
Ray Core Documentation

Release 0.1

The Ray team

July 17, 2016

1	Quick Start Guide	3
2	The Ray shell	5
3	Plasma: Storing objects in memory	7
3.1	The Buffer interface	7
3.2	The Plasma client interface	8
3.3	Plasma metadata	9
3.4	An example application	10
4	Numbuf: Fast serialization of numerical data	11
5	Connecting to Ray services	13
6	Developer Documentation for Numbuf	15
7	Developer documentation for Plasma	17
7.1	The IPC interface	17
7.2	Internal classes	17
8	Developer documentation for Ray	21
8.1	Client connections to the Shell	21
9	Indices and tables	23

Welcome to the documentation of Ray Core! The documentation consists of two parts:

- the system user documentation which describes the system API and how to use the system
- the developer documentation that describes the internals of the system and the developer API

These are separate from the Ray user documentation which can be found under <https://github.com/amplab/ray/blob/master/README.md>.

Ray Core user documentation:

Quick Start Guide

To build Ray Core, execute the following commands:

First, install the requirements:

```
sudo apt-get update
sudo apt-get install git subversion build-essential
sudo apt-get python-dev g++-multilib libcap-dev
```

Then, install depot_tools:

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
export PATH=`pwd`/depot_tools:"$PATH"
```

Check out and build the project:

```
git clone https://github.com/amplab/ray-core
cd ray-core
glclient sync
cd src
gn gen out/Debug
ninja -C out/Debug -j 16
```

To make sure everything works, you can try out the Mojo hello world example:

```
cd ray-core/src/out/Debug
./mojo_shell mojo:hello_mojo_client
```

Now, the Ray shell can be started with

```
cd ray-core/src/out/Debug
./mojo_shell --enable-multiprocess
               --external-connection-address=/home/ubuntu/shell
               ray_node_app.mojo
```

The Ray shell

The Ray shell is responsible for managing all the services that are running on a given node, like the local scheduler, the Plasma store and the Python workers. There is one shell per node.

You can start the shell using

