NVIDIA TensorRT Inference Server Documentation

Release 0.12.0dev

NVIDIA Corporation

Feb 15, 2019

CONTENTS

Warning: You are currently viewing unstable developer preview of the documentation. To see the documentation for the latest stable release click here.

The NVIDIA TensorRT Inference Server provides a cloud inferencing solution optimized for NVIDIA GPUs. The server provides an inference service via an HTTP or gRPC endpoint, allowing remote clients to request inferencing for any model being managed by the server. The inference server provides the following features:

- Multiple framework support. The server can manage any number and mix of models (limited by system disk and memory resources). Supports TensorRT, TensorFlow GraphDef, TensorFlow SavedModel and Caffe2 NetDef model formats. Also supports TensorFlow-TensorRT integrated models.
- Custom backend support. The inference server allows individual models to be implemented with custom backends instead of by a deep-learning framework. With a custom backend a model can implement any logic desired, while still benefiting from the GPU support, concurrent execution, dynamic batching and other features provided by the server.
- The inference server monitors the model repository for any change and dynamically reloads the model(s) when necessary, without requiring a server restart. Models and model versions can be added and removed, and model configurations can be modified while the server is running.
- Multi-GPU support. The server can distribute inferencing across all system GPUs.
- Concurrent model execution support. Multiple models (or multiple instances of the same model) can run simultaneously on the same GPU.
- Batching support. For models that support batching, the server can accept requests for a batch of inputs and respond with the corresponding batch of outputs. The inference server also supports dynamic batching where individual inference requests are dynamically combined together to improve inference throughput. Dynamic batching is transparent to the client requesting inference.
- Model repositories may reside on a locally accessible file system (e.g. NFS) or in Google Cloud Storage.
- Readiness and liveness health endpoints suitable for any orchestration or deployment framework, such as Kubernetes.
- Metrics indicating GPU utiliization, server throughput, and server latency.

CHAPTER

ONE

QUICKSTART

The TensorRT Inference Server is available in two ways:

- As a pre-built Docker container container available from the NVIDIA GPU Cloud (NGC). For more information, see *Using A Prebuilt Docker Container*.
- As buildable source code located in GitHub. Building the inference server yourself requires the usage of Docker and the TensorFlow and PyTorch containers available from NGC. For more information, see *Building From Source Code*.

1.1 Prerequisites

Regardless of which method you choose (starting with a pre-built container from NGC or building from source), you must perform the following prerequisite steps:

- Ensure you have access and are logged into NGC. For step-by-step instructions, see the NGC Getting Started Guide.
- Install Docker and nvidia-docker. For DGX users, see Preparing to use NVIDIA Containers. For users other than DGX, see the nvidia-docker installation documentation.
- Clone the TensorRT Inference Server GitHub repo. Even if you choose to get the pre-built inference server from NGC, you need the GitHub repo for the example model repository and to build the example applications. Go to https://github.com/NVIDIA/tensorrt-inference-server, select the r<xx.yy> release branch that you are using, and then select the clone or download drop down button.
- Create a model repository containing one or more models that you want the inference server to serve. An example model repository is included in the docs/examples/model_repository directory of the GitHub repo. Before using the repository, you must fetch any missing model definition files from their public model zoos via the provided docs/examples/fetch_models.sh script:

  ```
  $ cd docs/examples
  $ ./fetch_models.sh
  ```

1.2 Using A Prebuilt Docker Container

Make sure you log into NGC as described in *Prerequisites* before attempting the steps in this section. Use docker pull to get the TensorRT Inference Server container from NGC:

```
$ docker pull nvcr.io/nvidia/tensorrtserver:<xx.yy>-py3
```

Where <xx.yy> is the version of the inference server that you want to pull. Once you have the container follow these steps to run the server and the example client applications.

1. *Run the inference server*.
2. *Verify that the server is running correct*.
3. *Build the example client applications*.
4. *Run the image classification example*.

1.3 Building From Source Code

Make sure you complete the steps in *Prerequisites* before attempting to build the inference server. To build the inference server from source, change to the root directory of the GitHub repo and use docker to build:

```
$ docker build --pull -t tensorrtserver
```

After the build completes follow these steps to run the server and the example client applications.

1. *Run the inference server*.
2. *Verify that the server is running correct*.
3. *Build the example client applications*.
4. *Run the image classification example*.

1.4 Run TensorRT Inference Server

Assuming the example model repository is available in /full/path/to/example/model/repository, use the following command to run the inference server container:

```
$ nvidia-docker run --rm --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 -
↪p8000:8000 -p8001:8001 -p8002:8002 -v/full/path/to/example/model/repository:/models
↪<docker image> trtserver --model-store=/models
```

Where <docker image> is *nvcr.io/nvidia/tensorrtserver:<xx.yy>-py3* if you pulled the inference server container from NGC, or is *tensorrtserver* if you built the inference server from source.

For more information, see *Running The Inference Server*.

1.5 Verify Inference Server Is Running Correctly

Use the server's *Status* endpoint to verify that the server and the models are ready for inference. From the host system use curl to access the HTTP endpoint to request the server status. For example:

```
$ curl localhost:8000/api/status
id: "inference:0"
version: "0.6.0"
uptime_ns: 23322988571
model_status {
  key: "resnet50_netdef"
  value {
```

(continues on next page)

(continued from previous page)

```
    config {
      name: "resnet50_netdef"
      platform: "caffe2_netdef"
    }
    ...
    version_status {
      key: 1
      value {
        ready_state: MODEL_READY
      }
    }
  }
}
ready_state: SERVER_READY
```

The ready_state field should return SERVER_READY to indicate that the inference server is online, that models are properly loaded, and that the server is ready to receive inference requests.

For more information, see *Checking Inference Server Status*.

1.6 Building The Client Examples

To build the C++ client library, C++ and Python examples, and a Python wheel file for the Python client library, change to the root directory of the GitHub repo and use docker to build:

```
$ docker build -t tensorrtserver_clients --target trtserver_build --build-arg "BUILD_
↪CLIENTS_ONLY=1" .
```

After the build completes, the tensorrtserver_clients Docker image will contain the built client libraries and examples. Run the client image so that the client examples can access the inference server running in its own container:

```
$ docker run -it --rm --net=host tensorrtserver_clients
```

For more information, see *Building the Client Libraries and Examples*.

1.7 Running The Image Classification Example

From within the tensorrtserver_clients image, run the example image-client application to perform image classification using the example resnet50_netdef from the example model repository.

To send a request for the resnet50_netdef (Caffe2) model from the example model repository for an image from the qa/images directory:

```
$ /opt/tensorrtserver/bin/image_client -m resnet50_netdef -s INCEPTION qa/images/mug.
↪jpg
Request 0, batch size 1
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.723991
```

The Python version of the application accepts the same command-line arguments:

```
$ python3 /workspace/src/clients/python/image_client.py -m resnet50_netdef -s␣
↪INCEPTION qa/images/mug.jpg
Request 0, batch size 1
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.778078556061
```

For more information, see *Image Classification Example Application*.

CHAPTER

TWO

INSTALLING THE SERVER

The TensorRT Inference Server is available as a pre-built Docker container or you can *build it from source*.

2.1 Installing Prebuilt Containers

The inference server is provided as a pre-built container on the NVIDIA GPU Cloud (NGC). Before pulling the container you must have access and be logged into the NGC container registry as explained in the NGC Getting Started Guide.

Before you can pull a container from the NGC container registry, you must have Docker and nvidia-docker installed. For DGX users, this is explained in Preparing to use NVIDIA Containers Getting Started Guide. For users other than DGX, follow the nvidia-docker installation documentation to install the most recent version of CUDA, Docker, and nvidia-docker.

After performing the above setup, you can pull the TensorRT Inference Server container using the following command:

```
docker pull nvcr.io/nvidia/tensorrtserver:19.01-py3
```

Replace *19.01* with the version of inference server that you want to pull.

CHAPTER

THREE

RUNNING THE SERVER

For best performance the TensorRT Inference Server should be run on a system that contains Docker, nvidia-docker, CUDA and one or more supported GPUs, as explained in *Running The Inference Server*. The inference server can also be run on non-CUDA, non-GPU systems as described in *Running The Inference Server On A System Without A GPU*.

3.1 Example Model Repository

Before running the TensorRT Inference Server, you must first set up a model repository containing the models that the server will make available for inferencing.

An example model repository containing a Caffe2 ResNet50, a TensorFlow Inception model, and a simple TensorFlow GraphDef model (used by the *simple_client example*) are provided in the docs/examples/model_repository directory. Before using the example model repository you must fetch any missing model definition files from their public model zoos:

```
$ cd docs/examples
$ ./fetch_models.sh
```

3.2 Running The Inference Server

Before running the inference server, you must first set up a model repository containing the models that the server will make available for inferencing. Section *Model Repository* describes how to create your own model repository. You can also use *Example Model Repository* to set up an example model repository.

Assuming the sample model repository is available in /path/to/model/repository, the following command runs the container you pulled from NGC or built locally:

```
$ nvidia-docker run --rm --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 -
↪p8000:8000 -p8001:8001 -p8002:8002 -v/path/to/model/repository:/models
↪<tensorrtserver image name> trtserver --model-store=/models
```

Where <*tensorrtserver image name*> will be something like **nvcr.io/nvidia/tensorrtserver:19.01-py3** if you *pulled the container from the NGC registry*, or **tensorrtserver** if you *built it from source*.

The nvidia-docker -v option maps /path/to/model/repository on the host into the container at /models, and the –model-store option to the server is used to point to /models as the model repository.

The -p flags expose the container ports where the inference server listens for HTTP requests (port 8000), listens for GRPC requests (port 8001), and reports Prometheus metrics (port 8002).

The –shm-size and –ulimit flags are recommended to improve the server's performance. For –shm-size the minimum recommended size is 1g but larger sizes may be necessary depending on the number and size of models being served.

For more information on the Prometheus metrics provided by the inference server see *Metrics*.

3.3 Running The Inference Server On A System Without A GPU

On a system without GPUs, the inference server should be run using docker instead of nvidia-docker, but is otherwise identical to what is described in *Running The Inference Server*:

```
$ docker run --rm --shm-size=1g --ulimit memlock=-1 --ulimit stack=67108864 -
↪p8000:8000 -p8001:8001 -p8002:8002 -v/path/to/model/repository:/models
↪<tensorrtserver image name> trtserver --model-store=/models
```

Because a GPU is not available, the inference server will be unable to load any model configuration that requires a GPU or that specified a GPU instance by an *instance-group* configuration.

3.4 Checking Inference Server Status

The simplest way to verify that the inference server is running correctly is to use the Status API to query the server's status. From the host system use *curl* to access the HTTP endpoint to request server status. The response is protobuf text showing the status for the server and for each model being served, for example:

```
$ curl localhost:8000/api/status
id: "inference:0"
version: "0.6.0"
uptime_ns: 23322988571
model_status {
  key: "resnet50_netdef"
  value {
    config {
      name: "resnet50_netdef"
      platform: "caffe2_netdef"
    }
    ...
    version_status {
      key: 1
      value {
        ready_state: MODEL_READY
      }
    }
  }
}
ready_state: SERVER_READY
```

This status shows configuration information as well as indicating that version 1 of the resnet50_netdef model is MODEL_READY. This means that the server is ready to accept inferencing requests for version 1 of that model. A model version ready_state will show up as MODEL_UNAVAILABLE if the model failed to load for some reason.

CHAPTER

FOUR

CLIENT LIBRARIES AND EXAMPLES

The inference server *client libraries* make it easy to communicate with the TensorRT Inference Server from you C++ or Python application. Using these libraries you can send either HTTP or GRPC requests to server to check status or health and to make inference requests.

A couple of example applications show how to use the client libraries to perform image classification and to test performance:

- C++ and Python versions of *image_client*, an example application that uses the C++ or Python client library to execute image classification models on the TensorRT Inference Server.
- Python version of *grpc_image_client*, an example application that is functionally equivalent to *image_client* but that uses GRPC generated client code to communicate with the inference server (instead of the client library).
- C++ version of *perf_client*, an example application that issues a large number of concurrent requests to the inference server to measure latency and throughput for a given model. You can use this to experiment with different model configuration settings for your models.

4.1 Building the Client Libraries and Examples

The provided Dockerfile can be used to build just the client libraries and examples. Issue the following command to build the C++ client library, C++ and Python examples, and a Python wheel file for the Python client library:

```
$ docker build -t tensorrtserver_clients --target trtserver_build --build-arg "BUILD_
↪CLIENTS_ONLY=1" .
```

You can optionally add *–build-arg "PYVER=<ver>"* to set the Python version that you want the Python client library built for. Supported values for <*ver*> are 2.6 and 3.5, with 3.5 being the default.

After the build completes the tensorrtserver_clients docker image will contain the built client libraries and examples. The easiest way to try the examples described in the following sections is to run the client image with –net=host so that the client examples can access the inference server running in its own container (see *Running The Inference Server* for more information about running the inference server):

```
$ docker run -it --rm --net=host tensorrtserver_clients
```

In the client image you can find the example executables in /opt/tensorrtserver/bin, and the Python wheel in /opt/tensorrtserver/pip.

