orphan:

Remote Reference Protocol

This note describes the design details of Remote Reference protocol and walks through message flows in different scenarios. Make sure you're familiar with the :ref:`distributed-rpc-framework` before proceeding.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 8**); ***backlink***
>
> Unknown interpreted text role "ref".

Background

RRef stands for Remote REFerence. It is a reference of an object which is located on the local or remote worker, and transparently handles reference counting under the hood. Conceptually, it can be considered as a distributed shared pointer. Applications can create an RRef by calling :meth:`~torch.distributed.rpc.remote`. Each RRef is owned by the callee worker of the :meth:`~torch.distributed.rpc.remote` call (i.e., owner) and can be used by multiple users. The owner stores the real data and keeps track of the global reference count. Every RRef can be uniquely identified by a global `RRefId`, which is assigned at the time of creation on the caller of the :meth:`~torch.distributed.rpc.remote` call.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 15**); ***backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 15**); ***backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 15**); ***backlink***
>
> Unknown interpreted text role "meth".

On the owner worker, there is only one `OwnerRRef` instance, which contains the real data, while on user workers, there can be as many `UserRRefs` as necessary, and `UserRRef` does not hold the data. All usage on the owner will retrieve the unique `OwnerRRef` instance using the globally unique `RRefId`. A `UserRRef` will be created when it is used as an argument or return value in :meth:`~torch.distributed.rpc.rpc_sync`, :meth:`~torch.distributed.rpc.rpc_async` or :meth:`~torch.distributed.rpc.remote` invocation, and the owner will be notified according to update the reference count. An `OwnerRRef` and its data will be deleted when there is no `UserRRef` instances globally and there are no reference to the `OwnerRRef` on the owner as well.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 26**); ***backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 26**); ***backlink***
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, **line 26**); ***backlink***
>
> Unknown interpreted text role "meth".

Assumptions

RRef protocol is designed with the following assumptions.

- **Transient Network Failures**: The RRef design handles transient network failures by retrying messages. It cannot handle node crashes or permanent network partitions. When those incidents occur, the application should take down all workers, revert to the previous checkpoint, and resume training.
- **Non-idempotent UDFs**: We assume the user functions (UDF) provided to :meth:`~torch.distributed.rpc.rpc_sync`,

:meth:`~torch.distributed.rpc.rpc_async` or :meth:`~torch.distributed.rpc.remote` are not idempotent and therefore cannot be retried. However, internal RRef control messages are idempotent and retried upon message failure.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst, line 49); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst, line 49); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst, line 49); *backlink*

Unknown interpreted text role "meth".

- **Out of Order Message Delivery**: We do not assume message delivery order between any pair of nodes, because both sender and receiver are using multiple threads. There is no guarantee on which message will be processed first.

RRef Lifetime

The goal of the protocol is to delete an `OwnerRRef` at an appropriate time. The right time to delete an `OwnerRRef` is when there are no living `UserRRef` instances and user code is not holding references to the `OwnerRRef` either. The tricky part is to determine if there are any living `UserRRef` instances.

Design Reasoning

A user can get a `UserRRef` in three situations:

1. Receiving a `UserRRef` from the owner.
2. Receiving a `UserRRef` from another user.
3. Creating a new `UserRRef` owned by another worker.

Case 1 is the simplest where the owner passes its RRef to a user, where the owner calls :meth:`~torch.distributed.rpc.rpc_sync`, :meth:`~torch.distributed.rpc.rpc_async`, or :meth:`~torch.distributed.rpc.remote` and uses its RRef as an argument. In this case a new `UserRRef` will be created on the user. As the owner is the caller, it can easily update its local reference count on the `OwnerRRef`.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst, line 79); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst, line 79); *backlink*

Unknown interpreted text role "meth".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst, line 79); *backlink*

Unknown interpreted text role "meth".

The only requirement is that any `UserRRef` must notify the owner upon destruction. Hence, we need the first guarantee:

G1. The owner will be notified when any UserRRef is deleted.

