This doc covers the implementation plan and any unknowns for completing the distributed ref counting effort. Specifically, it will cover ref counting for object IDs that have been serialized. Ref counting for object IDs that were not serialized is straightforward and has already been implemented by increasing the ref count for each argument of a submitted task.

This doc assumes an ownership-based architecture where the owner of the object notifies the raylet holding the primary object copy when it is OK to delete the copy. It assumes that the owner process does not change and that the primary object copy's location does not change.

Requirements

1. If a remote task or actor has a reference to an ObjectID, then the owner's ref count for that object is > 0.
    a. This is only relevant if the owner serialized the ObjectID, since tasks/actors do not receive a reference to ObjectIDs that were passed as a task argument.
2. Correct in the presence of worker or node failures.
    a. Example: If the only references to an ObjectID are held by processes that have failed, then the owner's ref count should eventually go to 0.
3. Reduce complexity.
    a. Many distributed ref counting algorithms require certain operations to be synchronous to prevent race conditions. We should pick designs that avoid these cases as much as possible.
4. Nice-to-have: Reduce messaging load on the owner of an object. Ideally, we should piggy-back as many ref count operations onto existing messages as possible.

Algorithm

Terminology

Owner: The process that creates the initial ObjectID (note, not necessarily the value).
Borrower: A process that has a reference to an ObjectID that it does not own. Borrowers are created when either the owner or another borrower serializes the ObjectID into an object.
Parent: The process that gave a borrower its reference to an ObjectID. This is either the owner of the ObjectID or another borrower.
Task reference: A process (actor or regular worker) that has a reference to an object's value but not the ID. Task references are created when the owner or borrower submits a task that depends on the object.

The owner and all borrowers keep 4 different ref counts for an ObjectID:
Python ref count: Number of copies of the ObjectID() in Python
Task ref count: Number of submitted pending tasks that depend on the ObjectID
Serialized ref count: Number of ObjectIDs that contain the serialized ObjectID and that are still in scope. Can also keep this as a list of ObjectIDs for debugging purposes.
Borrowers list: List of borrowers

For simplicity, we will say:
Local ref count: All types of ref counts defined above except the list of borrowers.

Description

This design allows borrowers to accumulate a ref count for their local copy of an ObjectID. Each borrower merges its ref count into its parent. The merge algorithm has 2 cases:

1. If the borrower finished its task but is still using its reference (its local ref count is > 0), then its parent considers it a borrower.
    a. Example: An actor keeps its reference (Python ref count > 0).
    b. Example: A task that no longer has the reference but has pending tasks that depend on the object (task ref count > 0).
2. If the borrower has accumulated its own borrowers list, then these are popped from the borrower and appended to its parent's list of borrowers.

Eventually, all borrowers merge their ref count into the owner's. Borrowers merge their ref count into the owner's in 2 cases:

1. The borrower has finished a task and is responding to the task's caller (its parent). Note that the task's caller may be the owner, or it could be yet another borrower of the same ID. In the latter case, the task's caller recursively merges the ref count until all refs are merged into the owner.
2. The borrower's local ref count for an ID is 0 and it is responding to the owner's NotifyRefRemoved(id) call.

The owner monitors all borrowers in its local borrowers list via a grpc call (NotifyRefRemoved). A borrower will reply to the grpc call once its local ref count goes to 0. Or, if the grpc call fails, then the borrower is assumed to have failed. In both cases, the owner will remove the borrower from its list.

Corner cases

One case that this algorithm may not correctly handle in the case of failures is if a borrower is created from another borrower. This is code that looks like the following:

```
def owner():
  x_id = ray.put(...)
  borrower1.remote([x_id])

def borrower1(x_ids):
  # This borrower creates another borrower.
  borrower2.remote(x_ids)

def borrower2(x_ids):
  # Use x_id.
  # Until borrower1 merges its count into owner, the owner does not
know about us.
```

In this example, if borrower1 fails before it merges its borrowers list into the owner's list, then the owner will not know that borrower2 still has a reference. This is okay if borrower2 is a task, since it will fate-share with borrower1, but if it is an actor, it is possible that it can survive after borrower1 has finished.

One way to fix this is to fate-share borrower2's reference with borrower1. So, borrower2 would have to remember that its x_id reference came from borrower1, not the owner. If it receives the NotifyRefRemoved(id) message from the owner, then borrower2 can garbage-collect this information because the owner now knows that borrower2 exists. However, if borrower1 fails before NotifyRefRemoved(id) is received, then borrower2 must invalidate its reference to x_id (and recursively notify its children to do the same).

A simpler approach that we can do for now is to have borrower2 contact the owner periodically when trying to resolve x_id's value. If the owner no longer has a ref count for x_id, then it must be because borrower1 failed to merge its ref count into the owner. So, borrower2 can throw an exception for x_id instead of waiting for the value to appear. This approach does not cover all failure cases (i.e., it is possible for the owner to have a ref count of 0 even though there is still a borrower), but it is much simpler and should cover most cases. One alternative that is clearly bad is to never contact the owner, in which case borrower2 could hang.