```
./mojo_shell --enable-multiprocess
              --external-connection-address=/home/ubuntu/shell
              ray_node_app.mojo
```

This starts `ray_node_app.mojo`, which starts the object store and listens on the socket `/home/ubuntu/shell` to establish connections to Python and C++ clients.

Plasma: Storing objects in memory

Plasma is a shared region of memory that allows multiple processes running on the same machine to access shared data objects.

It can be used both as a Ray service and a library in your own programs.

An object is created in two distinct phases:

1. Allocate memory and write data into allocated memory. If the size of the data is not known in advance, the buffer can be resized. Note that during this phase the buffer is writable, but only by its creator. No one else can access the buffer during this phase.
2. Seal the buffer. Once the creator finishes writing data into buffer it seals the buffer. From this moment on the buffer becomes immutable and other processes can read it.

To create an object, the user specifies a unique identifier for the object and an optional name. Plasma keeps track of the process id that created the object, the creation time stamp, how long creation of the object took and the size of the object. During creation, the user can also specify metadata that will be associated with the object.

Other processes can request an object by its unique identifier (later also by name). If the object has not been created or sealed yet, the process requesting the object will block until the object has been sealed.

3.1 The Buffer interface

A buffer is the region of memory associated to a data object, as determined by a start address and a size in bytes. There are two kinds of buffers, read-only buffers and read-write buffers.

class `plasma::Buffer`

Read-only view on data

Subclassed by *`plasma::MutableBuffer`*

Public Functions

const `uint8_t *data ()`

Return the start address of the buffer.

const `uint8_t *data (uint64_t offset)`

Return an address corresponding to an “offset” in this buffer

`int64_t size ()`

Return the size of the object in bytes

MutableBuffers have a richer interface, they allow writing to and resizing the object. When the object creator has finished modifying the object, it calls the `Seal` method to make the object immutable, which allows other processes to read the object.

class `plasma::MutableBuffer`

Mutable view on data

Inherits from `plasma::Buffer`

Public Functions

MutableBuffer ()

After the default constructor has been called, the class is not functional and all methods will raise errors.

Only after it has been initialized by `ClientContext::BuildObject` can this class be used.

`uint8_t *mutable_data` ()

Return the start address of the buffer (mutable).

`uint8_t *mutable_data` (`uint64_t offset`)

Return an address corresponding to an “offset” in this buffer (mutable).

Status **Resize** (`int64_t new_size`)

Resize the buffer.

Parameters

- `new_size` - New size of the buffer (in bytes).

Status **Seal** ()

Make the data contained in this buffer immutable. After the buffer has been sealed, it is illegal to modify data from the buffer or to resize the buffer.

`bool sealed` ()

Has this `MutableBuffer` been sealed?

3.2 The Plasma client interface

The developer interacts with Plasma through the Plasma API. Each process needs to instantiate a `ClientContext`, which will give the process access to objects and their metadata and allow them to write objects.

class `plasma::ClientContext`

A client context is the primary interface through which clients interact with Plasma.

Public Functions

ClientContext (`const std::string &address`)

Create a new client context.

Parameters

- `address` - Address of the Ray shell socket we are connecting to

Status **BuildObject** (ObjectID *object_id*, int64_t *size*, *MutableBuffer* &*buffer*, const std::string &*name* = "", const std::map<std::string, *Buffer*> &*metadata* = EMPTY)

Build a new object. Building an object involves multiple steps. Once the creator process finishes to construct the objects, it seals the object. Only after that can it be shared with other processes.

Parameters

- *object_id* - The object ID of the newly created objects. Provided by the client, which must ensure it is globally unique.
- *size* - The number of bytes that are allocated for the object initially. Can be reallocated through the *MutableBuffer* interface.
- *buffer* - The function will pass the allocated buffer to the client using this argument.
- *name* - An optional name for the object through which it can be accessed without knowing its object ID.
- *metadata* - An optional dictionary of metadata for the object. The keys of the dictionary are strings and the values are arbitrary binary data represented by *Buffer* objects.

Status **GetObject** (ObjectID *object_id*, *Buffer* &*buffer*)

Get buffer associated to an object ID. If the object has not been sealed yet, this function will block the current thread.

Parameters

- *object_id* - The object ID of the object that shall be retrieved.
- *buffer* - The argument is used to pass the read-only buffer to the client.

Status **ListObjects** (std::vector<*ObjectInfo*> &*objects*)

Put object information of objects in the store into the vector objects.

Status **GetMetadata** (ObjectID *object_id*, const std::string &*key*, *Buffer* &*data*)

Retrieve metadata for a given object.

Return A view on the metadata associated to that key.

Parameters

- *key* - The key of the metadata information to be retrieved.

3.3 Plasma metadata

There are two parts to the object metadata: One internally maintained by Plasma and one provided by the user. The first part is represented by the *ObjectInfo* class.

```
class plasma::ObjectInfo
```

Public Members

std::string **name**

Name of the object as provided by the user during object construction.

int64_t **size**

Size of the object in bytes.

`int64_t create_time`

Time when object construction started, in microseconds since the Unix epoch.

`int64_t construct_duration`

Time in microseconds between object creation and sealing.

`int64_t creator_id`

Process ID of the process that created the object.

`std::string creator_address`

Cluster wide unique address for the process that created the object.

Each object has a small dictionary that can hold metadata provided by users. Users can provide arbitrary information here. It is most likely going to be used to store information like `format` (`binary`, `arrow`, `protobuf`, `json`) and `schema`, which could hold a schema for the data.

3.4 An example application

We are going to have more examples here. Currently, the best way of understanding the API is by looking at `libplasma`, the Python C extension for Plasma. It can be found in <https://github.com/amplab/ray-core/blob/master/src/plasma/client/plasma.cc>.

Note that this is not the Python API that users will interact with. It can be used like this:

```
import libplasma

plasma = libplasma.connect("/home/pcmoritz/shell")