If your host sytem is Ubuntu-16.04, an alternative to running the examples within the tensorrtserver_clients container is to instead copy the libraries and examples from the docker image to the host system:

```
$ docker run -it --rm -v/tmp:/tmp/host tensorrtserver_clients
# cp /opt/tensorrtserver/bin/image_client /tmp/host/.
# cp /opt/tensorrtserver/bin/perf_client /tmp/host/.
# cp /opt/tensorrtserver/bin/simple_client /tmp/host/.
# cp /opt/tensorrtserver/pip/tensorrtserver-*.whl /tmp/host/.
# cp /opt/tensorrtserver/lib/librequest.* /tmp/host/.
```

You can now access the files from /tmp on the host system. To run the C++ examples you must install some dependencies on your host system:

```
$ apt-get install curl libcurl3-dev libopencv-dev libopencv-core-dev
```

To run the Python examples you will need to additionally install the client whl file and some other dependencies:

```
$ apt-get install python3 python3-pip
$ pip3 install --user --upgrade tensorrtserver-*.whl numpy pillow
```

4.2 Image Classification Example Application

The image classification example that uses the C++ client API is available at src/clients/c++/image_client.cc. The Python version of the image classification client is available at src/clients/python/image_client.py.

To use image_client (or image_client.py) you must first have a running inference server that is serving one or more image classification models. The image_client application requires that the model have a single image input and produce a single classification output. If you don't have a model repository with image classification models see *Example Model Repository* for instructions on how to create one.

Follow the instructions in *Running The Inference Server* to launch the server using the model repository. Once the server is running you can use the image_client application to send inference requests to the server. You can specify a single image or a directory holding images. Here we send a request for the resnet50_netdef model from the *example model repository* for an image from the qa/images directory:

```
$ /opt/tensorrtserver/bin/image_client -m resnet50_netdef -s INCEPTION qa/images/mug.
↪jpg
Request 0, batch size 1
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.723991
```

The Python version of the application accepts the same command-line arguments:

```
$ python3 /workspace/src/clients/python/image_client.py -m resnet50_netdef -s␣
↪INCEPTION qa/images/mug.jpg
Request 0, batch size 1
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.778078556061
```

The image_client and image_client.py applications use the inference server client library to talk to the server. By default image_client instructs the client library to use HTTP protocol to talk to the server, but you can use GRPC protocol by providing the -i flag. You must also use the -u flag to point at the GRPC endpoint on the inference server:

```
$ /opt/tensorrtserver/bin/image_client -i grpc -u localhost:8001 -m resnet50_netdef -
↪s INCEPTION qa/images/mug.jpg
Request 0, batch size 1
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.723991
```

By default the client prints the most probable classification for the image. Use the -c flag to see more classifications:

```
$ /opt/tensorrtserver/bin/image_client -m resnet50_netdef -s INCEPTION -c 3 qa/images/
↪mug.jpg
Request 0, batch size 1
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.723991
    968 (CUP) = 0.270953
    967 (ESPRESSO) = 0.00115996
```

The -b flag allows you to send a batch of images for inferencing. The image_client application will form the batch from the image or images that you specified. If the batch is bigger than the number of images then image_client will just repeat the images to fill the batch:

```
$ /opt/tensorrtserver/bin/image_client -m resnet50_netdef -s INCEPTION -c 3 -b 2 qa/
↪images/mug.jpg
Request 0, batch size 2
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.778078556061
    968 (CUP) = 0.213262036443
    967 (ESPRESSO) = 0.00293014757335
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.778078556061
    968 (CUP) = 0.213262036443
    967 (ESPRESSO) = 0.00293014757335
```

Provide a directory instead of a single image to perform inferencing on all images in the directory:

```
$ /opt/tensorrtserver/bin/image_client -m resnet50_netdef -s INCEPTION -c 3 -b 2 qa/
↪images
Request 0, batch size 2
Image '../qa/images/car.jpg':
    817 (SPORTS CAR) = 0.836187
    511 (CONVERTIBLE) = 0.0708251
    751 (RACER) = 0.0597549
Image '../qa/images/mug.jpg':
    504 (COFFEE MUG) = 0.723991
    968 (CUP) = 0.270953
    967 (ESPRESSO) = 0.00115996
Request 1, batch size 2
Image '../qa/images/vulture.jpeg':
    23 (VULTURE) = 0.992326
    8 (HEN) = 0.00231854
    84 (PEACOCK) = 0.00201471
Image '../qa/images/car.jpg':
    817 (SPORTS CAR) = 0.836187
    511 (CONVERTIBLE) = 0.0708251
    751 (RACER) = 0.0597549
```

The grpc_image_client.py application at available at src/clients/python/grpc_image_client.py behaves the same as the image_client except that instead of using the inference server client library it uses the GRPC generated client library to communicate with the server.

4.3 Performance Example Application

The perf_client example application located at src/clients/c++/perf_client.cc uses the C++ client API to send concurrent requests to the server to measure latency and inferences per second under varying client loads.

To use perf_client you must first have a running inference server that is serving one or more models. The perf_client application works with any type of model by sending random data for all input tensors and by reading and ignoring all output tensors. If you don't have a model repository see *Example Model Repository* for instructions on how to create one.

Follow the instructions in *Running The Inference Server* to launch the inference server using the model repository.

The perf_client application has two major modes. In the first mode you specify how many concurrent clients you want to simulate and perf_client finds a stable latency and inferences/second for that level of concurrency. Use the -t flag to control concurrency and -v to see verbose output. The following example simulates four clients continuously sending requests to the inference server:

```
$ /opt/tensorrtserver/bin/perf_client -m resnet50_netdef -p3000 -t4 -v
*** Measurement Settings ***
  Batch size: 1
  Measurement window: 3000 msec

Request concurrency: 4
  Pass [1] throughput: 207 infer/sec. Avg latency: 19268 usec (std 910 usec)
  Pass [2] throughput: 206 infer/sec. Avg latency: 19362 usec (std 941 usec)
  Pass [3] throughput: 208 infer/sec. Avg latency: 19252 usec (std 841 usec)
  Client:
    Request count: 624
    Throughput: 208 infer/sec
    Avg latency: 19252 usec (standard deviation 841 usec)
    Avg HTTP time: 19224 usec (send 714 usec + response wait 18486 usec + receive 24␣
↪usec)
  Server:
    Request count: 749
    Avg request latency: 17886 usec (overhead 55 usec + queue 26 usec + compute 17805␣
↪usec)
```

In the second mode perf_client will generate an inferences/second vs. latency curve by increasing concurrency until a specific latency limit or concurrency limit is reached. This mode is enabled by using the -d option and -l to specify the latency limit and optionally the -c to specify a maximum concurrency limit:

```
$ /opt/tensorrtserver/bin/perf_client -m resnet50_netdef -p3000 -d -l50 -c 3
*** Measurement Settings ***
  Batch size: 1
  Measurement window: 3000 msec
  Latency limit: 50 msec
  Concurrency limit: 3 concurrent requests

Request concurrency: 1
  Client:
    Request count: 327
    Throughput: 109 infer/sec
    Avg latency: 9191 usec (standard deviation 822 usec)
    Avg HTTP time: 9188 usec (send/recv 1007 usec + response wait 8181 usec)
  Server:
    Request count: 391
    Avg request latency: 7661 usec (overhead 90 usec + queue 68 usec + compute 7503␣
↪usec)
```

(continues on next page)

(continued from previous page)

```
Request concurrency: 2
  Client:
    Request count: 521
    Throughput: 173 infer/sec
    Avg latency: 11523 usec (standard deviation 616 usec)
    Avg HTTP time: 11448 usec (send/recv 711 usec + response wait 10737 usec)
  Server:
    Request count: 629
    Avg request latency: 10018 usec (overhead 70 usec + queue 41 usec + compute 9907␣
↪usec)

Request concurrency: 3
  Client:
    Request count: 580
    Throughput: 193 infer/sec
    Avg latency: 15518 usec (standard deviation 635 usec)
    Avg HTTP time: 15487 usec (send/recv 779 usec + response wait 14708 usec)
  Server:
    Request count: 697
    Avg request latency: 14083 usec (overhead 59 usec + queue 30 usec + compute 13994␣
↪usec)

Inferences/Second vs. Client Average Batch Latency
Concurrency: 1, 109 infer/sec, latency 9191 usec
Concurrency: 2, 173 infer/sec, latency 11523 usec
Concurrency: 3, 193 infer/sec, latency 15518 usec
```

Use the -f flag to generate a file containing CSV output of the results:

```
$ /opt/tensorrtserver/bin/perf_client -m resnet50_netdef -p3000 -d -l50 -c 3 -f perf.
↪csv
```

You can then import the CSV file into a spreadsheet to help visualize the latency vs inferences/second tradeoff as well as see some components of the latency. Follow these steps:

- Open this spreadsheet
- Make a copy from the File menu "Make a copy…"
- Open the copy
- Select the A2 cell
- From the File menu select "Import…"
- Select "Upload" and upload the file
- Select "Replace data at selected cell" and then select the "Import data" button

4.4 Client API

The C++ client API exposes a class-based interface for querying server and model status and for performing inference. The commented interface is available at src/clients/c++/request.h and in the API Reference.

The Python client API provides similar capabilities as the C++ API. The commented interface is available at src/clients/python/__init__.py and in the API Reference.

A simple C++ example application at src/clients/c++/simple_client.cc and a Python version at src/clients/python/simple_client.py demonstrate basic client API usage.

To run the the C++ version of the simple example, first build as described in *Building the Client Libraries and Examples* and then:

```
$ /opt/tensorrtserver/bin/simple_client
0 + 1 = 1
0 - 1 = -1
1 + 1 = 2
1 - 1 = 0
2 + 1 = 3
2 - 1 = 1
3 + 1 = 4
3 - 1 = 2
4 + 1 = 5
4 - 1 = 3
5 + 1 = 6
5 - 1 = 4
6 + 1 = 7
6 - 1 = 5
7 + 1 = 8
7 - 1 = 6
8 + 1 = 9
8 - 1 = 7
9 + 1 = 10
9 - 1 = 8
10 + 1 = 11
10 - 1 = 9
11 + 1 = 12
11 - 1 = 10
12 + 1 = 13
12 - 1 = 11
13 + 1 = 14
13 - 1 = 12
14 + 1 = 15
14 - 1 = 13
15 + 1 = 16
15 - 1 = 14
```

To run the the Python version of the simple example, first build as described in *Building the Client Libraries and Examples* and install the tensorrtserver whl, then:

```
$ python3 /workspace/src/clients/python/simple_client.py
```

4.4.1 String Datatype

Some frameworks support tensors where each element in the tensor is a string (see *Datatypes* for information on supported datatypes). For the most part, the Client API is identical for string and non-string tensors. One exception is that in the C++ API a string input tensor must be initialized with SetFromString() instead of SetRaw().

String tensors are demonstrated in the C++ example application at src/clients/c++/simple_string_client.cc and a Python version at src/clients/python/simple_string_client.py.

4.4.2 Stream Inference

Some applications may prefer to send requests in one connection rather than establishing connections for individual requests. For instance, in the case where multiple instances of TensorRT Inference Server are created with the purpose of load balancing, requests sent in different connections may be routed to different server instances. This scenario will not fit the need if the requests are correlated, where they are expected to be processed by the same model instance, like inferencing with sequence models. By using stream inference, the requests will be sent to the same server instance once the connection is established, and then they will be processed by the same model instance if proper `correlation_id` is set.

Stream inference and use of correlation ID are demonstrated in the C++ example application at src/clients/c++/simple_sequence_client.cc and a Python version at src/clients/python/simple_sequence_client.py.

CHAPTER

FIVE

MODEL REPOSITORY

The TensorRT Inference Server accesses models from a locally accessible file path or from Google Cloud Storage. This path is specified when the server is started using the –model-store option.

For a locally accessible file-system the absolute path must be specified, for example, –model-store=/path/to/model/repository. For a model repository residing in Google Cloud Storage, the path must be prefixed with gs://, for example, –model-store=gs://bucket/path/to/model/repository.

Example Model Repository describes how to create an example repository with a couple if image classification models.

An example of a typical model repository layout is shown below:

```
<model-repository-path>/
  model_0/
    config.pbtxt
    output0_labels.txt
    1/
      model.plan
    2/
      model.plan
  model_1/
    config.pbtxt
    output0_labels.txt
    output1_labels.txt
    0/
      model.graphdef
    7/
      model.graphdef
```

Any number of models may be specified and the inference server will attempt to load all models into the CPU and GPU when the server starts. The *Status API* can be used to determine if any models failed to load successfully. The server's console log will also show the reason for any failures during startup.

The name of the model directory (model_0 and model_1 in the above example) must match the name of the model specified in the *model configuration file*, config.pbtxt. The model name is used in the *client API* and *server API* to identify the model. Each model directory must have at least one numeric subdirectory. Each of these subdirectories holds a version of the model with the version number corresponding to the directory name.

For more information about how the model versions are handled by the server see *Model Versions*. Within each version subdirectory there are one or more model definition files that specify the actual model. The model definition can be either a *framework-specific model file* or a shared library implementing a *custom backend*.

The *_labels.txt files are optional and are used to provide labels for outputs that represent classifications. The label file must be specified in the `label_filename` property of the output it corresponds to in the *model configuration*.

5.1 Modifying the Model Repository

By default, changes to the model repository will be detected and the server will attempt to add, remove, and reload models as necessary based on those changes. Changes to the model repository may not be detected immediately because the server polls the repository periodically. You can control the polling interval with the –repository-poll-secs options. The console log or the *Status API* can be used to determine when model repository changes have taken effect. You can disable the server from responding to repository changes by using the –allow-poll-model-repository=false option.

