Creating a Distributed Application

Table of contents

Distributed applications refer to those where the workflow is divided into multiple fragments that may be run on separate nodes. For example, data might be collected via a sensor at the edge, sent to a separate workstation for processing, and then the processed data could be sent back to the edge node for visualization. Each node would run a single fragment consisting of a computation graph built up of operators. Thus one fragment is the equivalent of a non-distributed application. In the distributed context, the Application initializes the different fragments and then defines the connections between them to build up the full distributed application workflow.

In this section we'll describe:

- how to define a distributed Application
- how to build and run a distributed application

Defining a Distributed Application Class

> **Tip**
>
> Defining distributed applications is also illustrated in the video_replayer_distributed and ping_distributed examples. The `ping_distributed` examples also illustrate how to update C++ or Python applications to parse user-defined arguments in a way that works without disrupting support for distributed application command line arguments (e.g. `--driver`, `--worker`).

Defining a single Fragment (`C++` / `Python`) involves adding operators using `make_operator()` (`C++`) or the operator constructor (`Python`), and defining the connections between them using the `add_flow()` method (`C++` / `Python`) in the `compose()` method. Thus, defining a Fragment is just like defining a non-distributed Application except that the class should inherit from Fragment instead of Application.

The application will then be defined by initializing fragments within the application's `compose()` method. The `add_flow()` method (`C++` / `Python`) can be used to define the connections across fragments.

Ingested Tab Module

Serialization of Custom Data Types for Distributed Applications

Transmission of data between fragments of a multi-fragment application is done via the Unified Communications X (UCX) library. In order to transmit data, it must be serialized into a binary form suitable for transmission over a network. For Tensors (`C++` / `Python`), strings and various scalar and vector numeric types, serialization is already built in. For more details on concrete examples of how to extend the data serialization support to additional user-defined classes, see the separate page on serialization.

Building and running a Distributed Application

Ingested Tab Module

Running an application in a distributed setting requires launching the application binary on all nodes involved in the distributed application. A single node must be selected to act as the application driver. This is achieved by using the `--driver` command-line option. Worker nodes are initiated by launching the application with the `--worker` command-line option. It's possible for the driver node to also serve as a worker if both options are specified.

The address of the driver node must be specified for each process (both the driver and worker(s)) to identify the appropriate network interface for communication. This can be done via the `--address` command-line option, which takes a value in the form of `[&lt;IPv4/IPv6 address or hostname&gt;][:&lt;port&gt;]` (e.g., `--address 192.168.50.68:10000`):

- The driver's IP (or hostname) **MUST** be set for each process (driver and worker(s)) when running distributed applications on multiple nodes (default: `0.0.0.0`). It can be set without the port (e.g., `--address 192.168.50.68`).
- In a single-node application, the driver's IP (or hostname) can be omitted, allowing any network interface (`0.0.0.0`) to be selected by the UCX library.
- The port is always optional (default: `8765`). It can be set without the IP (e.g., `--address :10000`).

The worker node's address can be defined using the `--worker-address` command-line option (`[<IPv4/IPv6 address or hostname>][:<port>]`). If it's not specified, the application worker will default to the host address (`0.0.0.0`) with a randomly chosen port number between `10000` and `32767` that is not currently in use. This argument automatically sets the `HOLOSCAN_UCX_SOURCE_ADDRESS` environment variable if the worker address is a local IP address. Refer to Environment Variables for Distributed Applications for details.

The `--fragments` command-line option is used in combination with `--worker` to specify a comma-separated list of fragment names to be run by a worker. If not specified, the application driver will assign a single fragment to the worker. To indicate that a worker should run all fragments, you can specify `--fragments all` .

The `--config` command-line option can be used to designate a path to a configuration file to be used by the application.

Below is an example launching a three fragment application named `my_app` on two separate nodes:

- The application driver is launched at `192.168.50.68:10000` on the first node (A), with a worker running two fragments, "fragment1" and "fragment3".
- On a separate node (B), the application launches a worker for "fragment2", which will connect to the driver at the address above.

Ingested Tab Module

Note

UCX Network Interface Selection

UCX is used in the Holoscan SDK for communication across fragments in distributed applications. It is designed to select the best network device based on performance characteristics (bandwidth, latency, NUMA locality, etc). In some scenarios (under investigation) UCX cannot find the correct network interface to use, and the application fails to run. In this case, you can manually specify the

network interface to use by setting the `UCX_NET_DEVICES` environment variable.