As messages might come delayed or out-of-order, we need one more guarantee to make sure the delete message is not processed too soon. If A sends a message to B that involves an RRef, we call the RRef on A (the parent RRef) and the RRef on B (the child RRef).

G2. Parent RRef will NOT be deleted until the child RRef is confirmed by the owner.

In cases 2 and 3, it is possible that the owner has only partial or no knowledge at all about the RRef fork graph. For example, an

RRef could be constructed on a user, and before the owner receives any RPC call, the creator user might have already shared the RRef with other users, and those users could further share the RRef. One invariant is that the fork graph of any RRef is always a tree, because forking an RRef always creates a new `UserRRef` instance on the callee (except if the callee is the owner), and hence every RRef has a single parent.

The owner's view on any `UserRRef` in the tree has three stages:

```
1) unknown -> 2) known -> 3) deleted.
```

The owner's view of the entire tree keeps changing. The owner deletes its `OwnerRRef` instance when it thinks there are no living `UserRRef` instances, i.e., when `OwnerRRef` is deleted, all `UserRRef` instances could be either indeed deleted or unknown. The dangerous case is when some forks are unknown and others are deleted.

G2 trivially guarantees that no parent `UserRRef` can be deleted before the owner knows all of its children `UserRRef` instances. However, it is possible that the child `UserRRef` may be deleted before the owner knows its parent `UserRRef`.

Consider the following example, where the `OwnerRRef` forks to A, then A forks to Y, and Y forks to Z:

```
OwnerRRef -> A -> Y -> Z
```

If all of Z's messages, including the delete message, are processed by the owner before Y's messages. the owner will learn of Z's deletion befores knowing Y exists. Nevertheless, this does not cause any problem. Because, at least one of Y's ancestors will be alive (A) and it will prevent the owner from deleting the `OwnerRRef`. More specifically, if the owner does not know Y, A cannot be deleted due to **G2**, and the owner knows A since it is A's parent.

Things get a little trickier if the RRef is created on a user:

```
OwnerRRef
    ^
    |
    A -> Y -> Z
```

If Z calls :meth:`~torch.distributed.rpc.RRef.to_here` on the `UserRRef`, the owner at least knows A when Z is deleted, because otherwise, :meth:`~torch.distributed.rpc.RRef.to_here` wouldn't finish. If Z does not call :meth:`~torch.distributed.rpc.RRef.to_here`, it is possible that the owner receives all messages from Z before any message from A and Y. In this case, as the real data of the `OwnerRRef` has not been created yet, there is nothing to be deleted either. It is the same as Z does not exist at all. Hence, it's still OK.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, line 153); ***backlink***

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, line 153); ***backlink***

Unknown interpreted text role "meth".

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master) (docs) (source) (rpc)rref.rst`, line 153); ***backlink***

Unknown interpreted text role "meth".

Implementation

G1 is implemented by sending out a delete message in `UserRRef` destructor. To provide **G2**, the parent `UserRRef` is put into a context whenever it is forked, indexed by the new `ForkId`. The parent `UserRRef` is only removed from the context when it receives an acknowledgement message (ACK) from the child, and the child will only send out the ACK when it is confirmed by the owner.

Protocol Scenarios

Let's now discuss how the above designs translate to the protocol in four scenarios.