The TensorRT Inference Server responds to the following changes:

- Versions may be added and removed from models by adding and removing the corresponding version subdirectory. The inference server will allow in-flight requests to complete even if they are using a removed version of the model. New requests for a removed model version will fail. Depending on the model's *version policy*, changes to the available versions may change which model version is served by default.
- Existing models can be removed from the repository by removing the corresponding model directory. The inference server will allow in-flight requests to any version of the removed model to complete. New requests for a removed model will fail.
- New models can be added to the repository by adding a new model directory.
- The *model configuration* (config.pbtxt) can be changed and the server will unload and reload the model to pick up the new model configuration.
- Labels files providing labels for outputs that represent classifications can be added, removed, or modified and the inference server will unload and reload the model to pick up the new labels. If a label file is added or removed the corresponding edit to the `label_filename` property of the output it corresponds to in the *model configuration* must be performed at the same time.

5.2 Model Versions

Each model can have one or more versions available in the model repository. Each version is stored in its own, numerically named, subdirectory where the name of the subdirectory corresponds to the version number of the model. Each model specifies a *version policy* that controls which of the versions in the model repository are made available by the server at any given time.

5.3 Framework Model Definition

Each model version subdirectory must contain at least one model definition. By default, the name of this file or directory must be:

- **model.plan** for TensorRT models
- **model.graphdef** for TensorFlow GraphDef models
- **model.savedmodel** for TensorFlow SavedModel models
- **model.netdef** and **init_model.netdef** for Caffe2 Netdef models

This default name can be overridden using the *default_model_filename* property in the *model configuration*.

Optionally, a model can provide multiple model definition files, each targeted at a GPU with a different Compute Capability. Most commonly, this feature is needed for TensorRT and TensorFlow/TensorRT integrated models where the model definition is valid for only a single compute capability. See the *cc_model_filenames* property in the *model configuration* for description of how to specify different model definitions for different compute capabilities.

5.3.1 TensorRT Models

A TensorRT model definition is called a *Plan*. A TensorRT Plan is a single file that by default must be named model.plan. A TensorRT Plan is specific to CUDA Compute Capability and so it is typically necessary to use the *model configuration's cc_model_filenames* property as described above.

A minimal model repository for a single TensorRT model would look like:

```
models/
  <model-name>/
    config.pbtxt
    1/
      model.plan
```

As described in *Generated Model Configuration* the config.pbtxt is optional for some models. In cases where it is not required the minimal model repository would look like:

```
models/
  <model-name>/
    1/
      model.plan
```

5.3.2 TensorFlow Models

TensorFlow saves trained models in one of two ways: *GraphDef* or *SavedModel*. The inference server supports both formats. Once you have a trained model in TensorFlow, you can save it as a GraphDef directly or convert it to a GraphDef by using a script like freeze_graph.py, or save it as a SavedModel using a SavedModelBuilder or tf.saved_model.simple_save.

A TensorFlow GraphDef is a single file that by default must be named model.graphdef. A minimal model repository for a single TensorFlow GraphDef model would look like:

```
models/
  <model-name>/
    config.pbtxt
    1/
      model.graphdef
```

A TensorFlow SavedModel is a directory containing multiple files. By default the directory must be named model.savedmodel. A minimal model repository for a single TensorFlow SavedModel model would look like:

```
models/
  <model-name>/
    config.pbtxt
    1/
      model.savedmodel/
         <saved-model files>
```

As described in *Generated Model Configuration* the config.pbtxt is optional for some models. In cases where it is not required the minimal model repository would look like:

```
models/
  <model-name>/
    1/
      model.savedmodel/
         <saved-model files>
```

5.3.3 Caffe2 Models

A Caffe2 model definition is called a *NetDef*. A Caffe2 NetDef is a single file that by default must be named model.netdef. A minimal model repository for a single NetDef model would look like:

```
models/
  <model-name>/
    config.pbtxt
    1/
      model.netdef
```

5.3.4 TensorRT/TensorFlow Models

TensorFlow 1.7 and later integrates TensorRT to enable TensorFlow models to benefit from the inference optimizations provided by TensorRT. The inference server supports models that have been optimized with TensorRT and can serve those models just like any other TensorFlow model. The inference server's TensorRT version (available in the Release Notes) must match the TensorRT version that was used when the model was created.

A TensorRT/TensorFlow integrated model is specific to CUDA Compute Capability and so it is typically necessary to use the *model configuration's cc_model_filenames* property as described above.

5.3.5 ONNX Models

The TensorRT Inference Server cannot directly perform inferencing using ONNX models. An ONNX model must be converted to either a TensorRT Plan or a Caffe2 NetDef. To convert your ONNX model to a TensorRT Plan use either the ONNX Parser included in TensorRT or the open-source TensorRT backend for ONNX. Another option is to convert your ONNX model to Caffe2 NetDef as described here.

5.4 Custom Backends

A model using a custom backend is represented in the model repository in the same way as models using a deep-learning framework backend. Each model version subdirectory must contain at least one shared library that implements the custom model backend. By default, the name of this shared library must be **libcustom.so** but the default name can be overridden using the *default_model_filename* property in the *model configuration*.

Optionally, a model can provide multiple shared libraries, each targeted at a GPU with a different Compute Capability. See the *cc_model_filenames* property in the *model configuration* for description of how to specify different shared libraries for different compute capabilities.

5.4.1 Custom Backend API

A custom backend must implement the C interface defined in custom.h. The interface is also documented in the API Reference.

5.4.2 Example Custom Backend

An example of a custom backend can be found in the addsub backend. You can see the custom backend being used as part of CI testing in L0_infer.

CHAPTER

SIX

MODEL CONFIGURATION

Each model in a *Model Repository* must include a model configuration that provides required and optional information about the model. Typically, this configuration is provided in a config.pbtxt file specified as ModelConfig protobuf. In some cases, discussed in *Generated Model Configuration*, the model configuration can be generated automatically by the inference server and so does not need to be provided explicitly.

A minimal model configuration must specify `name`, `platform`, `max_batch_size`, `input`, and `output`.

As a running example consider a TensorRT model called *mymodel* that has two inputs, *input0* and *input1*, and one output, *output0*, all of which are 16 entry float32 tensors. The minimal configuration is:

```
name: "mymodel"
platform: "tensorrt_plan"
max_batch_size: 8
input [
  {
    name: "input0"
    data_type: TYPE_FP32
    dims: [ 16 ]
  },
  {
    name: "input1"
    data_type: TYPE_FP32
    dims: [ 16 ]
  }
]
output [
  {
    name: "output0"
    data_type: TYPE_FP32
    dims: [ 16 ]
  }
]
```

The name of the model must match the `name` of the model repository directory containing the model. The `platform` must be one of **tensorrt_plan**, **tensorflow_graphdef**, **tensorflow_savedmodel**, **caffe2_netdef**, or **custom**.

The datatypes allowed for input and output tensors varies based on the type of the model. Section *Datatypes* describes the allowed datatypes and how they map to the datatypes of each model type.

For models that support batched inputs the `max_batch_size` value must be >= 1. The TensorRT Inference Server assumes that the batching occurs along a first dimension that is not listed in the inputs or outputs. For the above example, the server expects to receive input tensors with shape **[x, 16]** and produces an output tensor with shape **[x, 16]**, where **x** is the batch size of the request.

For models that do not support batched inputs the `max_batch_size` value must be zero. If the above example specified a `max_batch_size` of zero, the inference server would expect to receive input tensors with shape **[16]**, and would produce an output tensor with shape **[16]**.

For models that support input and output tensors with variable-size dimensions, those dimensions can be listed as -1 in the input and output configuration. For example, if a model requires a 2-dimensional input tensor where the first dimension must be size 4 but the second dimension can be any size, the model configuration for that input would include **dims: [4, -1]**. The inference server would then accept inference requests where that input tensor's second dimension was any value >= 1. The model configuration can be more restrictive than what is allowed by the underlying model. For example, even though the model allows the second dimension to be any size, the model configuration could be specific as **dims: [4, 4]**. In this case, the inference server would only accept inference requests where the input tensor's shape was exactly **[4, 4]**.

6.1 Generated Model Configuration

By default, the model configuration file containing the required settings must be provided with each model. However, if the server is started with the –strict-model-config=false option, then in some cases the required portions of the model configuration file can be generated automatically by the inference server. The required portion of the model configuration are those settings shown in the example minimal configuration above. Specifically:

- *TensorRT Plan* models do not require a model configuration file because the inference server can derive all the required settings automatically.
- Some *TensorFlow SavedModel* models do not require a model configuration file. The models must specify all inputs and outputs as fixed-size tensors (with an optional initial batch dimension) for the model configuration to be generated automatically. The easiest way to determine if a particular SavedModel is supported is to try it with the server and check the console log and *Status API* to determine if the model loaded successfully.

When using –strict-model-config=false you can see the model configuration that was generated for a model by using the *Status API*.

The TensorRT Inference Server only generates the required portion of the model configuration file. You must still provide the optional portions of the model configuration if necessary, such as `version_policy`, `optimization`, `dynamic_batching`, `instance_group`, `default_model_filename`, `cc_model_filenames`, and `tags`.

6.2 Datatypes

The following table shows the tensor datatypes supported by the TensorRT Inference Server. The first column shows the name of the datatype as it appears in the model configuration file. The other columns show the corresponding datatype for the model frameworks supported by the server and for the Python numpy library. If a model framework does not have an entry for a given datatype, then the inference server does not support that datatype for that model.

Type	TensorRT	TensorFlow	Caffe2	NumPy
TYPE_BOOL		DT_BOOL	BOOL	bool
TYPE_UINT8		DT_UINT8	UINT8	uint8
TYPE_UINT16		DT_UINT16	UINT16	uint16
TYPE_UINT32		DT_UINT32		uint32
TYPE_UINT64		DT_UINT64		uint64
TYPE_INT8	kINT8	DT_INT8	INT8	int8
TYPE_INT16		DT_INT16	INT16	int16
TYPE_INT32	kINT32	DT_INT32	INT32	int32
TYPE_INT64		DT_INT64	INT64	int64
TYPE_FP16	kHALF	DT_HALF	FLOAT16	float16
TYPE_FP32	kFLOAT	DT_FLOAT	FLOAT	float32
TYPE_FP64		DT_DOUBLE	DOUBLE	float64
TYPE_STRING		DT_STRING		dtype(object)

For TensorRT each value is in the nvinfer1::DataType namespace. For example, nvinfer1::DataType::kFLOAT is the 32-bit floating-point datatype.

For TensorFlow each value is in the tensorflow namespace. For example, tensorflow::DT_FLOAT is the 32-bit floating-point value.

For Caffe2 each value is in the caffe2 namespace and is prepended with **TensorProto_DataType_**. For example, caffe2::TensorProto_DataType_FLOAT is the 32-bit floating-point datatype.

For Numpy each value is in the numpy module. For example, numpy.float32 is the 32-bit floating-point datatype.

6.3 Version Policy

Each model can have one or more *versions available in the model repository*. The `nvidia::inferenceserver::ModelVersionPolicy` schema allows the following policies.

- `All`: All versions of the model that are available in the model repository are available for inferencing.
- `Latest`: Only the latest 'n' versions of the model in the repository are available for inferencing. The latest versions of the model are the numerically greatest version numbers.
- `Specific`: Only the specifically listed versions of the model are available for inferencing.

If no version policy is specified, then `Latest` (with num_version = 1) is used as the default, indicating that only the most recent version of the model is made available by the inference server. In all cases, the addition or removal of version subdirectories from the model repository can change which model version is used on subsequent inference requests.

Continuing the above example, the following configuration specifies that all versions of the model will be available from the server:

```
name: "mymodel"
platform: "tensorrt_plan"
max_batch_size: 8
input [
  {
    name: "input0"
    data_type: TYPE_FP32
    dims: [ 16 ]
  },
```

(continues on next page)

(continued from previous page)

```
  {
    name: "input1"
    data_type: TYPE_FP32
    dims: [ 16 ]
  }
]
output [
  {
    name: "output0"
    data_type: TYPE_FP32
    dims: [ 16 ]
  }
]
version_policy: { all { }}
```

6.4 Instance Groups

The TensorRT Inference Server can provide multiple *execution instances* of a model so that multiple simultaneous inference requests for that model can be handled simultaneously. The model configuration `ModelInstanceGroup` is used to specify the number of execution instances that should be made available and what compute resource should be used for those instances.

By default, a single execution instance of the model is created for each GPU available in the system. The instance-group setting can be used to place multiple execution instances of a model on every GPU or on only certain GPUs. For example, the following configuration will place two execution instances of the model to be available on each system GPU:

```
instance_group [
  {
    count: 2
    kind: KIND_GPU
  }
]
```

And the following configuration will place one execution instance on GPU 0 and two execution instances on GPUs 1 and 2:

```
instance_group [
  {
    count: 1
    kind: KIND_GPU
    gpus: [ 0 ]
  },
  {
    count: 2
    kind: KIND_GPU
    gpus: [ 1, 2 ]
  }
]
```

The instance group setting is also used to enable exection of a model on the CPU. The following places two execution instances on the CPU:

```
instance_group [
  {
    count: 2
    kind: KIND_CPU
  }
]
```

6.5 Dynamic Batching

The TensorRT Inference Server supports batch inferencing by allowing individual inference requests to specify a batch of inputs. The inferencing for a batch of inputs is processed at the same time which is especially important for GPUs since it can greatly increase inferencing throughput. In many use-cases the individual inference requests are not batched, therefore, they do not benefit from the throughput benefits of batching.

Dynamic batching is a feature of the inference server that allows non-batched inference requests to be combined by the server, so that a batch is created dynamically, resulting in the same increased throughput seen for batched inference requests.