A = libplasma.build_object(plasma, 1, 1000, "object-1")
libplasma.seal_object(A)
B = libplasma.build_object(plasma, 2, 2000, "object-2")
libplasma.seal_object(B)

libplasma.list_objects(plasma)
```

Numbuf: Fast serialization of numerical data

Numbuf is a library for the fast serialization of primitive Python objects (lists, tuples, dictionaries, NumPy arrays) to the [Apache Arrow](#) format.

Connecting to Ray services

All the Ray services expose an API via IPC that can be called by any other services or applications. To learn more about services, please have a look at <https://www.chromium.org/developers/design-documents/mojo> and <https://github.com/amplab/ray-core/tree/master/src/docs>.

template <typename *Service*>

class *shell::ClientContext*

The *ClientContext* is used to connect a client to a Ray service. The “Service” template parameter is the service class generated by *mojom* for this service (for example *mojo::examples::Echo* for the Mojo “echo” example).

Public Functions

void ConnectToShell (**const** std::string &*service_name*, **const** std::string &*address*)
Connect this client context to the Ray shell.

Parameters

- *service_name* - The name of the service you want to connect to
- *address* - The address to the shell socket

mojo::SynchronousInterfacePtr<*Service*> **GetInterface** ()
Get the Mojo Interface pointer for this connection.

Ray Core developer documentation:

Developer Documentation for Numbuf

Numbuf is a library for the fast serialization of primitive Python objects (lists, tuples, dictionaries, NumPy arrays) to the [Apache Arrow](#) format.

class `numbuf::DictBuilder`

Constructing dictionaries of key/value pairs. Sequences of keys and values are built separately using a pair of `SequenceBuilders`. The resulting Arrow representation can be obtained via the `Finish` method.

Public Functions

SequenceBuilder &**keys** ()

Builder for the keys of the dictionary.

SequenceBuilder &**vals** ()

Builder for the values of the dictionary.

`std::shared_ptr<arrow::StructArray>` **Finish** (`std::shared_ptr<arrow::Array>` *list_data*,
`std::shared_ptr<arrow::Array>` *dict_data*)

Construct an Arrow `StructArray` representing the dictionary. Contains a field “keys” for the keys and “vals” for the values.

Parameters

- *list_data* - List containing the data from nested lists in the value list of the dictionary
- *dict_data* - List containing the data from nested dictionaries in the value list of the dictionary

class `numbuf::SequenceBuilder`

Public Functions

Status **Append** ()

Appending a none to the list.

Status **Append** (`bool data`)

Appending a boolean to the list.

Status **Append** (`int64_t data`)

Appending an `int64_t` to the list.

Status **Append** (`const char *data`)

Appending a null-delimited string to the list.

arrow::Status **Append** (const std::string &*data*)
Appending a C++ string to the list.

Status **Append** (float *data*)
Appending a float to the list.

Status **Append** (double *data*)
Appending a double to the list.

Status **Append** (const std::vector<int64_t> &*dims*, double **data*)
Appending a tensor to the list

Parameters

- *dims* - A vector of dimensions
- *data* - A pointer to the start of the data block. The length of the data block will be the product of the dimensions

Status **AppendList** (int32_t *size*)
Add a sublist to the list. The data contained in the list will be specified in the “Finish” method.

To construct `l = [[11, 22], 33, [44, 55]]` you would for example run `list = ListBuilder(); list.AppendList(2); list.Append(33); list.AppendList(2); list.Finish([11, 22, 44, 55]); list.Finish();`

Parameters

- *size* - The size of the sublist

std::shared_ptr<DenseUnionArray> **Finish** (std::shared_ptr<arrow::Array> *list_data*,
std::shared_ptr<arrow::Array> *dict_data*)
Finish building the list and return the result.

template <typename *T*>

class numbuf::TensorBuilder

This is a class for building a dataframe where each row corresponds to a Tensor (= multidimensional array) of numerical data. There are two columns, “dims” which contains an array of dimensions for each Tensor and “data” which contains data buffer of the Tensor as a flattened array.

Public Functions

Status **Append** (const std::vector<int64_t> &*dims*, const elem_type **data*)
Append a new tensor.

Parameters

- *dims* - The dimensions of the Tensor
- *data* - Pointer to the beginning of the data buffer of the Tensor. The total length of the buffer is `sizeof(elem_type) * product of dims[i] over i`

std::shared_ptr<Array> **Finish** ()
Convert the tensors to an Arrow StructArray.

int32_t **length** ()
Number of tensors in the column.

Developer documentation for Plasma

7.1 The IPC interface

```

module plasma.service;

struct ObjectInfo {
    string name;
    uint64 size;
    int64 create_time;
    int64 construct_delta;
    int64 creator_id;
};