For example, if the user wants to use the network interface `eth0`, you can set the environment variable as follows, before running the application:

```
export UCX_NET_DEVICES=eth0
```

Or, if you are running a packaged distributed application with the Holoscan CLI, use the `--nic eth0` option to manually specify the network interface to use.

The available network interface names can be found by running the following command:

```
ucx_info -d | grep Device: | awk '{print $3}' | sort | uniq # or ip -o -4 addr show | awk '{print $2, $4}' # to show interface name and IP
```

⚠ Warning

Known limitations

The following are known limitations of the distributed application support in the SDK, which will be addressed in future updates:

1. A connection error message is displayed even when the distributed application is running correctly.

The message
`Connection dropped with status -25 (Connection reset by remote peer)`
appears in the console even when the application is functioning properly. This is a known issue and will be addressed in future

updates, ensuring that this message will only be displayed in the event of an actual connection error.

2. GPU tensors can only currently be sent/received by UCX from a single device on a given node.

By default, device ID 0 is used by the UCX extensions to send/receive data between fragments. To override this default, the user can set environment variable `HOLOSCAN_UCX_DEVICE_ID`.

3. “Address already in use” errors in distributed applications due to the health check service.

In scenarios where distributed applications have both the driver and workers running on the same host, either within a Docker container or directly on the host, there’s a possibility of encountering “Address already in use” errors. A potential solution is to assign a different port number to the `HOLOSCAN_HEALTH_CHECK_PORT` environment variable (default: `8777`), for example, by using `export HOLOSCAN_HEALTH_CHECK_PORT=8780`.

Note

GXF UCX Extension

Holoscan’s distributed application feature makes use of the GXF UCX Extension. Its documentation may provide useful additional context into how data is transmitted between fragments.

Tip

Given a CMake project, a pre-built executable, or a python application, you can also use the Holoscan CLI to package and run your Holoscan application in a OCI-compliant container image.

Environment Variables for Distributed Applications

Holoscan SDK environment variables.

You can set environment variables to modify the default actions of services and the scheduler when executing a distributed application.

- **HOLOSCAN_ENABLE_HEALTH_CHECK** : determines if the health check service should be active, even without specifying `--driver` or `--worker` in the CLI. By default, initiating the AppDriver (`--driver`) or AppWorker (`--worker`) service automatically triggers the GRPC Health Checking Service so grpc-health-probe can monitor liveness/readiness. Interprets values like "true", "1", or "on" (case-insensitive) as true (to enable the health check). It defaults to false if left unspecified.

- **HOLOSCAN_HEALTH_CHECK_PORT** : designates the port number on which the Health Checking Service is launched. It must be an integer value representing a valid port number. If unspecified, it defaults to `8777` .

- **HOLOSCAN_DISTRIBUTED_APP_SCHEDULER** : controls which scheduler is used for distributed applications. It can be set to either `greedy` , `multi_thread` or `event_based` . `multithread` is also allowed as a synonym for `multi_thread` for backwards compatibility. If unspecified, the default scheduler is `multi_thread` .

- **HOLOSCAN_STOP_ON_DEADLOCK** : can be used in combination with `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` to control whether or not the application will automatically stop on deadlock. Values of "True", "1" or "ON" will be interpreted as true (enable stop on deadlock). It is true if unspecified. This environment variable is only used when `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` is explicitly set.

- **HOLOSCAN_STOP_ON_DEADLOCK_TIMEOUT** : controls the delay (in ms) without activity required before an application is considered to be in deadlock. It must be an

integer value (units are ms).

- **HOLOSCAN_MAX_DURATION_MS** : sets the application to automatically terminate after the requested maximum duration (in ms) has elapsed. It must be an integer value (units are ms). This environment variable is only used when `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` is explicitly set.

- **HOLOSCAN_CHECK_RECESSION_PERIOD_MS** : controls how long (in ms) the scheduler waits before re-checking the status of operators in an application. It must be a floating point value (units are ms). This environment variable is only used when `HOLOSCAN_DISTRIBUTED_APP_SCHEDULER` is explicitly set.

- **HOLOSCAN_UCX_SERIALIZATION_BUFFER_SIZE** : can be used to override the default 7 kB serialization buffer size. This should typically not be needed as tensor types store only a small header in this buffer to avoid explicitly making a copy of their data. However, other data types do get directly copied to the serialization buffer and in some cases it may be necessary to increase it.

