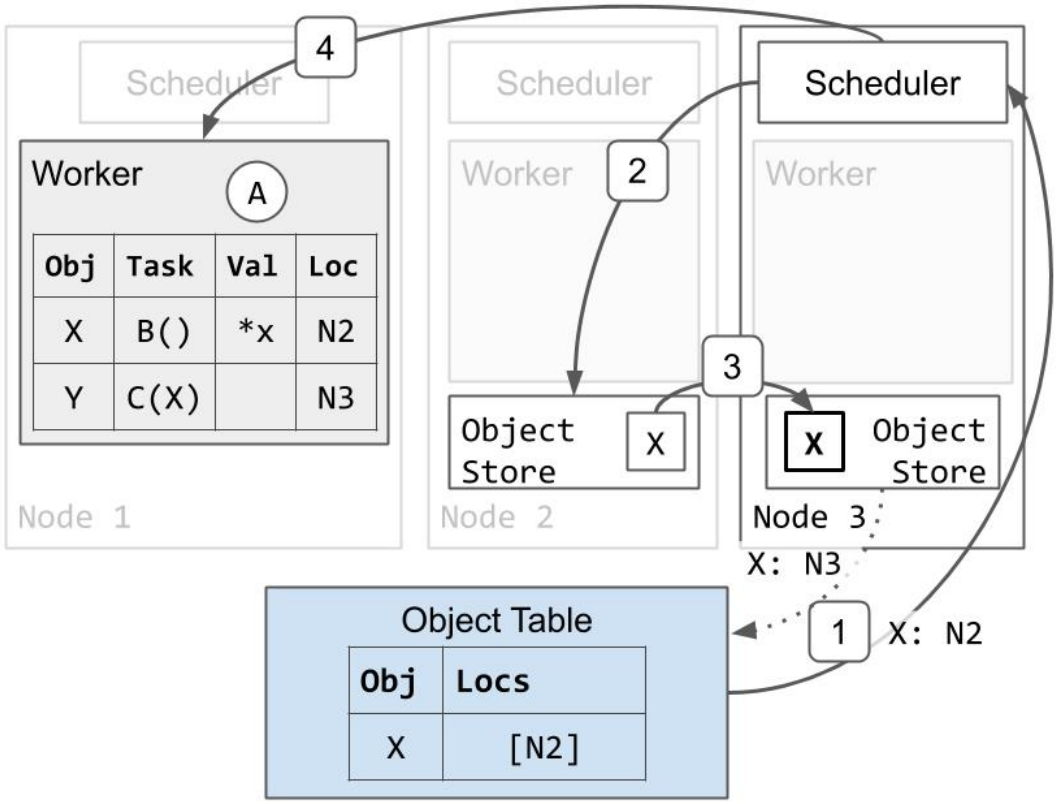
Now that B has finished, task C can start execution. Worker 1 schedules C next, using a similar protocol as for task

B:



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.