User Share RRef with Owner as Return Value

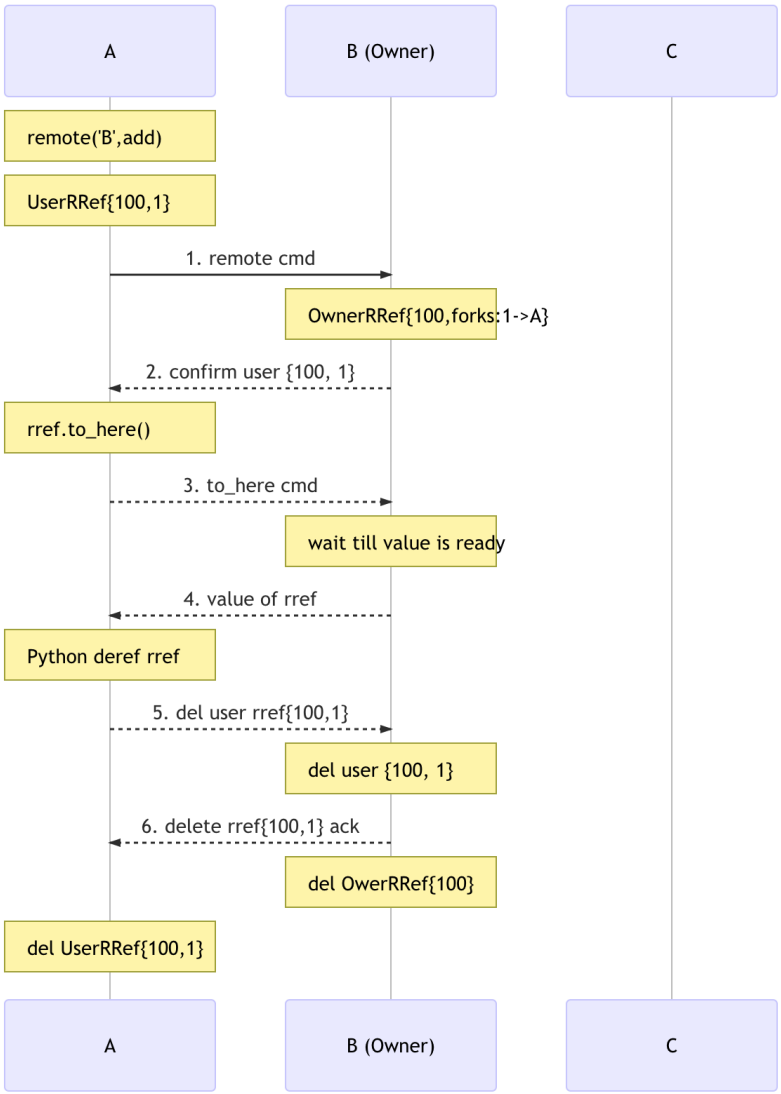
```
import torch
import torch.distributed.rpc as rpc

# on worker A
rref = rpc.remote('B', torch.add, args=(torch.ones(2), 1))
# say the rref has RRefId 100 and ForkId 1
rref.to_here()
```

In this case, the `UserRRef` is created on the user worker A, then it is passed to the owner worker B together with the remote message, and then B creates the `OwnerRRef`. The method :meth:`~torch.distributed.rpc.remote` returns immediately, meaning that the `UserRRef` can be forked/used before the owner knows about it.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst`, line 194); *backlink*
>
> Unknown interpreted text role "meth".

On the owner, when receiving the :meth:`~torch.distributed.rpc.remote` call, it will create the `OwnerRRef`, and returns an ACK to acknowledge `{100, 1}` (`RRefId`, `ForkId`). Only after receiving this ACK, can A delete its `UserRRef`. This involves both **G1** and **G2**. **G1** is obvious. For **G2**, the `OwnerRRef` is a child of the `UserRRef`, and the `UserRRef` is not deleted until it receives the ACK from the owner.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst`, line 200); *backlink*
>
> Unknown interpreted text role "meth".



The diagram above shows the message flow, where solid arrow contains user function and dashed arrow are builtin messages. Note that the first two messages from A to B (:meth:`~torch.distributed.rpc.remote` and :meth:`~torch.distributed.rpc.RRef.to_here`) may arrive at B in any order, but the final delete message will only be sent out when:

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst`, line 211); *backlink*
>
> Unknown interpreted text role "meth".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\pytorch-master\docs\source\rpc\(pytorch-master)(docs)(source)(rpc)rref.rst`, line 211); *backlink*

