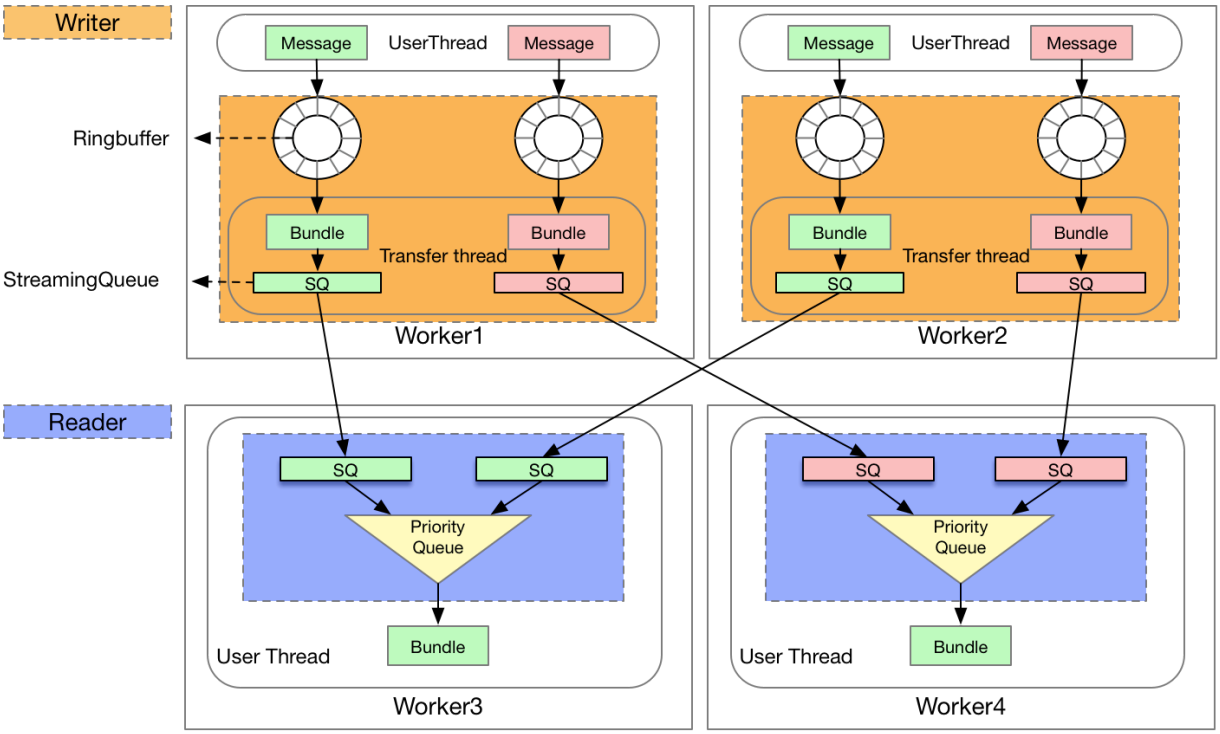
Streaming Data Transfer Proposal

This document describes the Data Transfer component of Ray Streaming mentioned in doc and our open source plan. The first part describes how ray streaming transfers data efficiently by introducing streaming writer/reader. The second part gives a detailed description of streaming queue which is the transport layer under streaming writer/reader. The third part extends our c++ implementations by providing a high level Java/Python interface.

1. Streaming Writer/Reader

Streaming writer/reader is implemented with C++ to transfer data between streaming workers. Each upstream worker has a streaming writer to write data to its all downstream workers, and each `downstream worker has a streaming reader to read data from its all upstream workers. A 2x2 example is shown as below.



1.1 Streaming Writer

From the writer's point of view, after the user writes the data, it does not immediately send the data to the downstream, but caches it in the corresponding memory ring buffer. There is a separate transfer thread to collect the messages from all the ring buffers, and write them to the corresponding transmission channel, which is backed by StreamingQueue(will be described later). The advantage is that the user thread will not be affected by the transmission speed of the data. And also the transfer thread can automatically batch the cached data from memory buffer into a data bundle to reduce transmission overhead. In addition, when there is no data in the ring buffer, it will also send an empty bundle, so downstream can know that and process accordingly. It will sleep for a short interval to save cpu if all ring buffers have no data.