Invariants

1. Owner_id and owner_address are non-nil for all object IDs that a process has a reference to except for objects created out-of-band (e.g., ray.ObjectID("custom ID string"))
2. If the owner's `python_ref_count + task_ref_count + serialized_ref_count + len(borrowers) > 0`, then the object is pending creation or available.
3. If a borrower exists for an object ID, then the object's owner has a ref count greater than 0.
    a. Because: By following the "borrowers" pointers that are local to each process, there is a path between the owner of an id and each borrower process (except in the failure case mentioned above).
    b. Except in the failure case mentioned above. This is why a borrower needs to check that the owner's ref count is > 0 when trying to fetch the object.
4. The owner's list of borrowers will eventually become empty.
    a. Because: the owner will send NotifyRefRemoved for each borrower in its list of borrowers. This RPC is guaranteed to eventually return with the reply or an error.
5. If a borrower of an ObjectID receives NotifyRefRemoved(id), then this must be *after* it has deserialized the reference to the ID.

Data structures

Local object table:

```
{
  ObjectID: Reference{
    size_t python_ref_count;
    size_t task_ref_count;
    TaskID owner_id;
    Address owner_address;
    std::function<void(const ObjectID &)> on_delete;

    /* New fields below to handle serialized ObjectIDs. */
    // ObjectIDs that contain this ObjectID.
    // Needed to keep the object in scope when the only copy of the
    // ObjectID remaining is a serialized copy.
    // Maintained by the owner and borrowers (in case they
deserialize
    // an ID, then serialize it again).
    // Invariant: This field contains only object IDs that we own.
    set<ObjectID> contained_in;
    // ObjectIDs that are serialized in this object ID's value.
    // Reverse mapping of contained_in.
    // Needed to decrement the ref count for ObjectIDs contained in
    // this object once this ref count goes to 0.
    // Maintained by the owner and borrowers that deserialize an
ObjectID that contains further ObjectIDs.
    // Note that this vector can contain ObjectIDs that we do not
own.
    const vector<ObjectID> contains;
    // Workers that have a reference to the ObjectID.
    // Owner has an outstanding grpc to each of these workers.
    // The worker keeps a local ref count for the object and replies
once its ref count for the ObjectID goes to 0.
    // If the message fails, then the worker is assumed to have
failed.
    // Maintained by the owner only.
    set<WorkerAddress> borrowers;

  }, ...
}
```

Protocol pseudocode

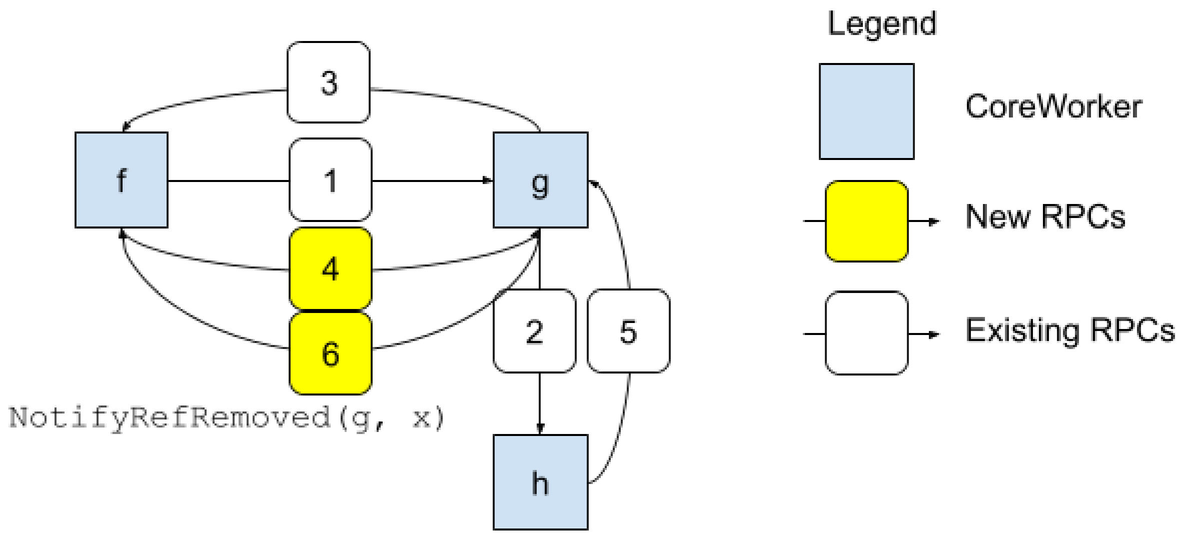
Operations:

1. Wrap an existing ObjectID into another ObjectID.
2. Unwrap an ObjectID from another ObjectID.
3. Submit a task that depends on an ObjectID.
    a. Send and receive.
4. Finish a task
    a. Send and receive.
    b. Return an ObjectID or return a normal value.
5. Local ObjectID goes out of scope.
6. NotifyRefRemoved on an ObjectID borrower.
    a. Borrower receives the NotifyRefRemoved message and owner receives the borrower's reply.
7. Worker fails.