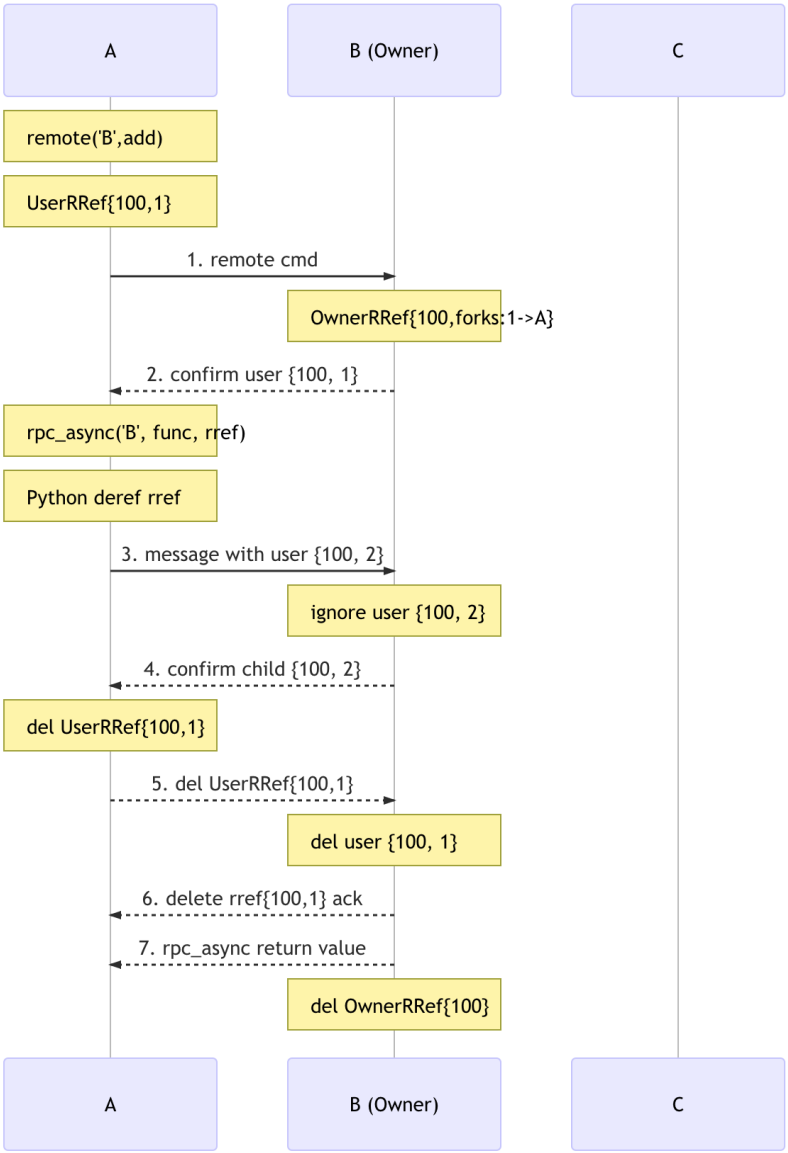
Unknown interpreted text role "meth".

- B acknowledges `UserRRef {100, 1}` (G2), and
- Python GC agrees to delete the local `UserRRef` instance. This occurs when the RRef is no longer in scope and is eligible for garbage collection.

User Share RRef with Owner as Argument

```
import torch
import torch.distributed.rpc as rpc

# on worker A and worker B
def func(rref):
  pass

# on worker A
rref = rpc.remote('B', torch.add, args=(torch.ones(2), 1))
# say the rref has RRefId 100 and ForkId 1
rpc.rpc_async('B', func, args=(rref, ))
```

In this case, after creating the `UserRRef` on A, A uses it as an argument in a followup RPC call to B. A will keep `UserRRef {100, 1}` alive until it receives the acknowledge from B (**G2**, not the return value of the RPC call). This is necessary because A should not send out the delete message until all previous messages are received, otherwise, the `OwnerRRef` could be deleted before usage as we do not guarantee message delivery order. This is done by creating a child `ForkId` of RRef, holding them in a map until receives the owner confirms the child `ForkId`. The figure below shows the message flow.