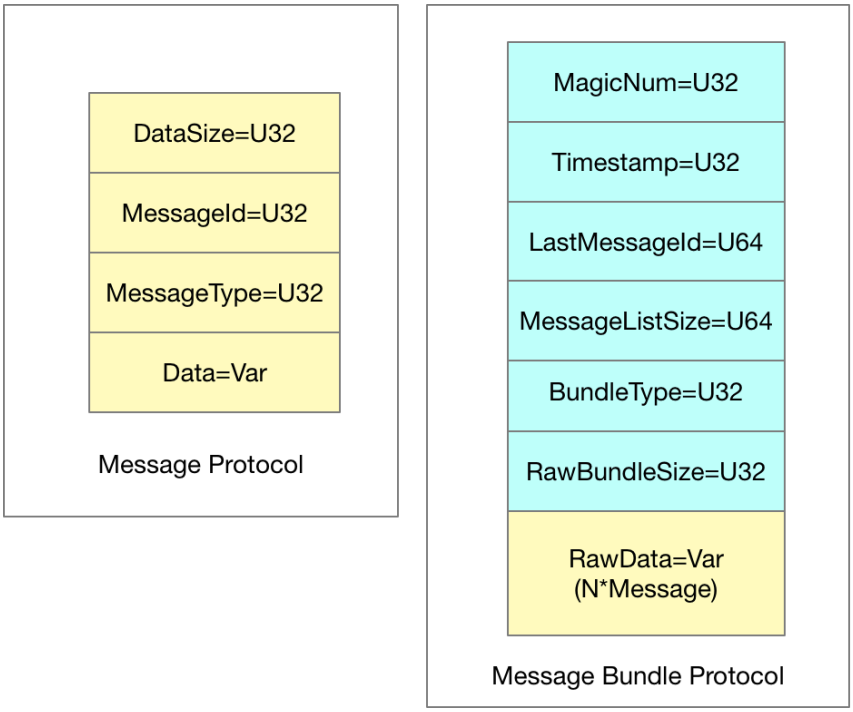
Since every memory ring buffer’s size is limited, when the writing buffer is full, the user thread will be blocked, which will cause back pressure naturally as we expect.

1.2 Streaming Reader

From the reader’s point of view, once invoked by user thread, it will fetch data bundles from all upstream workers, put them into a priority queue ordered by metadata(e.g. timestamp), then pop out the top bundle to user thread, so that the order of the message can be guaranteed, which will also facilitate our future implementation of fault tolerance. User thread can extract messages from the bundle and process one by one.

1.3 Message & Bundle Protocol

Since we have added some meta data for message and bundle, here we introduce message protocol and bundle protocol. U32 means unsigned int with 32 bits.



Message protocol:

- DataSize (raw data size)
- MessageId (this message id(0,INF])
- MessageType (a. Barrier = 1, b. Message =2)

Message bundle protocol:

- MagicNum = 0xcafebaba
- Timestamp 64bits timestamp (milliseconds from 1970)
- LastMessageId(the last id of bundle) (0,INF]
- MessageListSize(bundle len of message)
- BundleType(a. bundle = 3 , b. barrier =2, c. empty = 1)
- RawBundleSize（binary length of data)
- RawData (binary data)

2. StreamingQueue

This part aims to introduce streaming queue, the transport layer based on Ray's direct actor call and C++ Core Worker APIs. The primary goal of streaming queue is to transfer data between upstream producer and downstream consumer, including data and control messages. Meanwhile, it has the ability to cache data for future streaming fault tolerance.