- **HOLOSCAN_UCX_DEVICE_ID** : The GPU ID of the device that will be used by UCX transmitter/receivers in distributed applications. If unspecified, it defaults to 0. A list of discrete GPUs available in a system can be obtained via `nvidia-smi -L`. GPU data sent between fragments of a distributed application must be on this device.

- **HOLOSCAN_UCX_PORTS** : This defines the preferred port numbers for the SDK when specific ports for UCX communication need to be predetermined, such as in a Kubernetes environment. If the distributed application requires three ports (UCX receivers) and the environment variable is unset, the SDK chooses three unused ports sequentially from the range 10000~32767. Specifying a value, for example, `HOLOSCAN_UCX_PORTS=10000`, results in the selection of ports 10000, 10001, and 10002. Multiple starting values can be comma-separated. The system increments from the last provided port if more ports are needed. Any unused specified ports are ignored.

- **HOLOSCAN_UCX_SOURCE_ADDRESS** : This environment variable specifies the local IP address (source) for the UCX connection. This variable is especially beneficial when a node has multiple network interfaces, enabling the user to determine which one should be utilized for establishing a UCX client (UCXTransmitter). If it is not explicitly specified, the default address is set to `0.0.0.0`, representing any available interface.

UCX-specific environment variables

Transmission of data between fragments of a multi-fragment application is done via the Unified Communications X (UCX) library, a point-to-point communication framework designed to utilize the best available hardware resources (shared memory, TCP, GPUDirect RDMA, etc). UCX has many parameters that can be controlled via environment variables. A few that are particularly relevant to Holoscan SDK distributed applications are listed below:

- The `UCX_TLS` environment variable can be used to control which transport layers are enabled. By default, `UCX_TLS=all` and UCX will attempt to choose the optimal transport layer automatically.

- The `UCX_NET_DEVICES` environment variable is by default set to `all` meaning that UCX may choose to use any available network interface controller (NIC). In some cases it may be necessary to restrict UCX to a specific device or set of devices, which can be done by setting `UCX_NET_DEVICES` to a comma separated list of the device names (i.e. as obtained by linux command `ifconfig -a` or `ip link show`).

- Setting `UCX_TCP_CM_REUSEADDR=y` is recommended to enable ports to be reused without having to wait the full socket TIME_WAIT period after a socket is closed.

- The `UCX_LOG_LEVEL` environment variable can be used to control the logging level of UCX. The default is setting is WARN, but changing to a lower level such as INFO will provide more verbose output on which transports and devices are being used.

- By default, Holoscan SDK will automatically set `UCX_PROTO_ENABLE=y` upon application launch to enable the newer "v2" UCX protocols. If for some reason, the older v1 protocols are needed, one can set `UCX_PROTO_ENABLE=n` in the environment to override this setting. When the v2 protocols are enabled, one can optionally set `UCX_PROTO_INFO=y` to enable detailed logging of what protocols are being used at runtime.

- By default, Holoscan SDK will automatically set `UCX_MEMTYPE_CACHE=n` upon application launch to disable the UCX memory type cache (See UCX documentation for more information. It can cause about 0.2 microseconds of pointer type checking overhead with the cudacudaPointerGetAttributes() CUDA API). If for some reason, the memory type cache is needed, one can set `UCX_MEMTYPE_CACHE=y` in the environment to override this setting.

- By default, the Holoscan SDK will automatically set `UCX_CM_USE_ALL_DEVICES=n` at application startup to disable consideration of all devices for data transfer. If for some reason the opposite behavior is desired, one can set `UCX_CM_USE_ALL_DEVICES=y` in the environment to override this setting. Setting `UCX_CM_USE_ALL_DEVICES=n` can be used to workaround an issue where UCX sometimes defaults to a device that might not be the most suitable for data transfer based on the host's available devices. On a host with address 10.111.66.60, UCX, for instance, might opt for the `br-80572179a31d` (192.168.49.1) device due to its superior bandwidth as compared to `eno2` (10.111.66.60). With `UCX_CM_USE_ALL_DEVICES=n`, UCX will ensure consistency by using the same device for data transfer that was initially used to establish the connection. This ensures more predictable behavior and can avoid potential issues stemming from device mismatches during the data transfer process.

- Setting `UCX_TCP_PORT_RANGE=&lt;start&gt;-&lt;end&gt;` can be used to define a specific range of ports that UCX should utilize for data transfer. This is particularly useful in environments where ports need to be predetermined, such as in a Kubernetes setup. In such contexts, Pods often have ports that need to be exposed, and these ports must be specified ahead of time. Moreover, in scenarios where firewall configurations are stringent and only allow specified ports, having a predetermined range ensures that the UCX communication does not get blocked. This complements the `HOLOSCAN_UCX_SOURCE_ADDRESS`, which specifies the local IP address for the UCX connection, by giving further control over which ports on that specified address should be used. By setting a port range, users can ensure that UCX operates within the boundaries of the network and security policies of their infrastructure.