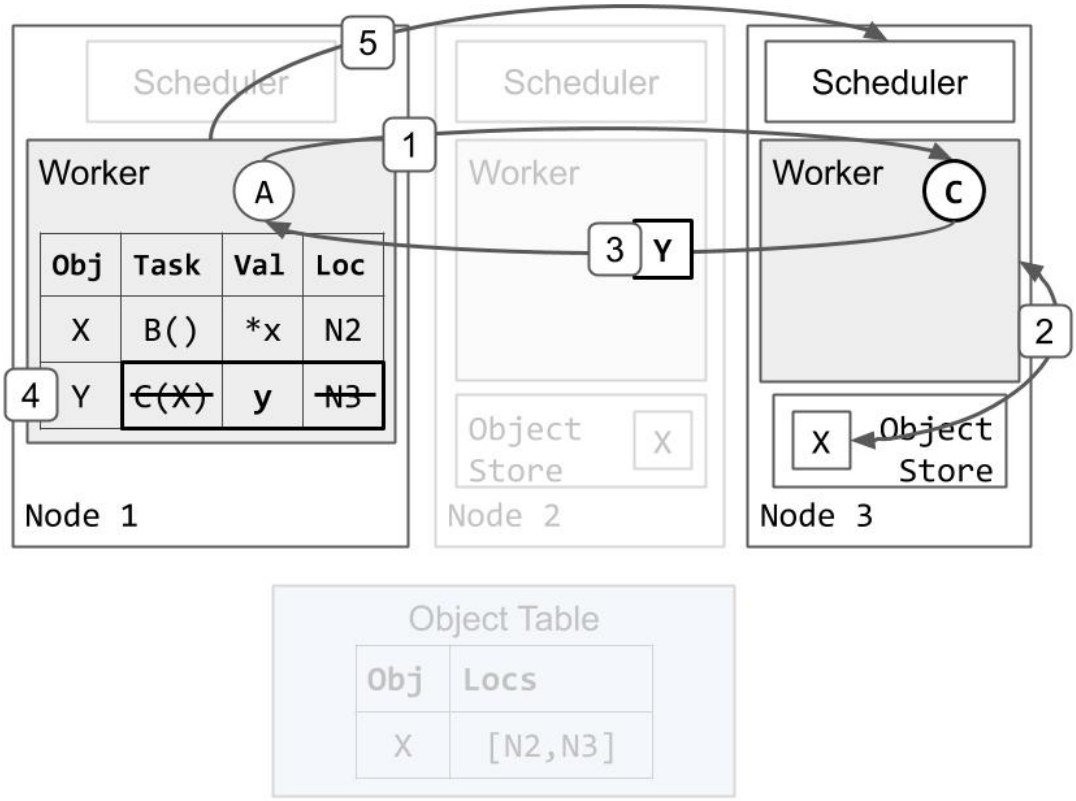
Task C requires a local copy of X to begin execution, so node 3 fetches a copy of X:



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

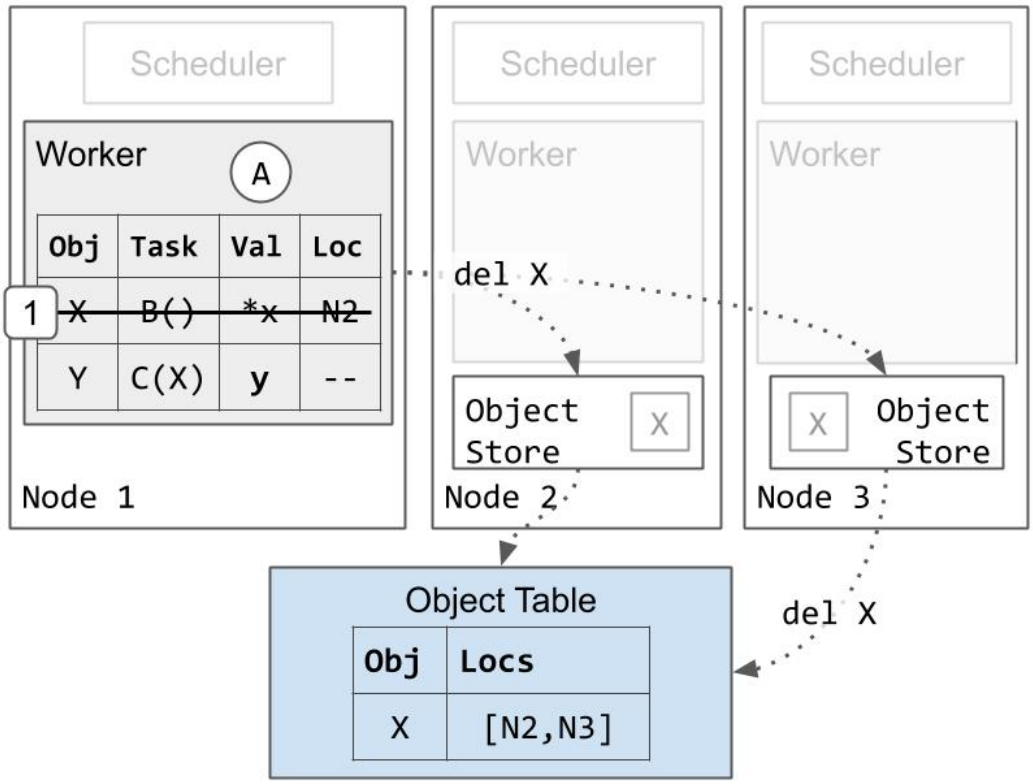
Task C executes and returns an object small enough to be stored in the in-process memory store:



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 description 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:



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.