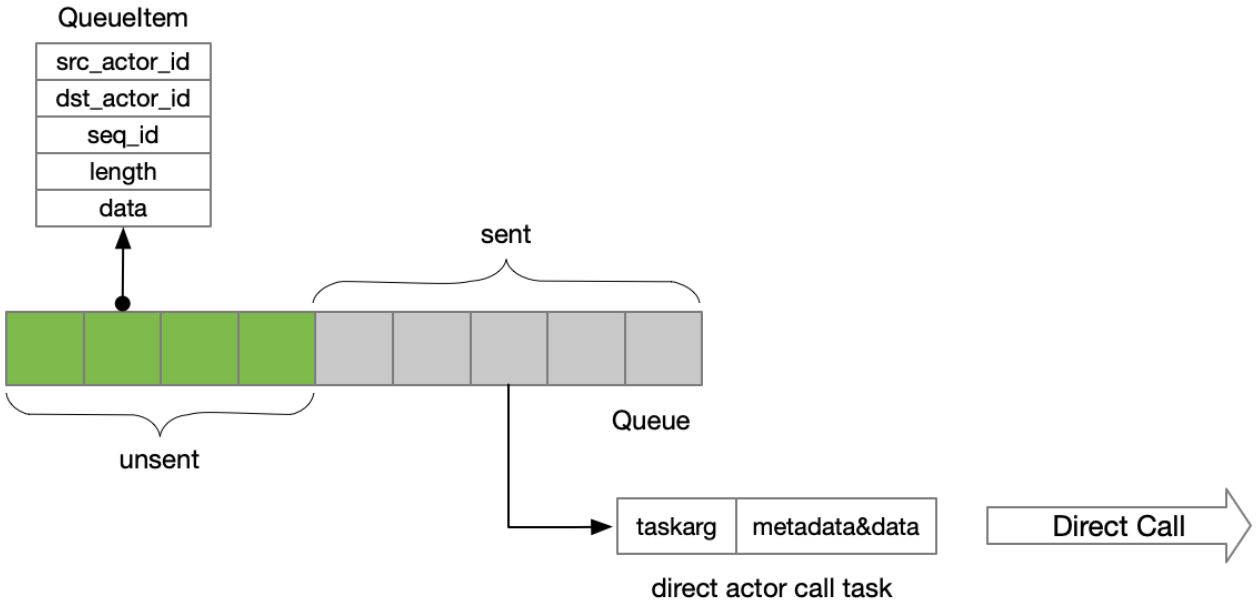
In practice, streaming queue is implemented in C++ with an upstream queue for producer, and a downstream queue for consumer, both of which are described as follows.

2.1 Upstream Queue

When producer pushes a data bundle into queue, the bundle is packed to an item with some meta, and pushed back into a std::list. The item will be sent to peer consumer immediately or later if failed to send. Each upstream queue is initialized with a max memory size limitation, PushQueueItem will return OutOfMemory when the upper limit is reached.

The items which has been sent, will not be evicted immediately, upstream queue maintains all items in internal list, until the peer consumer notifies deletion or the producer itself tells to delete some items, this will also help our future fault tolerance implementation.

The following picture shows how queue items are prepared and sent through direct actor call.



2.2 Downstream Queue

When queue item arrives, it is pushed back into a std::list, where the consumer reads items from.

The data structure is similar to upstream queue, so we can share the same implementation.