Tip

A list of all available UCX environment variables and a brief description of each can be obtained by running `ucx_info -f` from the Holoscan SDK container. Holoscan SDK uses UCX's active message (AM) protocols, so environment variables related to other protocols such as tag-mat

Serialization

Distributed applications must serialize any objects that are to be sent between the fragments of a multi-fragment application. Serialization involves binary serialization to a buffer that will be sent from one fragment to another via the Unified Communications X (UCX) library. For tensor types (e.g. holoscan::Tensor), no actual copy is made, but instead transmission is done directly from the original tensor's data and only a small amount of header information is copied to the serialization buffer.

A table of the types that have codecs pre-registered so that they can be serialized between fragments using Holoscan SDK is given below.

Type Class	Specific Types
integers	int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t
floating point	float, double, complex , complex
boolean	bool
strings	std::string
std::vector	T is std::string or any of the boolean, integer or floating point types above
std::vector >	T is std::string or any of the boolean, integer or floating point types above
std::vector	a vector of InputSpec objects that are specific to HolovizOp
std::shared_ptr<%>	T is any of the scalar, vector or std::string types above
tensor types	holoscan::Tensor, nvidia::gxf::Tensor, nvidia::gxf::VideoBuffer, nvidia::gxf::AudioBuffer
GXF-specific types	nvidia::gxf::TimeStamp, nvidia::gxf::EndOfStream

> ⚠ **Warning**
>
> If an operator transmitting both CPU and GPU tensors is to be used in distributed applications, the same output port cannot mix both GPU and CPU tensors. CPU and GPU tensor outputs should be placed

on separate output ports. This is a limitation of the underlying UCX library being used for zero-copy tensor serialization between operators.

As a concrete example, assume an operator, `MyOperator` with a single output port named “out” defined in it’s setup method. If the output port is only ever going to connect to other operators within a fragment, but never across fragments then it is okay to have a `TensorMap` with a mixture of host and device arrays on that single port.

Ingested Tab Module

However, this mixing of CPU and GPU arrays on a single port will not work for distributed apps and instead separate ports should be used if it is necessary for an operator to communicate across fragments.

Ingested Tab Module

Python

For the Python API, any array-like object supporting the DLPack interface, `__array_interface__` or `__cuda_array_interface__` will be transmitted using `Tensor` serialization. This is done to avoid data copies for performance reasons. Objects of type `list[holoscan.HolovizOp.InputSpec]` will be sent using the underlying C++ serializer for `std::vector&lt;HolovizOp::InputSpec&gt;`. All other Python objects will be serialized to/from a `std::string` using the cloudpickle library.

⚠ Warning

A restriction imposed by the use of cloudpickle is that all fragments in a distributed application must be running the same Python version.

⚠ Warning

Distributed applications behave differently than single fragment applications when

```
<a
href="api/python/holoscan_python_api_core.html#holoscan.core.OutputContext.e
</a>
```

is called to emit a tensor-like Python object. Specifically, for array-like objects such as a PyTorch tensor, the same Python object will **not** be received by any call to

```
<a
href="api/python/holoscan_python_api_core.html#holoscan.core.InputContext.rec
</a>
```

in a downstream Python operator (even if the upstream and downstream operators are part of the same fragment). An object of type `holoscan.Tensor` will be received as a `holoscan.Tensor`. Any other array-like objects with data stored on device (GPU) will be received as a CuPy tensor. Similarly, any array-like object with data stored on the host (CPU) will be received as a NumPy array. The user must convert back to the original array-like type if needed (typically possible in a zero-copy fashion via DLPack or array interfaces).