Dynamic batching is enabled and configured independently for each model using the `ModelDynamicBatching` settings in the model configuration. These settings control the preferred size(s) of the dynamically created batches as well as a maximum time that requests can be delayed in the scheduler to allow other requests to join the dynamic batch.

The following configuration enables dynamic batching with preferred batch sizes of 4 and 8, and a maximum delay time of 100 microseconds:

```
dynamic_batching {
  preferred_batch_size: [ 4, 8 ]
  max_queue_delay_microseconds: 100
}
```

6.6 Optimization Policy

The model configuration `ModelOptimizationPolicy` is used to specify optimization and prioritization settings for a model. These settings control if/how a model is optimized by the backend framework and how it is scheduled and executed by the inference server. See the protobuf documentation for the currently available settings.

CHAPTER

SEVEN

INFERENCE SERVER API

The TensorRT Inference Server exposes both HTTP and GRPC endpoints. Three endpoints with identical functionality are exposed for each protocol.

- *Health*: The server health API for determining server liveness and readiness.
- *Status*: The server status API for getting information about the server and about the models being served.
- *Inference*: The inference API that accepts model inputs, runs inference and returns the requested outputs.

The inference server also exposes an endpoint based on GRPC streams that is only available when using the GRPC protocol:

- *Stream Inference*: The stream inference API is the same as the Inference API, except that once the connection is established, the requests are sent in the same connection until it is closed.

The HTTP endpoints can be used directly as described in this section, but for most use-cases, the preferred way to access the inference server is via the *C++ and Python Client libraries*.

The GRPC endpoints can also be used via the *C++ and Python Client libraries* or a GRPC-generated API can be used directly as shown in the grpc_image_client.py example.

7.1 Health

Performing an HTTP GET to /api/health/live returns a 200 status if the server is able to receive and process requests. Any other status code indicates that the server is still initializing or has failed in some way that prevents it from processing requests.

Once the liveness endpoint indicates that the server is active, performing an HTTP GET to /api/health/ready returns a 200 status if the server is able to respond to inference requests for some or all models (based on the inference server's –strict-readiness option explained below). Any other status code indicates that the server is not ready to respond to some or all inference requests.

For GRPC the `GRPCService` uses the `HealthRequest` and `HealthResponse` messages to implement the endpoint.

By default, the readiness endpoint will return success if the server is responsive and all models loaded successfully. Thus, by default, success indicates that an inference request for any model can be handled by the server. For some use cases, you want the readiness endpoint to return success even if all models are not available. In this case, use the –strict-readiness=false option to cause the readiness endpoint to report success as long as the server is responsive (even if one or more models are not available).

7.2 Status

Performing an HTTP GET to /api/status returns status information about the server and all the models being served. Performing an HTTP GET to /api/status/<model name> returns information about the server and the single model specified by <model name>. The server status is returned in the HTTP response body in either text format (the default) or in binary format if query parameter format=binary is specified (for example, /api/status?format=binary). The success or failure of the status request is indicated in the HTTP response code and the **NV-Status** response header. The **NV-Status** response header returns a text protobuf formatted `RequestStatus` message.

For GRPC the `GRPCService` uses the `StatusRequest` and `StatusResponse` messages to implement the endpoint. The response includes a `RequestStatus` message indicating success or failure.

For either protocol the status itself is returned as a `ServerStatus` message.

7.3 Inference

Performing an HTTP POST to /api/infer/<model name> performs inference using the latest version of the model that is being made available by the model's *version policy*. The latest version is the numerically greatest version number. Performing an HTTP POST to /api/infer/<model name>/<model version> performs inference using a specific version of the model.

The request uses the **NV-InferRequest** header to communicate an `InferRequestHeader` message that describes the input tensors and the requested output tensors. For example, for a resnet50 model the following **NV-InferRequest** header indicates that a batch-size 1 request is being made with a single input named "input", and that the result of the tensor named "output" should be returned as the top-3 classification values:

```
NV-InferRequest: batch_size: 1 input { name: "input" } output { name: "output" cls {␣
↪count: 3 } }
```

The input tensor values are communicated in the body of the HTTP POST request as raw binary in the order as the inputs are listed in the request header.

The HTTP response includes an **NV-InferResponse** header that communicates an `InferResponseHeader` message that describes the outputs. For example the above response could return the following:

```
NV-InferResponse: model_name: "mymodel" model_version: 1 batch_size: 1 output { name:
↪"output" raw { dims: 4 dims: 4 batch_byte_size: 64 } }
```

This response shows that the output in a tensor with shape [4, 4] and has a size of 64 bytes. The output tensor contents are returned in the body of the HTTP response to the POST request. For outputs where full result tensors were requested, the result values are communicated in the body of the response in the order as the outputs are listed in the **NV-InferResponse** header. After those, an `InferResponseHeader` message is appended to the response body. The `InferResponseHeader` message is returned in either text format (the default) or in binary format if query parameter format=binary is specified (for example, /api/infer/foo?format=binary).

For example, assuming an inference request for a model that has 'n' outputs, the outputs specified in the **NV-InferResponse** header in order are "output[0]", . . . , "output[n-1]" the response body would contain:

```
<raw binary tensor values for output[0] >
...
<raw binary tensor values for output[n-1] >
<text or binary encoded InferResponseHeader proto>
```

The success or failure of the inference request is indicated in the HTTP response code and the **NV-Status** response header. The **NV-Status** response header returns a text protobuf formatted `RequestStatus` message.

For GRPC the `GRPCService` uses the `InferRequest` and `InferResponse` messages to implement the endpoint. The response includes a `RequestStatus` message indicating success or failure, `InferResponseHeader` message giving response meta-data, and the raw output tensors.

7.4 Stream Inference

For GRPC the `GRPCService` uses the `InferRequest` and `InferResponse` messages to implement the endpoint. The response includes a `RequestStatus` message indicating success or failure, `InferResponseHeader` message giving response meta-data, and the raw output tensors.

CHAPTER

EIGHT

METRICS

The TensorRT Inference server provides Prometheus metrics indicating GPU and request statistics. By default, these metrics are available at http://localhost:8002/metrics. The inference server –metrics-port option can be used to select a different port. The following table describes the available metrics.

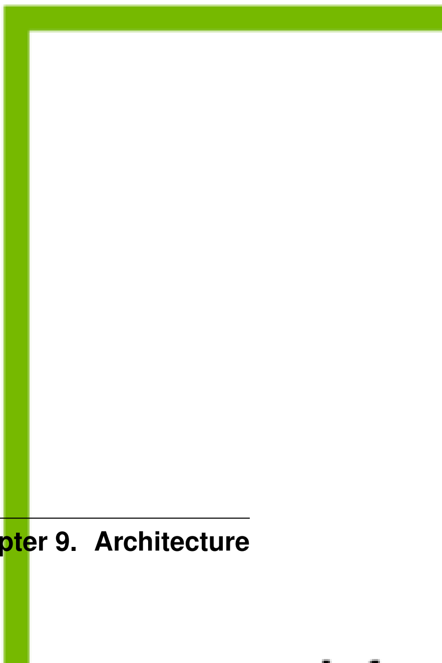
Category	Metric	Description	Granularity	Frequency
GPU Utilization	Power Usage	GPU instantaneous power	Per GPU	Per second
	Power Limit	Maximum GPU power limit	Per GPU	Per second
	Energy Consumption	GPU energy consumption in joules since the server started	Per GPU	Per second
	GPU Utilization	GPU utilization rate (0.0 - 1.0)	Per GPU	Per second
GPU Memory	GPU Total Memory	Total GPU memory, in bytes	Per GPU	Per second
	GPU Used Memory	Used GPU memory, in bytes	Per GPU	Per second
Count	Request Count	Number of inference requests	Per model	Per request
	Execution Count	Number of inference executions (request count / execution count = average dynamic batch size)	Per model	Per request
	Inference Count	Number of inferences performed (one request counts as "batch size" inferences)	Per model	Per request
Latency	Request Time	End-to-end inference request handling time	Per model	Per request
	Compute Time	Time a request	Per model	Per request

CHAPTER

NINE

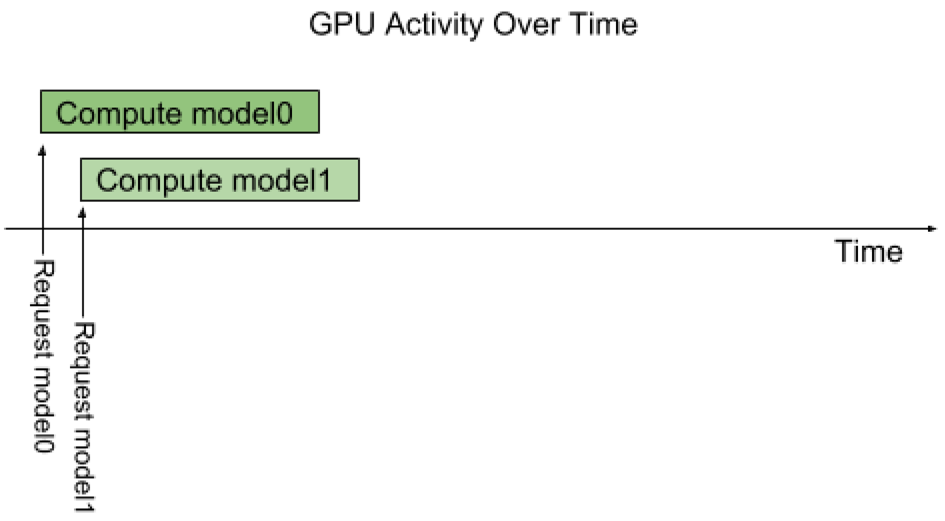
ARCHITECTURE

The following figure shows the TensorRT Inference Server high-level architecture. The *model repository* is a file-system based store of the models that the inference server will make available for inferencing. Inference requests arrive at the server via either *HTTP or GRPC* and are then routed to the appropriate per-model scheduler queue. The scheduler performs fair scheduling and dynamic batching for each model's requests. The schedule passes each request to the framework backend corresponding to the model type. The framework backend performs inferencing using the inputs provided in the request to produce the requested outputs. The outputs are then formatted and a response is sent.

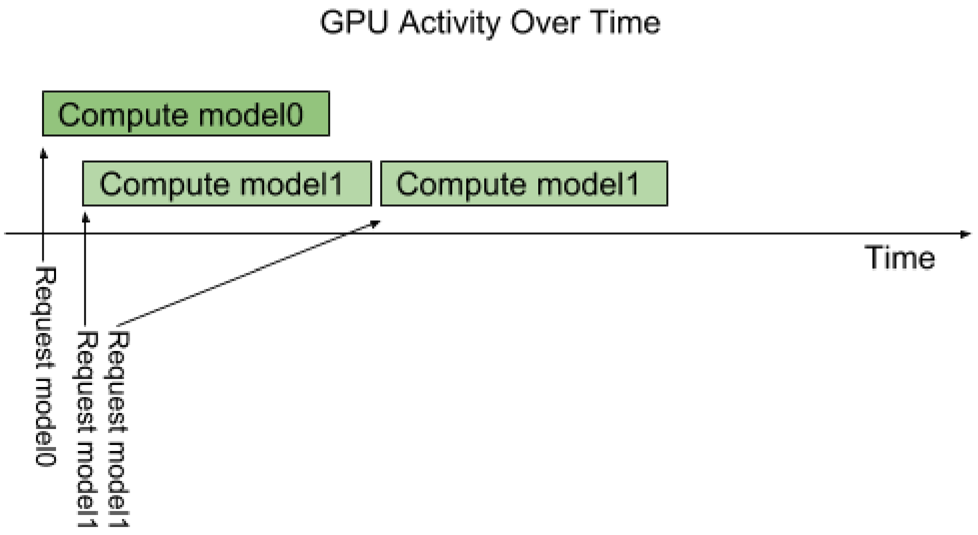


9.1 Concurrent Model Execution

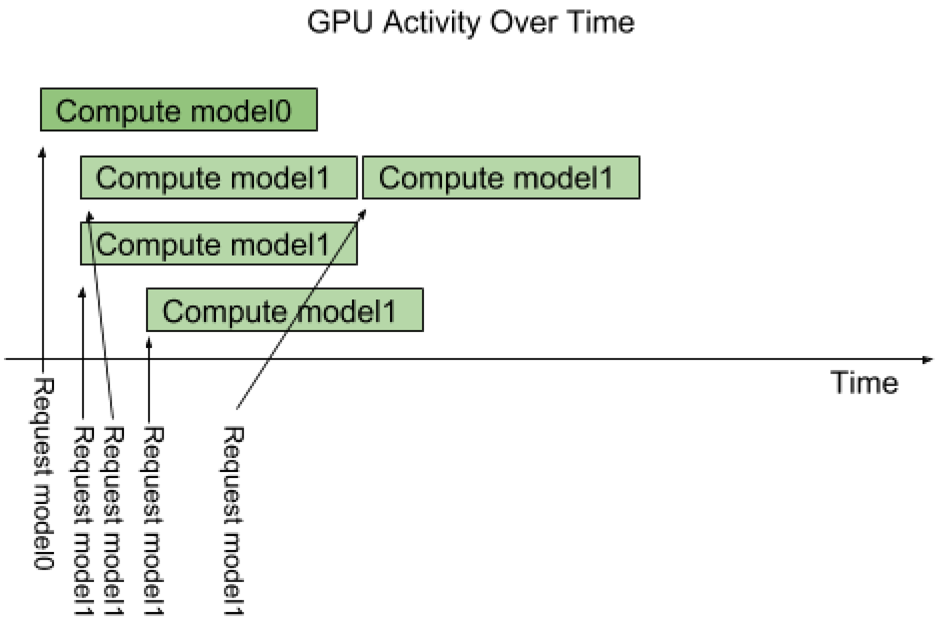
The TensorRT Inference Server architecture allows multiple models and/or multiple instances of the same model to execute in parallel on a single GPU. The following figure shows an example with two models; model0 and model1. Assuming the server is not currently processing any request, when two requests arrive simultaneously, one for each model, the server immediately schedules both of them onto the GPU and the GPU's hardware scheduler begins working on both computations in parallel.