[ServiceName="plasma::service::Plasma"]
interface Plasma {
    CreateObject(int64 object_id, uint64 size, string name, int64 creator_id)
        => (handle<shared_buffer> buffer);
    ResizeObject(int64 object_id, uint64 new_size)
        => (handle<shared_buffer> buffer);
    SealObject(int64 object_id);
    GetObject(int64 object_id, bool block)
        => (handle<shared_buffer> buffer, uint64 size);
    ListObjects()
        => (array<ObjectInfo> info);
};

```

7.2 Internal classes

class `plasma::service::PlasmaEntry`

An entry in the hash table of objects stored in the local object store.

Public Members

`mojo::ScopedSharedBufferHandle` **handle**

Handle to the shared memory buffer where the object is stored.

`ObjectInfoPtr` **object_info**

ObjectInfo (see `plasma.mojom`)

class `plasma::service::PlasmaImpl`

Implementation of the Plasma service interface. This implementation is single threaded, which means we do not have to lock the datastructures.

Inherits from Plasma

Public Functions

void CreateObject (`int64 object_id`, `uint64 size`, **const** `mojo::String &name`, `int64 creator_id`, **const** `CreateObjectCallback &callback`)

Creates a new object..

Return Shared memory handle to the read-write memory of the object

Parameters

- `object_id` - Unique identifier of the object to be build
- `size` - Initial number of bytes to be allocated for the object
- `name` - User defined name of the object

void pass_sealed_object (`int64 object_id`, **const** `GetObjectCallback &callback`)

Pass a sealed object to a client that has been waiting.

void SealObject (`int64 object_id`)

Seal an object, making it immutable.

Parameters

- `object_id` - Unique identifier of the object to be sealed

void GetObject (`int64 object_id`, `bool block`, **const** `GetObjectCallback &callback`)

Get an object from the object store.

Return Handle to the object and size of the object in bytes

Parameters

- `object_id` - Unique identifier of the object that shall be returned
- `block` - If true, this call will block until the object becomes available. Otherwise, if the object is not in the object store, an error will be raised.

void ListObjects (**const** `ListObjectsCallback &callback`)

List objects from the object store.

Return A list of `ObjectInfoData` objects that describe the objects in the store.

class `plasma::service::PlasmaServerApp`

Implementation of the Plasma server. This follows the “SingletonServer” pattern in examples/echo/echo_server.cc (see documentation there). This means that the object store is shared between all the clients running on a given node.

Inherits from `ApplicationImplBase`

Public Functions

bool **OnAcceptConnection** (mojo::ServiceProviderImpl **service_provider_impl*)
Accept a new connection from a client.

Developer documentation for Ray

8.1 Client connections to the Shell

Most of the complexity of this code comes from the fact that we need to be able to connect to the Ray shell from an outside process (i.e. a Python process) that was started independently of the Ray shell. This is not supported in Mojo, they use fork to start child processes.

class ray::FileDescriptorSender

Send a file descriptor of a process to another process. This is needed because Mojo bootstraps the IPC communication between processes via a file handle (this makes sure no artifacts like actual files remain on the computer once the IPC has finished).

Public Functions

FileDescriptorSender (const std::string &address)

Initialize the *FileDescriptorSender*.

Parameters

- address - Address of the socket that is used to send the file descriptor

bool **Send** (int file_descriptor, const std::string &payload)

Send the file descriptor over the socket.

Return Bool that indicates if the sending was successful

Parameters

- file_descriptor - The file descriptor that will be sent
- payload - Additional payload that can be sent (< MAX_PAYLOAD_SIZE)

class ray::FileDescriptorReceiver

Receive a file descriptor from another process. This is needed because Mojo bootstraps the IPC communication between processes via a file handle (to make sure no artifacts like actual files remain on the computer once the IPC has finished).