2.3 Message Categories

- Data Message, from up to down, to transfer data.
- Notify Message, from down to up, for the consumer to inform the producer of the consumed offset.
- GetLastItem Message, from up to down, for the producer to initiatively obtain consumer's consumed offset, to fail over.
- GetLastItemRsp Message, from down to up, the response to GetLastItem message.
- Pull Message, from down to up, request to replay items to downstream.
- PullRsp Message, from up to down, the response to Pull message.

```
table StreamingQueueDataMsg {
 src_actor_id: string;
 dst_actor_id: string;
 queue_id: string;
 seq_id: ulong;
 length: ulong;
}

table StreamingQueueNotificationMsg {
 src_actor_id: string;
 dst_actor_id: string;
 queue_id: string;
 seq_id: ulong;
}

table StreamingQueueGetLastMsgId {
 src_actor_id: string;
 dst_actor_id: string;
 queue_id: string;
}

table StreamingQueueGetLastMsgIdRsp {
 src_actor_id: string;
 dst_actor_id: string;
 Queue_id: string;
 seq_id: ulong;
 msg_id: ulong;
```

```
 err_code: StreamingQueueError;
}

table StreamingQueuePullRequestMsg {
 src_actor_id: string;
 dst_actor_id: string;
 queue_id: string;
 seq_id: ulong;
 async: bool;
}

table StreamingQueuePullResponseMsg {
 src_actor_id: string;
 dst_actor_id: string;
 queue_id: string;
 err_code: StreamingQueueError;
}

enum StreamingQueueError:int {
  OK,
  QUEUE_NOT_EXIST,
  NO_VALID_DATA_TO_PULL,
}
```

2.4 How Direct Actor Call Is Used

In order to use direct actor call to transfer data, the following things are required:

- The Core Worker pointer, the C++ pointer to call Core Worker's api. For java worker, the pointer can be obtained from RayNativeRuntime object through java reflection, it's a java long variable, cast to C++ pointer in jni; Python worker is similar.
- ActorID of the peer, which identifies the peer's destination. All these ids are prepared in the Ray Streaming DAG graph.
- Remote function descriptor of the peer. Java/Python functions must be used instead of C++ functions at present, due to the lack of C++ direct call. When Java/Python receives direct actor call, args are delivered to C++ through JNI and Cython.

In current implementation, asynchronous and synchronous version remote functions are both provided, the former for data messages, the latter for control messages. The function descriptor looks like this:

```
package org.ray.streaming.runtime;
```

```java
@RayRemote
public class JobWorker implements Serializable {
  /**
   * Entrypoint function for streaming data transfer.
   * Triggered in peer's streaming C++ layer through CoreWorker SubmitActorTask.
   * @param buffer param in flatbuffer format
   */
  public void onStreamingTransfer(byte[] buffer) {
  }

  /**
   * Synchronous version of onStreamingTransfer
   * @param buffer param in flatbuffer format
   * @return return value in flatbuffer format
   */
  public byte[] onStreamingTransferSync(byte[] buffer) {
  }
}
```

Among many interfaces of Core Worker, we mainly use SubmitActorTask() to send our messages, Wait() and Get() to get return objects for synchronous calls. The following code snippet shows how these APIs called.

```
/// Prepare arguments required by core worker
std::unordered_map<std::string, double> resources;
TaskOptions options{1, resources};
RayFunction func{
 ray::Language::JAVA,
 { "org.ray.streaming.runtime", "onStreamingTransferSync", "(J[B)[B"}
};

char meta_data[3] = {'R', 'A', 'W'};
std::shared_ptr<LocalMemoryBuffer> meta =
      std::make_shared<LocalMemoryBuffer>((uint8_t *)meta_data, 3, true);

/// Serialize out message to buffer and put into TaskArg
PullRequestMessage msg(actor_id, peer_actor_id, queue_id, start_seq_id);
std::unique_ptr<LocalMemoryBuffer> buffer = msg.ToBytes();

std::vector<TaskArg> args;
args.emplace_back(
      TaskArg::PassByValue(std::make_shared<RayObject>(buffer, meta, true)));

/// Call core worker api SubmitActorTask to send out
std::vector<ObjectID> return_ids;
```