By default, if multiple requests for the same model arrive at the same time, the inference server will serialize their execution by scheduling only one at a time on the GPU, as shown in the following figure.

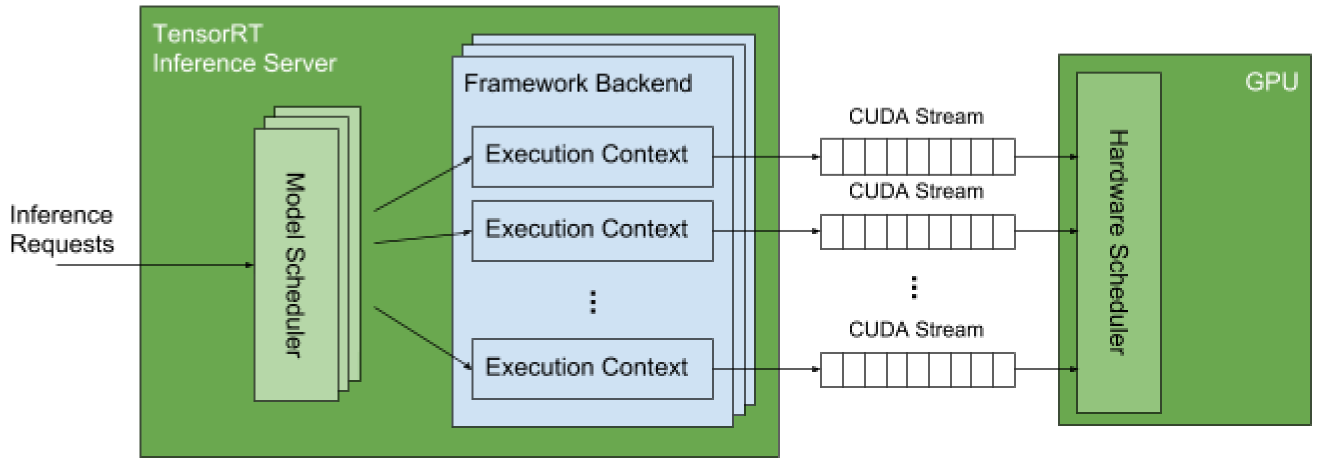


The TensorRT inference server provides an *instance-group* feature that allows each model to specify how many parallel executions of that model should be allowed. Each such enabled parallel execution is referred to as an *execution instance*. By default, the server gives each model a single execution instance, which means that only a single execution of the model is allowed to be in progress at a time as shown in the above figure. By using instance-group the number of execution instances for a model can be increased. The following figure shows model execution when model1 is configured to allow three execution instances. As shown in the figure, the first three model1 inference requests are immediately executed in parallel on the GPU. The fourth model1 inference request must wait until one of the first three executions completes before beginning.



To provide the current model execution capabilities shown in the above figures, the inference server uses CUDA streams to exploit the GPU's hardware scheduling capabilities. CUDA streams allow the server to communicate independent sequences of memory-copy and kernel executions to the GPU. The hardware scheduler in the GPU takes advantage of the independent execution streams to fill the GPU with independent memory-copy and kernel executions. For example, using streams allows the GPU to execute a memory-copy for one model, a kernel for another model, and a different kernel for yet another model at the same time.

The following figure shows some details of how this works within the TensorRT Inference Server. Each framework backend (TensorRT, TensorFlow, Caffe2) provides an API for creating an execution context that is used to execute a given model (each framework uses different terminology for this concept but here we refer to them generally as execution contexts). Each framework allows an execution context to be associated with a CUDA stream. This CUDA stream is used by the framework to execute all memory copies and kernels needed for the model associated with the execution context. For a given model, the inference server creates one execution context for each execution instance specified for the model. When an inference request arrives for a given model, that request is queued in the model scheduler associated with that model. The model scheduler waits for any execution context associated with that model to be idle and then sends the queued request to the context. The execution context then issues all the memory copies and kernel executions required to execute the model to the CUDA stream associated with that execution context. The memory copies and kernels in each CUDA stream are independent of memory copies and kernels in other CUDA streams. The GPU hardware scheduler looks across all CUDA streams to find independent memory copies and kernels to execute on the GPU.


TensorRT
Inference Server
Inference Requests
Model Scheduler
Framework Backend
Execution Context
Execution Context
Execution Context
CUDA Stream
CUDA Stream
CUDA Stream
GPU
Hardware Scheduler

CHAPTER

TEN

CONTRIBUTING

Contributions to TensorRT Inference Server are more than welcome. To contribute make a pull request and follow the guidelines outlined in the CONTRIBUTING document.

10.1 Coding Convention

Use clang-format to format all source files (*.h, *.cc, *.proto) to a consistent format. You should run clang-format on all source files before submitting a pull request:

```
$ apt-get install clang-format clang-format-6.0
$ clang-format-6.0 --style=file -i *.proto *.cc *.h
```

For convenience there is a format.py script in tools/ that can be used to clang-format all files within the repo:

```
$ cd .../tensorrt-inference-server    # top-level of repo
$ python format.py *
```

CHAPTER

ELEVEN

BUILDING

The TensorRT Inference Server is built using Docker and the TensorFlow and PyTorch containers from NVIDIA GPU Cloud (NGC). Before building you must install Docker and nvidia-docker and login to the NGC registry by following the instructions in *Installing Prebuilt Containers*.

11.1 Building the Server

To build a release version of the TensorRT Inference Server container, change directory to the root of the repo and issue the following command:

```
$ docker build --pull -t tensorrtserver .
```

11.1.1 Incremental Builds

For typical development you will want to run the *build* container with your local repo's source files mounted so that your local changes can be incrementally built. This is done by first building the *tensorrtserver_build* container:

```
$ docker build --pull -t tensorrtserver_build --target trtserver_build .
```

By mounting /path/to/tensorrtserver/src into the container at /workspace/src, changes to your local repo will be reflected in the container:

```
$ nvidia-docker run -it --rm -v/path/to/tensorrtserver/src:/workspace/src␣
↪tensorrtserver_build
```

Within the container you can perform an incremental server build with:

```
# cd /workspace
# bazel build -c opt --config=cuda src/servers/trtserver
# cp /workspace/bazel-bin/src/servers/trtserver /opt/tensorrtserver/bin/trtserver
```

Similarly, within the container you can perform an incremental build of the C++ and Python client libraries and example executables with:

```
# cd /workspace
# bazel build -c opt --config=cuda src/clients/...
# mkdir -p /opt/tensorrtserver/bin
# cp bazel-bin/src/clients/c++/image_client /opt/tensorrtserver/bin/.
# cp bazel-bin/src/clients/c++/perf_client /opt/tensorrtserver/bin/.
# cp bazel-bin/src/clients/c++/simple_client /opt/tensorrtserver/bin/.
```

(continues on next page)

(continued from previous page)

```
# mkdir -p /opt/tensorrtserver/lib
# cp bazel-bin/src/clients/c++/librequest.so /opt/tensorrtserver/lib/.
# cp bazel-bin/src/clients/c++/librequest.a /opt/tensorrtserver/lib/.
# mkdir -p /opt/tensorrtserver/pip
# bazel-bin/src/clients/python/build_pip /opt/tensorrtserver/pip/.
```

Some source changes seem to cause bazel to get confused and not correctly rebuild all required sources. You can force bazel to rebuild all of the inference server source without requiring a complete rebuild of the TensorFlow and Caffe2 components by doing the following before issuing the above build command:

```
# rm -fr bazel-bin/src
```

11.2 Building the Client Libraries and Examples

The provided Dockerfile can be used to build just the client libraries and examples. Issue the following command to build the C++ client library, C++ and Python examples, and a Python wheel file for the Python client library:

```
$ docker build -t tensorrtserver_clients --target trtserver_build --build-arg "BUILD_
↪CLIENTS_ONLY=1" .
```

You can optionally add *–build-arg "PYVER=<ver>"* to set the Python version that you want the Python client library built for. Supported values for *<ver>* are 2.6 and 3.5, with 3.5 being the default.

After the build completes the tensorrtserver_clients docker image will contain the built client libraries and examples. The easiest way to try the examples described in the following sections is to run the client image with –net=host so that the client examples can access the inference server running in its own container (see *Running The Inference Server* for more information about running the inference server):

```
$ docker run -it --rm --net=host tensorrtserver_clients
```

In the client image you can find the example executables in /opt/tensorrtserver/bin, and the Python wheel in /opt/tensorrtserver/pip.

If your host sytem is Ubuntu-16.04, an alternative to running the examples within the tensorrtserver_clients container is to instead copy the libraries and examples from the docker image to the host system:

```
$ docker run -it --rm -v/tmp:/tmp/host tensorrtserver_clients
# cp /opt/tensorrtserver/bin/image_client /tmp/host/.
# cp /opt/tensorrtserver/bin/perf_client /tmp/host/.
# cp /opt/tensorrtserver/bin/simple_client /tmp/host/.
# cp /opt/tensorrtserver/pip/tensorrtserver-*.whl /tmp/host/.
# cp /opt/tensorrtserver/lib/librequest.* /tmp/host/.
```

You can now access the files from /tmp on the host system. To run the C++ examples you must install some dependencies on your host system:

```
$ apt-get install curl libcurl3-dev libopencv-dev libopencv-core-dev
```

To run the Python examples you will need to additionally install the client whl file and some other dependencies:

```
$ apt-get install python3 python3-pip
$ pip3 install --user --upgrade tensorrtserver-*.whl numpy pillow
```

11.3 Building the Documentation

The inference server documentation is found in the docs/ directory and is based on Sphinx. Doxygen integrated with Exhale is used for C++ API docuementation.

To build the docs install the required dependencies:

```
$ apt-get update
$ apt-get install -y --no-install-recommends doxygen
$ pip install --upgrade sphinx sphinx-rtd-theme nbsphinx exhale
```

To get the Python client library API docs the TensorRT Inference Server Python package must be installed:

```
$ pip install --upgrade tensorrtserver-*.whl
```

Then use Sphinx to build the documentation into the build/html directory:

```
$ cd docs
$ make clean html
```

To build the PDF version of documentation *LaTEX <https://www.tug.org/texlive/quickinstall.html>* needs to be installed first. Additional requirements of python modules can be met by simply running the following commands in docs directory:

```
$ pip install -r requirements.txt
```

Once latex and python modules have been installed and updated single PDF for documentation can be generated by running the following command. It will generate *NVIDIATRTIS.pdf* in *build/latex* directory:

```
$make clean latexpdf
```

Corrections and enhancements in Documentation are always welcome however it is advised that before creating a pull request for the changes these are validated locally. A simple way to do it is to run web server inside *docs/build/html* directory with following command and navigate through the modified documentation in the browser at http://localhost:8000

```
$ python -m SimpleHTTPServer          # run inside docs/build/html
```

CHAPTER TWELVE

TESTING

Currently there is no CI testing enabled for the open-source version of the TensorRT Inference Server. We will enable CI testing in a future update.

There is a set of tests in the qa/ directory that can be run manually to provide some testing. Before running these tests you must first generate a couple of test model repositories containing the models needed by the tests.

12.1 Generate QA Model Repositories

The QA model repositories contain some simple models that are used to verify the correctness of the inference server. To generate the QA model repositories:

```
$ cd qa/common
$ ./gen_qa_model_repository
```

This will generate two model repositories: in /tmp/qa_model_repository and /tmp/qa_variable_model_repository. The TensorRT models will be created for the GPU on the system that CUDA considers device 0 (zero). If you have multiple GPUs on your system see the documentation in the script for how to target a specific GPU.

12.2 Build QA Container

Next you need to build a QA version of the inference server container. This container will contain the inference server, the QA tests, and all the dependencies needed to run the QA tests. You must first build the tensorrtserver_build and tensorrtserver containers as described in *Building the Server* and then build the QA container:

```
$ docker build -t tensorrtserver_qa -f Dockerfile.QA .
```

12.3 Run QA Container

Now run the QA container and mount the QA model repository into the container so the tests will be able to access it:

```
$ nvidia-docker run -it --rm -v/tmp:/data/inferenceserver tensorrtserver_qa
```

Within the container the QA tests are in /opt/tensorrtserver/qa. To run a test:

```
$ cd <test directory>
$ ./test.sh
```

CHAPTER

THIRTEEN

PROTOBUF API

13.1 HTTP/GRPC API

- src/core/api.proto
- src/core/grpc_service.proto
- src/core/request_status.proto

13.2 Model Configuration

- src/core/model_config.proto

13.3 Status

- src/core/server_status.proto

CHAPTER

FOURTEEN

C++ API

14.1 Class Hierarchy

14.2 File Hierarchy

14.3 Full API

14.3.1 Namespaces

Namespace nvidia

Contents

Namespaces

- *Namespace nvidia::inferenceserver*

Namespace nvidia::inferenceserver

Contents

Namespaces

- *Namespace nvidia::inferenceserver::client*

Namespace nvidia::inferenceserver::client

Contents

Classes

- *Struct Result::ClassResult*
- *Struct InferContext::Stat*
- *Class Error*
- *Class InferContext*
- *Class InferContext::Input*
- *Class InferContext::Options*
- *Class InferContext::Output*
- *Class InferContext::Request*
- *Class InferContext::RequestTimers*
- *Class InferContext::Result*
- *Class InferGrpcContext*
- *Class InferGrpcStreamContext*
- *Class InferHttpContext*
- *Class ProfileContext*
- *Class ProfileGrpcContext*
- *Class ProfileHttpContext*
- *Class ServerHealthContext*
- *Class ServerHealthGrpcContext*
- *Class ServerHealthHttpContext*
- *Class ServerStatusContext*
- *Class ServerStatusGrpcContext*
- *Class ServerStatusHttpContext*

Functions

- *Function nvidia::inferenceserver::client::operator<<*

14.3.2 Classes and Structs

Struct custom_payload_struct

- Defined in *File custom.h*

Struct Documentation

`struct custom_payload_struct`

Public Members

uint32_t **`batch_size`**

uint32_t **`input_cnt`**

`const` char ***`*input_names`**

`const` size_t ***`input_shape_dim_cnts`**

`const` int64_t ****`input_shape_dims`**

uint32_t **`output_cnt`**

`const` char ****`required_output_names`**

void ***`input_context`**

void ***`output_context`**

int **`error_code`**

Struct Result::ClassResult

- Defined in *File request.h*

Nested Relationships

This struct is a nested type of *Class InferContext::Result*.