Protocol examples

Example 1: Forwarding an ObjectID that we own.

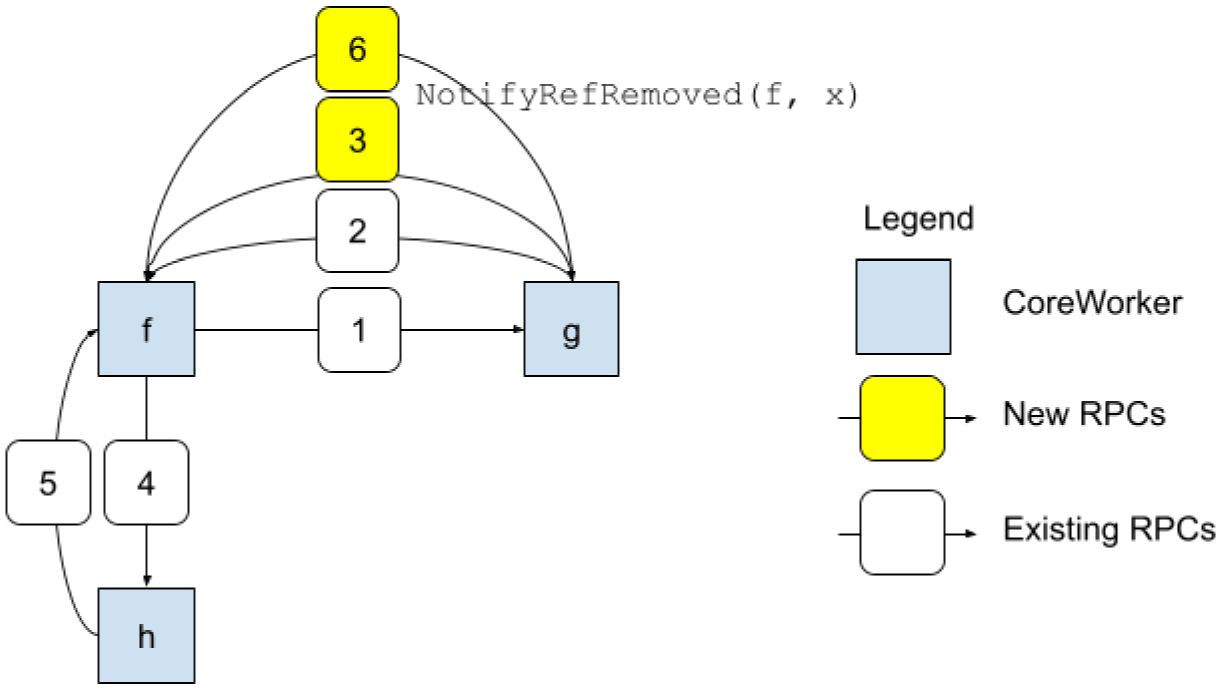
```
def f():
  x_id = ray.put(...)
  y_id = ray.put([x_id])
  g.remote(y_id)

def g(x_id_list):
  x_id = x_id_list[0]
  h.remote(x_id)

def h(x):
  ...
```



1. F submits g(y).
    a. F: x.contained_in.append(y)
    b. F: y.contains.append(x)
    c. F: y.task_refs.append(g)
2. G submits h(x).
    a. G: x.task_refs.append(h)
3. G returns to F. Note that H may still be executing, so G's task ref count for x is > 0.
    a. G tells F that its ref count for x is > 0.
    b. F: x.borrowers.append(g)
    c. F: y.task_refs.remove(g)
    d. F: delete y => x.contained_in.remove(y)

4. F asks G to notify F once G is done using x.
    a. G: Add x.on_delete callback to notify F once x's ref count goes to 0.
5. H returns to G.
    a. G: x.task_refs.remove(h)
    b. G: delete x, call on_delete
6. G replies to F.
    a. F: x.borrowers.remove(g).
    b. F: delete x.

Example 2: Returning an ObjectID.



```
def f():
  outer_id = g.remote()
  h.remote(outer_id)

def g():
  x_id = ray.put(...)
  return [x_id]

def h(x_id_list):
  x_id = x_id_list[0]
```

1. F submits g().
2. G returns reply to F.

    a. G: x.borrowers.append(f)

3. G asks F to notify G once F is done using x.
    a. F: If outer_id in references and x not in references:
        i. Add x = Reference(contained_in=[outer])
        ii. outer.contains = [x_id]
    b. F: Add x.on_delete callback to notify G once x's local ref count goes to 0.
4. F submits h(outer).
    a. outer.task_refs.append(h)
5. H returns to F.
    a. F: outer.task_refs.remove(h)
    b. F: delete outer, which deletes x. Call x.on_delete.
6. F replies to G.
    a. G: x.borrowers.remove(f)
    b. G: delete x.