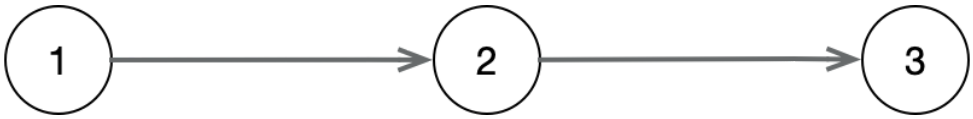
```
ray::Status st = core_worker_->SubmitActorTask(peer_actor_id_, func, args,
options, &return_ids);
if (!st.ok()) {
  STREAMING_LOG(ERROR) << "SubmitActorTask fail.";
  return;
}

std::vector<bool> wait_results;
Status wait_st = core_worker_->Objects().Wait(return_ids, 1, timeout_ms,
&wait_results);
if (!wait_st.ok()) {
  STREAMING_LOG(ERROR) << "Wait fail.";
  return nullptr;
}
std::vector<std::shared_ptr<RayObject>> results;
Status get_st = core_worker_->Objects().Get(return_ids, -1, &results);
if (!get_st.ok()) {
  STREAMING_LOG(ERROR) << "Get fail.";
  return;
}

if (results[0]->IsException()) {
  STREAMING_LOG(INFO) << "peer actor may has exceptions, should retry.";
  return;
}

/// Get return value and parse to our message
std::shared_ptr<Buffer> result_buffer = results[0]->GetData();
std::shared_ptr<LocalMemoryBuffer> return_buffer =
std::make_shared<LocalMemoryBuffer>(
      result_buffer->Data(), result_buffer->Size(), true);

std::shared_ptr<Message> result_msg = ParseMessage(return_buffer);
STREAMING_CHECK(result_msg->Type() ==
                  queue::flatbuf::MessageType::StreamingQueuePullResponseMsg);
std::shared_ptr<PullResponseMessage> response_msg =
    std::dynamic_pointer_cast<PullResponseMessage>(result_msg);
```

All payloads transferred through direct actor call are packed into a unified package, consisting of flatbuf-formatted metadata and data, including data and control messages. The general payload format is:

src_actor_id	dst_actor_id	queue_id	magicnum	type	data

2.5 Threading Model

For intermediate nodes in the DAG graph, similar to 2, as the following picture shows.



Upstream and downstream queues coexist in worker process, so the data messages and control messages arrive at the same time. To avoid making the ray call thread too busy, we use a separate thread(an asio service) called queue thread to handle messages. When ray call thread receives a message, just post it into the queue thread and return immediately.

3. Java/Python Integration

In java/python, we add an abstraction of QueueLink/QueueProducer/QueueConsumer to adapt c++ data transfer. QueueLink is used to create QueueProducer/QueueConsumer, and it also behaves as an entrypoint of direct actor call used by streaming queue, as current Ray call only supports Java/Python remote function. We use jni to call c++ data transfer in java and cython in python. We also add a simple memory QueueLink implementation in java/python for unit test convenience in single process mode. When it is used, data transfer won't call to streaming c++ layer.

```
public interface QueueLink {
  /**
   * Create queue producer of output queues
   *
   * @param outputQueueIds output queue ids
   * @param outputActorHandles output actor handles
   * @return queue producer
   */
  QueueProducer registerQueueProducer(Collection<String> outputQueueIds, Map<String, Long>
outputActorHandles);
```

```
  /**
   * Create queue consumer of input queues
   *
   * @param inputQueueIds     input queue ids
   * @param inputActorHandles input actor handles
   * @return queue consumer
   */
  QueueConsumer registerQueueConsumer(Collection<String> inputQueueIds, Map<String, Long>
inputActorHandles);

  /**
   * used in direct call mode, called by JobWorker.onStreamingTransfer
   */
  void onQueueTransfer(byte[] buffer);

  /**
   * used in direct call mode, called by JobWorker.onStreamingTransferSync
   */
  byte[] onQueueTransferSync(byte[] buffer);
}


public interface QueueProducer {
  /**
   * produce msg into the specified queue
   *
   * @param id   queue id
   * @param item message item data section is specified by [position, limit).
   */
  void produce(QueueID id, ByteBuffer item);
}


public interface QueueConsumer {
  /**
   * pull msg from input queues, if timeout, return null.
   *
   * @param timeoutMillis timeout
   * @return message or null
   */
  QueueItem pull(long timeoutMillis);
}
```