Struct Documentation

`struct ClassResult`

The result value for CLASS format results.

Public Members

size_t **`idx`**

The index of the class in the result vector.

float **`value`**

The value of the class.

std::string **label**
The label for the class, if provided by the model.

Struct InferContext::Stat

- Defined in *File request.h*

Nested Relationships

This struct is a nested type of *Class InferContext*.

Struct Documentation

struct Stat
Cumulative statistic of the *InferContext*.

Note For GRPC protocol, 'cumulative_send_time_ns' represents the time for marshaling infer request. 'cumulative_receive_time_ns' represents the time for unmarshaling infer response.

Public Functions

Stat()
Create a new *Stat* object with zero-ed statistics.

Public Members

size_t **completed_request_count**
Total number of requests completed.

uint64_t **cumulative_total_request_time_ns**
Time from the request start until the response is completely received.

uint64_t **cumulative_send_time_ns**
Time from the request start until the last byte is sent.

uint64_t **cumulative_receive_time_ns**
Time from receiving first byte of the response until the response is completely received.

Class Error

- Defined in *File request.h*

Class Documentation

class Error
Error status reported by client API.

Public Functions

Error(**const** RequestStatus &*status*)
: Create an error from a RequestStatus.

Parameters

- `status`: The RequestStatus object

Error(RequestStatusCode *code* = RequestStatusCode::SUCCESS)
: Create an error from a RequestStatusCode.

Parameters

- `code`: The status code for the error

Error(RequestStatusCode *code*, **const** std::string &*msg*)
: Create an error from a RequestStatusCode and a detailed message.

Parameters

- `code`: The status code for the error
- `msg`: The detailed message for the error

RequestStatusCode **Code**() **const**
: Accessor for the RequestStatusCode of this error.

Return The RequestStatusCode for the error.

const std::string &**Message**() **const**
: Accessor for the message of this error.

Return The detailed messsage for the error. Empty if no detailed message.

const std::string &**ServerId**() **const**
: Accessor for the ID of the inference server associated with this error.

Return The ID of the inference server associated with this error, or empty-string if no inference server is associated with the error.

uint64_t **RequestId**() **const**
: Accessor for the ID of the request associated with this error.

Return The ID of the request associated with this error, or 0 (zero) if no request ID is associated with the error.

bool **IsOk**() **const**
: Does this error indicate OK status?

Return True if this error indicates “ok”/”success”, false if error indicates a failure.

Public Static Attributes

const *Error* **Success**

Convenience "success" value.

Can be used as *Error::Success* to indicate no error.

Class InferContext

- Defined in *File request.h*

Nested Relationships

Nested Types

- *Class InferContext::Input*
- *Class InferContext::Options*
- *Class InferContext::Output*
- *Class InferContext::Request*
- *Class InferContext::RequestTimers*
- *Class InferContext::Result*
- *Struct Result::ClassResult*
- *Struct InferContext::Stat*

Inheritance Relationships

Derived Types

- `public nvidia::inferenceserver::client::InferGrpcContext` (*Class InferGrpcContext*)
- `public nvidia::inferenceserver::client::InferHttpContext` (*Class InferHttpContext*)

Class Documentation

class InferContext

An *InferContext* object is used to run inference on an inference server for a specific model.

Once created an *InferContext* object can be used repeatedly to perform inference using the model. *Options* that control how inference is performed can be changed in between inference runs.

A *InferContext* object can use either HTTP protocol or GRPC protocol depending on the Create function (*InferHttpContext::Create* or *InferGrpcContext::Create*). For example:

```
std::unique_ptr<InferContext> ctx;
InferHttpContext::Create(&ctx, "localhost:8000", "mnist");
...
std::unique_ptr<Options> options0;
Options::Create(&options0);
```

(continues on next page)

(continued from previous page)

```
options->SetBatchSize(b);
options->AddClassResult(output, topk);
ctx->SetRunOptions(*options0);
...
ctx->Run(&results0);  // run using options0
ctx->Run(&results1);  // run using options0
...
std::unique_ptr<Options> options1;
Options::Create(&options1);
options->AddRawResult(output);
ctx->SetRunOptions(*options);
...
ctx->Run(&results2);  // run using options1
ctx->Run(&results3);  // run using options1
...
```

Note InferContext::Create methods are thread-safe. All other *InferContext* methods, and nested class methods are not thread-safe.

The *Run()* calls are not thread-safe but a new *Run()* can be invoked as soon as the previous completes. The returned result objects are owned by the caller and may be retained and accessed even after the *InferContext* object is destroyed.

AsyncRun() and GetAsyncRunStatus() calls are not thread-safe. What's more, calling one method while the other one is running will result in undefined behavior given that they will modify the shared data internally.

For more parallelism multiple *InferContext* objects can access the same inference server with no serialization requirements across those objects.

Subclassed by *nvidia::inferenceserver::client::InferGrpcContext*, *nvidia::inferenceserver::client::InferHttpContext*

Public Types

`using ResultMap` = std::map<std::string, std::unique_ptr<*Result*>>

Public Functions

`virtual ~InferContext``()`
Destroy the inference context.

`const` std::string &**`ModelName`**`()` **`const`**

Return The name of the model being used for this context.

int64_t **`ModelVersion`**`()` **`const`**

Return The version of the model being used for this context. -1 indicates that the latest (i.e. highest version number) version of that model is being used.

uint64_t **`MaxBatchSize`**`()` **`const`**

Return The maximum batch size supported by the context. A maximum batch size indicates that the context does not support batching and so only a single inference at a time can be performed.

`const` std::vector<std::shared_ptr<*Input*>> &**`Inputs`**`()` **`const`**

Return The inputs of the model.

const std::vector<std::shared_ptr<*Output*>> &**Outputs**() **const**

Return The outputs of the model.

Error **GetInput**(**const** std::string &*name*, std::shared_ptr<*Input*> **input*) **const**
Get a named input.

Return *Error* object indicating success or failure.

Parameters

- `name`: The name of the input.
- `input`: Returns the *Input* object for 'name'.

Error **GetOutput**(**const** std::string &*name*, std::shared_ptr<*Output*> **output*) **const**
Get a named output.

Return *Error* object indicating success or failure.

Parameters

- `name`: The name of the output.
- `output`: Returns the *Output* object for 'name'.

Error **SetRunOptions**(**const** *Options* &*options*)
Set the options to use for all subsequent *Run()* invocations.

Return *Error* object indicating success or failure.

Parameters

- `options`: The options.

Error **GetStat**(*Stat* **stat*)
Get the current statistics of the *InferContext*.

Return *Error* object indicating success or failure.

Parameters

- `stat`: Returns the *Stat* object holding the statistics.

virtual *Error* **Run**(*ResultMap* **results*) = 0
Send a synchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `results`: Returns *Result* objects holding inference results as a map from output name to *Result* object.

virtual *Error* **AsyncRun**(std::shared_ptr<*Request*> **async_request*) = 0
Send an asynchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `async_request`: Returns a *Request* object that can be used to retrieve the inference results for the request.

virtual *Error* **GetAsyncRunResults**(*ResultMap *results*, **const** std::shared_ptr<*Request*> *&async_request*, bool *wait*) = 0

Get the results of the asynchronous request referenced by 'async_request'.

Return *Error* object indicating success or failure. Success will be returned only if the request has been completed succesfully. UNAVAILABLE will be returned if 'wait' is false and the request is not ready.

Parameters

- `results`: Returns *Result* objects holding inference results as a map from output name to *Result* object.
- `async_request`: *Request* handle to retrieve results.
- `wait`: If true, block until the request completes. Otherwise, return immediately.

Error **GetReadyAsyncRequest**(std::shared_ptr<*Request*> **async_request*, bool *wait*)

Get any one completed asynchronous request.

Return *Error* object indicating success or failure. Success will be returned only if a completed request was returned.. UNAVAILABLE will be returned if 'wait' is false and no request is ready.

Parameters

- `async_request`: Returns the *Request* object holding the completed request.
- `wait`: If true, block until the request completes. Otherwise, return immediately.

Protected Types

using AsyncReqMap = std::map<uintptr_t, std::shared_ptr<*Request*>>

Protected Functions

InferContext(**const** std::string&, int64_t, CorrelationID, bool)

virtual void **AsyncTransfer**() = 0

virtual *Error* **PreRunProcessing**(std::shared_ptr<*Request*> *&request*) = 0

Error **IsRequestReady**(**const** std::shared_ptr<*Request*> *&async_request*, bool *wait*)

Error **UpdateStat**(**const** *RequestTimers &timer*)

Protected Attributes

AsyncReqMap **ongoing_async_requests_**

const std::string **model_name_**

const int64_t **model_version_**

const CorrelationID **correlation_id_**

const bool **verbose_**

uint64_t **max_batch_size_**

uint64_t **batch_size_**

uint64_t **async_request_id_**

std::vector<std::shared_ptr<*Input*>> **inputs_**

std::vector<std::shared_ptr<*Output*>> **outputs_**

InferRequestHeader **infer_request_**

std::shared_ptr<*Request*> **sync_request_**

Stat **context_stat_**

std::thread **worker_**

std::mutex **mutex_**

std::condition_variable **cv_**

bool **exiting_**

class Input
An input to the model.

Public Functions

virtual ~Input ()
Destroy the input.

virtual const std::string &**Name** () **const** = 0
Return The name of the input.

virtual int64_t **ByteSize** () **const** = 0
Return The size in bytes of this input. This is the size for one instance of the input, not the entire size of a batched input. When the byte-size is not known, for example for non-fixed-sized types like TYPE_STRING or for inputs with variable-size dimensions, this will return -1.

virtual size_t **TotalByteSize** () **const** = 0
Return The size in bytes of entire batch of this input. For fixed-sized types this is just *ByteSize()* * batch-size, but for non-fixed-sized types like TYPE_STRING it is the only way to get the entire input size.

virtual DataType **DType** () **const** = 0
Return The data-type of the input.

virtual ModelInput::Format **Format** () **const** = 0
Return The format of the input.

virtual const DimsList &**Dims** () **const** = 0
Return The dimensions/shape of the input specified in the model configuration. Variable-size dimensions are reported as -1.

virtual *Error* **Reset**() = 0

Prepare this input to receive new tensor values.

Forget any existing values that were set by previous calls to *SetRaw()*.

Return *Error* object indicating success or failure.

virtual const std::vector<int64_t> &**Shape**() **const** = 0

Get the shape for this input that was most recently set by SetShape.

Return The shape, or empty vector if SetShape has not been called.

virtual *Error* **SetShape**(**const** std::vector<int64_t> &*dims*) = 0

Set the shape for this input.

The shape must be set for inputs that have variable-size dimensions and is optional for other inputs. The shape must be set before calling SetRaw or SetFromString.

Return *Error* object indicating success or failure.

Parameters

- `dims`: The dimensions of the shape.

virtual *Error* **SetRaw**(**const** uint8_t **input*, size_t *input_byte_size*) = 0

Set tensor values for this input from a byte array.

The array is not copied and so it must not be modified or destroyed until this input is no longer needed (that is until the *Run()* call(s) that use the input have completed). For batched inputs this function must be called batch-size times to provide all tensor values for a batch of this input.

Return *Error* object indicating success or failure.

Parameters

- `input`: The pointer to the array holding the tensor value.
- `input_byte_size`: The size of the array in bytes, must match the size expected by the input.

virtual *Error* **SetRaw**(**const** std::vector<uint8_t> &*input*) = 0

Set tensor values for this input from a byte vector.

The vector is not copied and so it must not be modified or destroyed until this input is no longer needed (that is until the *Run()* call(s) that use the input have completed). For batched inputs this function must be called batch-size times to provide all tensor values for a batch of this input.

Return *Error* object indicating success or failure.

Parameters

- `input`: The vector holding tensor values.

virtual *Error* **SetFromString**(**const** std::vector<std::string> &*input*) = 0

Set tensor values for this input from a vector or strings.

This method can only be used for tensors with STRING data-type. The strings are assigned in row-major order to the elements of the tensor. The strings are copied and so the ‘input’ does not need to be preserved as with *SetRaw()*. For batched inputs this function must be called batch-size times to provide all tensor values for a batch of this input.

Return *Error* object indicating success or failure.