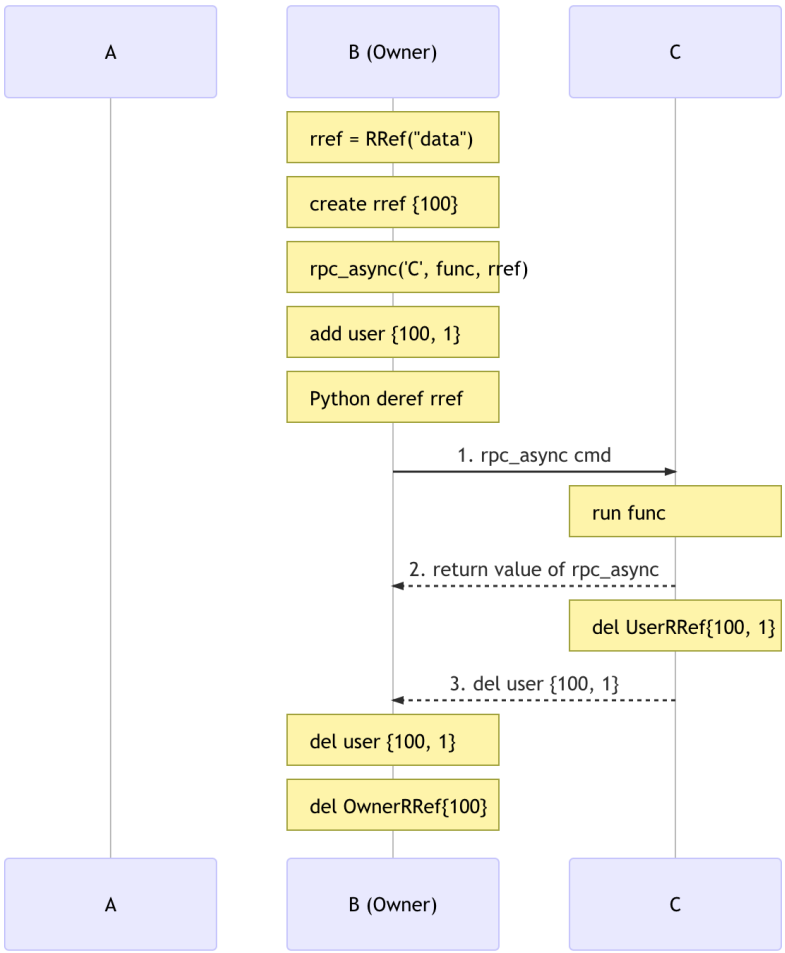
Note that the `UserRRef` could be deleted on B before func finishes or even starts. However this is OK, as at the time B sends out ACK for the child `ForkId`, it already acquired the `OwnerRRef` instance, which would prevent it been deleted too soon.

Owner Share RRef with User

Owner to user is the simplest case, where the owner can update reference counting locally, and does not need any additional control

message to notify others. Regarding **G2**, it is same as the parent receives the ACK from the owner immediately, as the parent is the owner.

```
import torch
import torch.distributed.rpc as RRef, rpc

# on worker B and worker C
def func(rref):
  pass

# on worker B, creating a local RRef
rref = RRef("data")
# say the rref has RRefId 100
dist.rpc_async('C', func, args=(rref, ))
```