Public Functions

int **Receive** (std::string &payload)
Receive file descriptor from the socket.

Return The file descriptor that was sent or -1 if not successful.

Parameters

- payload - The payload carried by this receive will be appended to this string

template <typename *Service*>

class shell::ServiceConnectionApp

The *ServiceConnectionApp* runs in a separate thread in the client and maintains a connection to the shell. It allows the client to connect synchronously to services, one service per *ServiceConnectionApp*. It allows the client to get InterfaceHandles to these services. These handles can be transferred to any client thread.

Inherits from ApplicationImplBase

Public Functions

ServiceConnectionApp (const std::string &service_name, std::condition_variable *notify_caller,
mojo::InterfaceHandle<Service> *service_handle)
Construct a new *ServiceConnectionApp* that will connect to a service.

Parameters

- service_name - The name of the service we want to connect to
- notify_caller - Condition that will be triggered to notify the calling thread that the connection to the service is established
- service_handle - A pointer to an InterfaceHandle that is owned by the calling thread

Indices and tables

- `genindex`
- `modindex`
- `search`

N

numbuf::DictBuilder (C++ class), 15
 numbuf::DictBuilder::Finish (C++ function), 15
 numbuf::DictBuilder::keys (C++ function), 15
 numbuf::DictBuilder::vals (C++ function), 15
 numbuf::SequenceBuilder (C++ class), 15
 numbuf::SequenceBuilder::Append (C++ function), 15, 16
 numbuf::SequenceBuilder::AppendList (C++ function), 16
 numbuf::SequenceBuilder::Finish (C++ function), 16
 numbuf::TensorBuilder (C++ class), 16
 numbuf::TensorBuilder::Append (C++ function), 16
 numbuf::TensorBuilder::Finish (C++ function), 16
 numbuf::TensorBuilder::length (C++ function), 16

P

plasma::Buffer (C++ class), 7
 plasma::Buffer::data (C++ function), 7
 plasma::Buffer::size (C++ function), 7
 plasma::ClientContext (C++ class), 8
 plasma::ClientContext::BuildObject (C++ function), 8
 plasma::ClientContext::ClientContext (C++ function), 8
 plasma::ClientContext::GetMetadata (C++ function), 9
 plasma::ClientContext::GetObject (C++ function), 9
 plasma::ClientContext::ListObjects (C++ function), 9
 plasma::MutableBuffer (C++ class), 8
 plasma::MutableBuffer::mutable_data (C++ function), 8
 plasma::MutableBuffer::MutableBuffer (C++ function), 8
 plasma::MutableBuffer::Resize (C++ function), 8
 plasma::MutableBuffer::Seal (C++ function), 8
 plasma::MutableBuffer::sealed (C++ function), 8
 plasma::ObjectInfo (C++ class), 9
 plasma::ObjectInfo::construct_duration (C++ member), 10
 plasma::ObjectInfo::create_time (C++ member), 9
 plasma::ObjectInfo::creator_address (C++ member), 10
 plasma::ObjectInfo::creator_id (C++ member), 10
 plasma::ObjectInfo::name (C++ member), 9
 plasma::ObjectInfo::size (C++ member), 9

plasma::service::PlasmaEntry (C++ class), 17
 plasma::service::PlasmaEntry::handle (C++ member), 17
 plasma::service::PlasmaEntry::object_info (C++ member), 17
 plasma::service::PlasmaImpl (C++ class), 17
 plasma::service::PlasmaImpl::CreateObject (C++ function), 18
 plasma::service::PlasmaImpl::GetObject (C++ function), 18
 plasma::service::PlasmaImpl::ListObjects (C++ function), 18
 plasma::service::PlasmaImpl::pass_sealed_object (C++ function), 18
 plasma::service::PlasmaImpl::SealObject (C++ function), 18
 plasma::service::PlasmaServerApp (C++ class), 18
 plasma::service::PlasmaServerApp::OnAcceptConnection (C++ function), 19

R

ray::FileDescriptorReceiver (C++ class), 21
 ray::FileDescriptorReceiver::Receive (C++ function), 22
 ray::FileDescriptorSender (C++ class), 21
 ray::FileDescriptorSender::FileDescriptorSender (C++ function), 21
 ray::FileDescriptorSender::Send (C++ function), 21

S

shell::ClientContext (C++ class), 13
 shell::ClientContext::ConnectToShell (C++ function), 13
 shell::ClientContext::GetInterface (C++ function), 13
 shell::ServiceConnectionApp (C++ class), 22
 shell::ServiceConnectionApp::ServiceConnectionApp (C++ function), 22