Parameters

- `input`: The vector holding tensor string values.

class Options

Run options to be applied to all subsequent *Run()* invocations.

Public Functions

virtual ~Options()

virtual bool **Flag**(InferRequestHeader::Flag *flag*) **const** = 0
Get the value of a request flag being used for all subsequent inferences.

Cannot be used with FLAG_NONE.
Return The true/false value currently set for the flag. If 'flag' is FLAG_NONE then return false.
Parameters
- `flag`: The flag to get the value for.

virtual void **SetFlag**(InferRequestHeader::Flag *flag*, bool *value*) = 0
Set a request flag to be used for all subsequent inferences.

Parameters
- `flag`: The flag to set. Cannot be used with FLAG_NONE.
- `value`: The true/false value to set for the flag. If 'flag' is FLAG_NONE then do nothing.

virtual uint32_t **Flags**() **const** = 0
Get the value of all request flags being used for all subsequent inferences.

Return The bitwise-or of flag values as a single uint32_t value.

virtual void **SetFlags**(uint32_t *flags*) = 0
Set all request flags to be used for all subsequent inferences.

Parameters
- `flags`: The bitwise-or of flag values to set.

virtual size_t **BatchSize**() **const** = 0
Return The batch size to use for all subsequent inferences.

virtual void **SetBatchSize**(size_t *batch_size*) = 0
Set the batch size to use for all subsequent inferences.

Parameters
- `batch_size`: The batch size.

virtual *Error* **AddRawResult**(**const** std::shared_ptr<*InferContext*::*Output*> &*output*) = 0
Add 'output' to the list of requested RAW results.

Run() will return the output's full tensor as a result.
Return *Error* object indicating success or failure.
Parameters
- `output`: The output.

virtual *Error* **AddClassResult**(**const** std::shared_ptr<*InferContext*::*Output*> &*output*, uint64_t *k*) = 0
Add 'output' to the list of requested CLASS results.

Run() will return the highest 'k' values of 'output' as a result.
Return *Error* object indicating success or failure.
Parameters
- `output`: The output.
- `k`: Set how many class results to return for the output.

Public Static Functions

static *Error* **Create**(std::unique_ptr<*Options*> **options*)
Create a new *Options* object with default values.

Return *Error* object indicating success or failure.

class Output
An output from the model.

Public Functions

virtual ~Output()
Destroy the output.

virtual const std::string &**Name**() **const** = 0
Return The name of the output.

virtual DataType **DType**() **const** = 0
Return The data-type of the output.

virtual const DimsList &**Dims**() **const** = 0
Return The dimensions/shape of the output specified in the model configuration. Variable-size dimensions are reported as -1.

class Request
Handle to a inference request.

The request handle is used to get request results if the request is sent by *AsyncRun()*.

Public Functions

virtual ~Request()
Destroy the request handle.

virtual uint64_t **Id**() **const** = 0
Return The unique identifier of the request.

class RequestTimers
Timer to record the timestamp for different stages of request handling.

Public Types

enum Kind
The kind of the timer.

Values:

REQUEST_START
The start of request handling.

REQUEST_END
The end of request handling.

SEND_START
The start of sending request bytes to the server (i.e. first byte).

SEND_END
The end of sending request bytes to the server (i.e. last byte).

RECEIVE_START
The start of receiving response bytes from the server (i.e.

first byte).

RECEIVE_END
The end of receiving response bytes from the server (i.e.

last byte).

Public Functions

RequestTimers()
Construct a timer with zero-ed timestamps.

Error **Reset**()
Reset all timestamp values to zero.

Must be called before re-using the timer.
Return *Error* object indicating success or failure.

Error **Record**(*Kind kind*)
Record the current timestamp for a request stage.

Return *Error* object indicating success or failure.
Parameters
- `kind`: The Kind of the timestamp.

class Result
An inference result corresponding to an output.

Public Types

enum ResultFormat
Format in which result is returned.

Values:

RAW = 0
RAW format is the entire result tensor of values.

CLASS = 1
CLASS format is the top-k highest probability values of the result and the associated class label (if provided by the model).

Public Functions

virtual ~Result()
Destroy the result.

virtual const std::string &**ModelName**() **const** = 0
Return The name of the model that produced this result.

virtual int64_t **ModelVersion**() **const** = 0

Return The version of the model that produced this result.

virtual const std::shared_ptr<*Output*> **GetOutput**() **const** = 0

Return The *Output* object corresponding to this result.

virtual *Error* **GetRawShape**(std::vector<int64_t> **shape*) **const** = 0

Get the shape of a raw result.

The shape does not include the batch dimension.

Return *Error* object indicating success or failure.

Parameters

- `shape`: Returns the shape.

virtual *Error* **GetRaw**(size_t *batch_idx*, **const** std::vector<uint8_t> ***buf*) **const** = 0

Get a reference to entire raw result data for a specific batch entry.

Returns error if this result is not RAW format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: Returns the results for this entry of the batch.
- `buf`: Returns the vector of result bytes.

virtual *Error* **GetRawAtCursor**(size_t *batch_idx*, **const** uint8_t ***buf*, size_t *adv_byte_size*) = 0

Get a reference to raw result data for a specific batch entry at the current "cursor" and advance the cursor by the specified number of bytes.

More typically use *GetRawAtCursor<T>()* method to return the data as a specific type T. Use *ResetCursor()* to reset the cursor to the beginning of the result. Returns error if this result is not RAW format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: Returns results for this entry of the batch.
- `buf`: Returns pointer to 'adv_byte_size' bytes of data.
- `adv_byte_size`: The number of bytes of data to get a reference to.

template <typename T>
Error **GetRawAtCursor**(size_t *batch_idx*, T **out*)

Read a value for a specific batch entry at the current "cursor" from the result tensor as the specified type T and advance the cursor.

Use *ResetCursor()* to reset the cursor to the beginning of the result. Returns error if this result is not RAW format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: Returns results for this entry of the batch.
- `out`: Returns the value at the cursor.

virtual *Error* **GetClassCount**(size_t *batch_idx*, size_t **cnt*) **const** = 0

Get the number of class results for a batch.

Returns error if this result is not CLASS format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: The index in the batch.
- `cnt`: Returns the number of *ClassResult* entries for the batch entry.

virtual *Error* **GetClassAtCursor**(size_t *batch_idx*, *ClassResult* **result*) = 0

Get the *ClassResult* result for a specific batch entry at the current cursor.

Use *ResetCursor()* to reset the cursor to the beginning of the result. Returns error if this result is not CLASS format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: The index in the batch.
- `result`: Returns the *ClassResult* value for the batch at the cursor.

virtual *Error* **ResetCursors**() = 0

Reset cursor to beginning of result for all batch entries.

Return *Error* object indicating success or failure.

virtual *Error* **ResetCursor**(size_t *batch_idx*) = 0

Reset cursor to beginning of result for specified batch entry.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: The index in the batch.

template <>

Error **GetRawAtCursor**(size_t *batch_idx*, std::string **out*)

struct ClassResult

The result value for CLASS format results.

Public Members

size_t **idx**

The index of the class in the result vector.

float **value**

The value of the class.

std::string **label**

The label for the class, if provided by the model.

struct Stat

Cumulative statistic of the *InferContext*.

Note For GRPC protocol, 'cumulative_send_time_ns' represents the time for marshaling infer request. 'cumulative_receive_time_ns' represents the time for unmarshaling infer response.

Public Functions

Stat()

Create a new *Stat* object with zero-ed statistics.

Public Members

size_t **completed_request_count**

Total number of requests completed.

uint64_t **cumulative_total_request_time_ns**

Time from the request start until the response is completely received.

uint64_t **cumulative_send_time_ns**
Time from the request start until the last byte is sent.

uint64_t **cumulative_receive_time_ns**
Time from receiving first byte of the response until the response is completely received.

Class InferContext::Input

- Defined in *File request.h*

Nested Relationships

This class is a nested type of *Class InferContext*.

Class Documentation

class Input
An input to the model.

Public Functions

virtual ~Input()
Destroy the input.

virtual const std::string &**Name**() **const** = 0

Return The name of the input.

virtual int64_t **ByteSize**() **const** = 0

Return The size in bytes of this input. This is the size for one instance of the input, not the entire size of a batched input. When the byte-size is not known, for example for non-fixed-sized types like TYPE_STRING or for inputs with variable-size dimensions, this will return -1.

virtual size_t **TotalByteSize**() **const** = 0

Return The size in bytes of entire batch of this input. For fixed-sized types this is just *ByteSize()* * batch-size, but for non-fixed-sized types like TYPE_STRING it is the only way to get the entire input size.

virtual DataType **DType**() **const** = 0

Return The data-type of the input.

virtual ModelInput::Format **Format**() **const** = 0

Return The format of the input.

virtual const DimsList &**Dims**() **const** = 0

Return The dimensions/shape of the input specified in the model configuration. Variable-size dimensions are reported as -1.

virtual *Error* **Reset**() = 0
Prepare this input to receive new tensor values.

Forget any existing values that were set by previous calls to *SetRaw()*.

Return *Error* object indicating success or failure.

virtual const std::vector<int64_t> &**Shape**() **const** = 0

Get the shape for this input that was most recently set by SetShape.

Return The shape, or empty vector if SetShape has not been called.

virtual *Error* **SetShape**(**const** std::vector<int64_t> &*dims*) = 0

Set the shape for this input.

The shape must be set for inputs that have variable-size dimensions and is optional for other inputs. The shape must be set before calling SetRaw or SetFromString.

Return *Error* object indicating success or failure.

Parameters

- `dims`: The dimensions of the shape.

virtual *Error* **SetRaw**(**const** uint8_t **input*, size_t *input_byte_size*) = 0

Set tensor values for this input from a byte array.

The array is not copied and so it must not be modified or destroyed until this input is no longer needed (that is until the *Run()* call(s) that use the input have completed). For batched inputs this function must be called batch-size times to provide all tensor values for a batch of this input.

Return *Error* object indicating success or failure.

Parameters

- `input`: The pointer to the array holding the tensor value.
- `input_byte_size`: The size of the array in bytes, must match the size expected by the input.

virtual *Error* **SetRaw**(**const** std::vector<uint8_t> &*input*) = 0

Set tensor values for this input from a byte vector.

The vector is not copied and so it must not be modified or destroyed until this input is no longer needed (that is until the *Run()* call(s) that use the input have completed). For batched inputs this function must be called batch-size times to provide all tensor values for a batch of this input.

Return *Error* object indicating success or failure.

Parameters

- `input`: The vector holding tensor values.

virtual *Error* **SetFromString**(**const** std::vector<std::string> &*input*) = 0

Set tensor values for this input from a vector or strings.

This method can only be used for tensors with STRING data-type. The strings are assigned in row-major order to the elements of the tensor. The strings are copied and so the 'input' does not need to be preserved as with *SetRaw()*. For batched inputs this function must be called batch-size times to provide all tensor values for a batch of this input.

Return *Error* object indicating success or failure.

Parameters

- `input`: The vector holding tensor string values.

Class InferContext::Options

- Defined in *File request.h*

Nested Relationships

This class is a nested type of *Class InferContext*.

Class Documentation

class Options

Run options to be applied to all subsequent *Run()* invocations.

Public Functions

virtual ~Options()

virtual bool **Flag**(InferRequestHeader::Flag *flag*) **const** = 0

Get the value of a request flag being used for all subsequent inferences.

Cannot be used with FLAG_NONE.

Return The true/false value currently set for the flag. If 'flag' is FLAG_NONE then return false.

Parameters

- `flag`: The flag to get the value for.

virtual void **SetFlag**(InferRequestHeader::Flag *flag*, bool *value*) = 0

Set a request flag to be used for all subsequent inferences.

Parameters

- `flag`: The flag to set. Cannot be used with FLAG_NONE.
- `value`: The true/false value to set for the flag. If 'flag' is FLAG_NONE then do nothing.

virtual uint32_t **Flags**() **const** = 0

Get the value of all request flags being used for all subsequent inferences.

Return The bitwise-or of flag values as a single uint32_t value.

virtual void **SetFlags**(uint32_t *flags*) = 0

Set all request flags to be used for all subsequent inferences.

Parameters

- `flags`: The bitwise-or of flag values to set.

virtual size_t **BatchSize**() **const** = 0

Return The batch size to use for all subsequent inferences.

virtual void **SetBatchSize**(size_t *batch_size*) = 0

Set the batch size to use for all subsequent inferences.

Parameters

- `batch_size`: The batch size.

virtual *Error* **AddRawResult**(**const** std::shared_ptr<*InferContext*::*Output*> &*output*) = 0

Add 'output' to the list of requested RAW results.

Run() will return the output's full tensor as a result.

Return *Error* object indicating success or failure.

Parameters

- `output`: The output.

virtual *Error* **AddClassResult**(**const** std::shared_ptr<*InferContext*::*Output*> &*output*, uint64_t *k*) = 0

Add 'output' to the list of requested CLASS results.

Run() will return the highest 'k' values of 'output' as a result.

Return *Error* object indicating success or failure.

Parameters

- `output`: The output.
- `k`: Set how many class results to return for the output.

Public Static Functions

static *Error* **Create**(std::unique_ptr<*Options*> **options*)

Create a new *Options* object with default values.

Return *Error* object indicating success or failure.

Class InferContext::Output

- Defined in *File request.h*

Nested Relationships

This class is a nested type of *Class InferContext*.

Class Documentation

class Output

An output from the model.

Public Functions

virtual ~Output()

Destroy the output.

virtual const std::string &**Name**() **const** = 0

Return The name of the output.