C++

For any additional C++ classes that need to be serialized for transmission between fragments in a distributed application, the user must create their own codec and register it with the Holoscan SDK framework. As a concrete example, suppose that we had the following simple Coordinate class that we wish to send between fragments.

```
struct Coordinate { float x; float y; float z; };
```

To create a codec capable of serializing and deserializing this type one should define a `holoscan::codec` class for it as shown below.

```
#include "holoscan/core/codec_registry.hpp" #include "holoscan/core/errors.hpp"
#include "holoscan/core/expected.hpp" namespace holoscan { template <> struct
codec<Coordinate> { static expected<size_t, RuntimeError> serialize(const
Coordinate& value, Endpoint* endpoint) { return serialize_trivial_type<Coordinate>
```

```
(value, endpoint); } static expected<Coordinate, RuntimeError>
deserialize(Endpoint* endpoint) { return deserialize_trivial_type<Coordinate>
(endpoint); } }; } // namespace holoscan
```

where the first argument to `serialize` is a const reference to the type to be serialized and the return value is an `expected` containing the number of bytes that were serialized. The `deserialize` method returns an `expected` containing the deserialized object. The `Endpoint` class is a base class representing the serialization endpoint (For distributed applications, the actual endpoint class used is `UcxSerializationBuffer`).

The helper functions `serialize_trivial_type` (`deserialize_trivial_type`) can be used to serialize (deserialize) any plain-old-data (POD) type. Specifically, POD types can be serialized by just copying `sizeof(Type)` bytes to/from the endpoint. The `read_trivial_type()` and `~holoscan::Endpoint::write_trivial_type` methods could be used directly instead.

```
template <> struct codec<Coordinate> { static expected<size_t, RuntimeError>
serialize(const Coordinate& value, Endpoint* endpoint) { return endpoint-
>write_trivial_type(&value); } static expected<Coordinate, RuntimeError>
deserialize(Endpoint* endpoint) { Coordinate encoded; auto maybe_value =
endpoint->read_trivial_type(&encoded); if (!maybe_value) { return
forward_error(maybe_value); } return encoded; } };
```

In practice, one would not actually need to define `codec&lt;Coordinate&gt;` at all since `Coordinate` is a trivially serializable type and the existing `codec` treats any types for which there is not a template specialization as a trivially serializable type. It is, however, still necessary to register the codec type with the `CodecRegistry` as described below.

For non-trivial types, one will likely also need to use the `read()` and `write()` methods to implement the codec. Example use of these for the built-in codecs can be found in `holoscan/core/codecs.hpp`.

Once such a codec has been defined, the remaining step is to register it with the static `CodecRegistry` class. This will make the UCX-based classes used by distributed applications aware of the existence of a codec for serialization of this object type. If the

type is specific to a particular operator, then one can register it via the `register_codec()` class.

```
#include "holoscan/core/codec_registry.hpp" namespace holoscan::ops { void MyCoordinateOperator::initialize() { register_codec<Coordinate>("Coordinate"); // ... // parent class initialize() call must be after the argument additions above Operator::initialize(); } } // namespace holoscan::ops
```

Here, the argument provided to `register_codec` is the name the registry will use for the codec. This name will be serialized in the message header so that the deserializer knows which deserialization function to use on the received data. In this example, we chose a name that matches the class name, but that is not a requirement. If the name matches one that is already present in the `CodecRegistry` class, then any existing codec under that name will be replaced by the newly registered one.

It is also possible to directly register the type outside of the context of `initialize()` by directly retrieving the static instance of the codec registry as follows.

```
namespace holoscan { CodecRegistry::get_instance().add_codec<Coordinate> ("Coordinate"); } // namespace holoscan
```

Tip

CLI arguments (such as `--driver`, `--worker`, `--fragments`) are parsed by the `Application` (

```
<a href="api/cpp/classholoscan_1_1Application.html#_CPPv4N8holoscan11Applicatio </a>
```

/

```
<a href="api/python/holoscan_python_api_core.html#holoscan.core.Application">Pyt
```

) class and the remaining arguments are available as `app.argv` (

```
<a href="api/cpp/classholoscan_1_1Application.html#_CPPv4N8holoscan11Applicatio </a>
```

/
<a
href="api/python/holoscan_python_api_core.html#holoscan.core.Application.argv"
).

Ingested Tab Module

Adding user-defined command line arguments

When adding user-defined command line arguments to an application, one should avoid the use of any of the default command line argument names as `--help`, `--version`, `--config`, `--driver`, `--worker`, `--address`, `--worker-address`, `--fragments` as covered in the section on running a distributed application. It is recommended to parse user-defined arguments from the `argv` ((`C++` / `Python`)) method/property of the application as covered in the note above instead of using C++ `char* argv[]` or Python `sys.argv` directly. This way, only the new, user-defined arguments will need to be parsed.

A concrete example of this for both C++ and Python can be seen in the existing ping_distributed example where an application-defined boolean argument (`--gpu`) is specified in addition to the default set of application arguments.

Ingested Tab Module