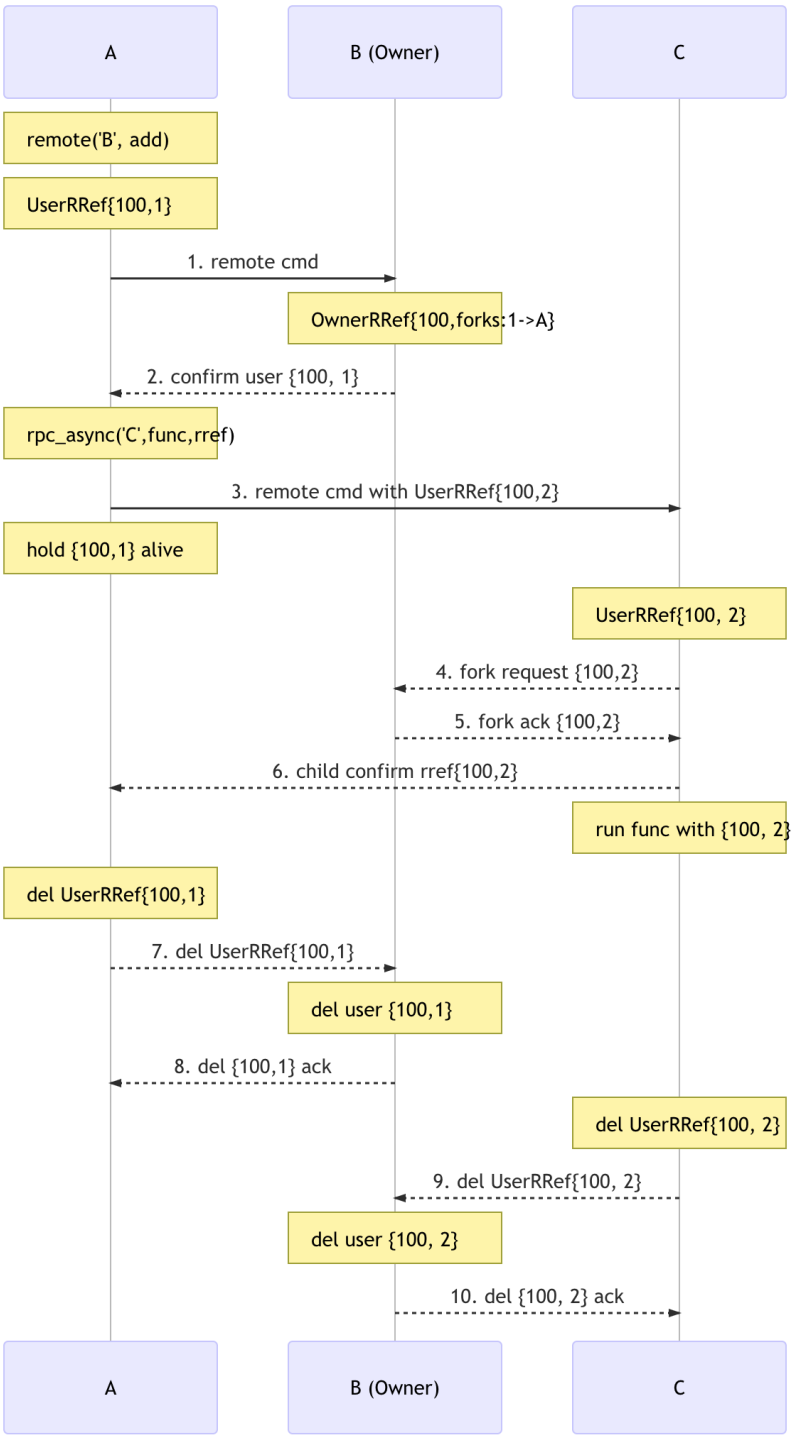


The figure above shows the message flow. Note that when the `OwnerRRef` exits scope after the rpc_async call, it will not be deleted, because internally there is a map to hold it alive if there is any known forks, in which case is `UserRRef {100, 1}`. (**G2**)

User Share RRef with User

This is the most complicated case where caller user (parent `UserRRef`), callee user (child `UserRRef`), and the owner all need to get involved.

```
import torch
import torch.distributed.rpc as rpc

# on worker A and worker C
def func(rref):
  pass

# on worker A
rref = rpc.remote('B', torch.add, args=(torch.ones(2), 1))
# say the rref has RRefId 100 and ForkId 1
rpc.rpc_async('C', func, args=(rref, ))
```



When C receives the child `UserRRef` from A, it sends out a fork request to the owner B. Later, when the B confirms the `UserRRef` on C, C will perform two actions in parallel: 1) send out the child ACK to A ,and 2) run the user provided function. During this time, the parent (A) will hold its `UserRRef {100, 1}` alive to achieve **G2**.