`virtual` DataType **`DType`**`()` **`const`** = 0

Return The data-type of the output.

`virtual` **`const`** DimsList &**`Dims`**`()` **`const`** = 0

Return The dimensions/shape of the output specified in the model configuration. Variable-size dimensions are reported as -1.

Class InferContext::Request

- Defined in *File request.h*

Nested Relationships

This class is a nested type of *Class InferContext*.

Class Documentation

`class Request`

Handle to a inference request.

The request handle is used to get request results if the request is sent by *AsyncRun()*.

Public Functions

`virtual` **`~Request`**`()`

Destroy the request handle.

`virtual` uint64_t **`Id`**`()` **`const`** = 0

Return The unique identifier of the request.

Class InferContext::RequestTimers

- Defined in *File request.h*

Nested Relationships

This class is a nested type of *Class InferContext*.

Class Documentation

`class RequestTimers`

Timer to record the timestamp for different stages of request handling.

Public Types

`enum Kind`
The kind of the timer.

Values:

`REQUEST_START`
The start of request handling.

`REQUEST_END`
The end of request handling.

`SEND_START`
The start of sending request bytes to the server (i.e. first byte).

`SEND_END`
The end of sending request bytes to the server (i.e. last byte).

`RECEIVE_START`
The start of receiving response bytes from the server (i.e.

first byte).

`RECEIVE_END`
The end of receiving response bytes from the server (i.e.

last byte).

Public Functions

`RequestTimers``()`
Construct a timer with zero-ed timestamps.

Error **`Reset`**`()`
Reset all timestamp values to zero.

Must be called before re-using the timer.

Return *Error* object indicating success or failure.

Error **`Record`**`(`*Kind kind*`)`
Record the current timestamp for a request stage.

Return *Error* object indicating success or failure.

Parameters

- `kind`: The Kind of the timestamp.

Class InferContext::Result

- Defined in *File request.h*

Nested Relationships

This class is a nested type of *Class InferContext*.

Nested Types

- *Struct Result::ClassResult*

Class Documentation

class Result
An inference result corresponding to an output.

Public Types

enum ResultFormat
Format in which result is returned.

Values:

RAW = 0
RAW format is the entire result tensor of values.

CLASS = 1
CLASS format is the top-k highest probability values of the result and the associated class label (if provided by the model).

Public Functions

virtual ~Result()
Destroy the result.

virtual const std::string &**ModelName**() **const** = 0

Return The name of the model that produced this result.

virtual int64_t **ModelVersion**() **const** = 0

Return The version of the model that produced this result.

virtual const std::shared_ptr<*Output*> **GetOutput**() **const** = 0

Return The *Output* object corresponding to this result.

virtual *Error* **GetRawShape**(std::vector<int64_t> **shape*) **const** = 0
Get the shape of a raw result.

The shape does not include the batch dimension.

Return *Error* object indicating success or failure.

Parameters

- `shape`: Returns the shape.

virtual *Error* **GetRaw**(size_t *batch_idx*, **const** std::vector<uint8_t> ***buf*) **const** = 0
Get a reference to entire raw result data for a specific batch entry.

Returns error if this result is not RAW format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: Returns the results for this entry of the batch.
- `buf`: Returns the vector of result bytes.

`virtual` *Error* **`GetRawAtCursor`** (size_t *batch_idx*, **`const`** uint8_t ***buf*, size_t *adv_byte_size*) = 0

Get a reference to raw result data for a specific batch entry at the current "cursor" and advance the cursor by the specified number of bytes.

More typically use *GetRawAtCursor<T>()* method to return the data as a specific type T. Use *ResetCursor()* to reset the cursor to the beginning of the result. Returns error if this result is not RAW format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: Returns results for this entry of the batch.
- `buf`: Returns pointer to 'adv_byte_size' bytes of data.
- `adv_byte_size`: The number of bytes of data to get a reference to.

`template <typename` T>
Error **`GetRawAtCursor`** (size_t *batch_idx*, T **out*)

Read a value for a specific batch entry at the current "cursor" from the result tensor as the specified type T and advance the cursor.

Use *ResetCursor()* to reset the cursor to the beginning of the result. Returns error if this result is not RAW format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: Returns results for this entry of the batch.
- `out`: Returns the value at the cursor.

`virtual` *Error* **`GetClassCount`** (size_t *batch_idx*, size_t **cnt*) **`const`** = 0

Get the number of class results for a batch.

Returns error if this result is not CLASS format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: The index in the batch.
- `cnt`: Returns the number of *ClassResult* entries for the batch entry.

`virtual` *Error* **`GetClassAtCursor`** (size_t *batch_idx*, *ClassResult* **result*) = 0

Get the *ClassResult* result for a specific batch entry at the current cursor.

Use *ResetCursor()* to reset the cursor to the beginning of the result. Returns error if this result is not CLASS format.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: The index in the batch.
- `result`: Returns the *ClassResult* value for the batch at the cursor.

`virtual` *Error* **`ResetCursors`** () = 0

Reset cursor to beginning of result for all batch entries.

Return *Error* object indicating success or failure.

virtual *Error* **ResetCursor**(size_t *batch_idx*) = 0

Reset cursor to beginning of result for specified batch entry.

Return *Error* object indicating success or failure.

Parameters

- `batch_idx`: The index in the batch.

template <>
Error **GetRawAtCursor**(size_t *batch_idx*, std::string **out*)

struct ClassResult

The result value for CLASS format results.

Public Members

size_t **idx**

The index of the class in the result vector.

float **value**

The value of the class.

std::string **label**

The label for the class, if provided by the model.

Class InferGrpcContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::InferContext` (*Class InferContext*)

Derived Type

- `public nvidia::inferenceserver::client::InferGrpcStreamContext` (*Class InferGrpcStreamContext*)

Class Documentation

class InferGrpcContext : **public** nvidia::inferenceserver::client::*InferContext*

InferGrpcContext is the GRPC instantiation of *InferContext*.

Subclassed by *nvidia::inferenceserver::client::InferGrpcStreamContext*

Public Functions

`virtual ~InferGrpcContext`()

`virtual` *Error* **`Run`**(ResultMap **results*)

Send a synchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `results`: Returns Result objects holding inference results as a map from output name to Result object.

`virtual` *Error* **`AsyncRun`**(std::shared_ptr<Request> **async_request*)

Send an asynchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `async_request`: Returns a Request object that can be used to retrieve the inference results for the request.

Error **`GetAsyncRunResults`**(ResultMap **results*, **`const`** std::shared_ptr<Request> &*async_request*, bool *wait*)

Get the results of the asynchronous request referenced by 'async_request'.

Return *Error* object indicating success or failure. Success will be returned only if the request has been completed succesfully. UNAVAILABLE will be returned if 'wait' is false and the request is not ready.

Parameters

- `results`: Returns Result objects holding inference results as a map from output name to Result object.
- `async_request`: Request handle to retrieve results.
- `wait`: If true, block until the request completes. Otherwise, return immediately.

Public Static Functions

`static` *Error* **`Create`**(std::unique_ptr<*InferContext*> **ctx*, **`const`** std::string &*server_url*, **`const`** std::string &*model_name*, int64_t *model_version* = -1, bool *verbose* = false)

Create context that performs inference for a non-sequence model using the GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *InferGrpcContext* object.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.

- `model_version`: The version of the model to use for inference, or -1 to indicate that the latest (i.e. highest version number) version should be used.
- `verbose`: If true generate verbose output when contacting the inference server.

static *Error* **Create**(std::unique_ptr<*InferContext*> **ctx*, CorrelationID *correlation_id*, **const** std::string &*server_url*, **const** std::string &*model_name*, int64_t *model_version* = -1, bool *verbose* = false)

Create context that performs inference for a sequence model using a given correlation ID and the GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *InferGrpcContext* object.
- `correlation_id`: The correlation ID to use for all inferences performed with this context. A value of 0 (zero) indicates that no correlation ID should be used.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `model_version`: The version of the model to use for inference, or -1 to indicate that the latest (i.e. highest version number) version should be used.
- `verbose`: If true generate verbose output when contacting the inference server.

Protected Functions

InferGrpcContext(**const** std::string&, **const** std::string&, int64_t, CorrelationID, bool)

Error **InitHelper**(**const** std::string &*server_url*, **const** std::string &*model_name*, bool *verbose*)

virtual void **AsyncTransfer**()

Error **PreRunProcessing**(std::shared_ptr<Request> &*request*)

Protected Attributes

grpc::CompletionQueue **async_request_completion_queue_**

std::unique_ptr<GRPCService::Stub> **stub_**

InferRequest **request_**

Class InferGrpcStreamContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::InferGrpcContext` (*Class InferGrpcContext*)

Class Documentation

`class InferGrpcStreamContext` : **`public`** nvidia::inferenceserver::client::*InferGrpcContext*

InferGrpcStreamContext is the streaming instantiation of *InferGrpcContext*.

All synchronous and asynchronous requests sent from this context will be sent in the same stream.

Public Functions

`~InferGrpcStreamContext`()

Error **`Run`**(ResultMap **results*)

Send a synchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `results`: Returns Result objects holding inference results as a map from output name to Result object.

Error **`AsyncRun`**(std::shared_ptr<Request> **async_request*)

Send an asynchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `async_request`: Returns a Request object that can be used to retrieve the inference results for the request.

Public Static Functions

`static` *Error* **`Create`**(std::unique_ptr<*InferContext*> **ctx*, **`const`** std::string &*server_url*, **`const`** std::string &*model_name*, int64_t *model_version* = -1, bool *verbose* = false)

Create streaming context that performs inference for a non-sequence model using the GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *InferGrpcContext* object.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `model_version`: The version of the model to use for inference, or -1 to indicate that the latest (i.e. highest version number) version should be used.
- `verbose`: If true generate verbose output when contacting the inference server.

static *Error* **Create**(std::unique_ptr<*InferContext*> **ctx*, CorrelationID *correlation_id*, **const** std::string &*server_url*, **const** std::string &*model_name*, int64_t *model_version* = -1, bool *verbose* = false)

Create streaming context that performs inference for a sequence model using a given correlation ID and the GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *InferGrpcContext* object.
- `correlation_id`: The correlation ID to use for all inferences performed with this context. A value of 0 (zero) indicates that no correlation ID should be used.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `model_version`: The version of the model to use for inference, or -1 to indicate that the latest (i.e. highest version number) version should be used.
- `verbose`: If true generate verbose output when contacting the inference server.

Class InferHttpContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::InferContext` (*Class InferContext*)

Class Documentation

class InferHttpContext : **public** nvidia::inferenceserver::client::*InferContext*

InferHttpContext is the HTTP instantiation of *InferContext*.

Public Functions

~InferHttpContext()

Error **Run**(ResultMap **results*)

Send a synchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `results`: Returns Result objects holding inference results as a map from output name to Result object.

Error **AsyncRun**(std::shared_ptr<Request> **async_request*)

Send an asynchronous request to the inference server to perform an inference to produce results for the outputs specified in the most recent call to *SetRunOptions()*.

Return *Error* object indicating success or failure.

Parameters

- `async_request`: Returns a Request object that can be used to retrieve the inference results for the request.

Error **GetAsyncRunResults**(ResultMap **results*, **const** std::shared_ptr<Request> &*async_request*, bool *wait*)

Get the results of the asynchronous request referenced by 'async_request'.

Return *Error* object indicating success or failure. Success will be returned only if the request has been completed succesfully. UNAVAILABLE will be returned if 'wait' is false and the request is not ready.

Parameters

- `results`: Returns Result objects holding inference results as a map from output name to Result object.
- `async_request`: Request handle to retrieve results.
- `wait`: If true, block until the request completes. Otherwise, return immediately.

Public Static Functions

static *Error* **Create**(std::unique_ptr<*InferContext*> **ctx*, **const** std::string &*server_url*, **const** std::string &*model_name*, int64_t *model_version* = -1, bool *verbose* = false)

Create context that performs inference for a non-sequence model using HTTP protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *InferHttpContext* object.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `model_version`: The version of the model to use for inference, or -1 to indicate that the latest (i.e. highest version number) version should be used.
- `verbose`: If true generate verbose output when contacting the inference server.

static *Error* **Create**(std::unique_ptr<*InferContext*> **ctx*, CorrelationID *correlation_id*, **const** std::string &*server_url*, **const** std::string &*model_name*, int64_t *model_version* = -1, bool *verbose* = false)

Create context that performs inference for a sequence model using a given correlation ID and the HTTP protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *InferHttpContext* object.

- `correlation_id`: The correlation ID to use for all inferences performed with this context. A value of 0 (zero) indicates that no correlation ID should be used.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `model_version`: The version of the model to use for inference, or -1 to indicate that the latest (i.e. highest version number) version should be used.
- `verbose`: If true generate verbose output when contacting the inference server.

Class ProfileContext

- Defined in *File request.h*

Inheritance Relationships

Derived Types

- `public nvidia::inferenceserver::client::ProfileGrpcContext` (*Class ProfileGrpcContext*)
- `public nvidia::inferenceserver::client::ProfileHttpContext` (*Class ProfileHttpContext*)

Class Documentation

`class ProfileContext`

A *ProfileContext* object is used to control profiling on the inference server.

Once created a *ProfileContext* object can be used repeatedly.

A *ProfileContext* object can use either HTTP protocol or GRPC protocol depending on the Create function (*ProfileHttpContext::Create* or *ProfileGrpcContext::Create*). For example:

```
std::unique_ptr<ProfileContext> ctx;
ProfileGrpcContext::Create(&ctx, "localhost:8000");
ctx->StartProfile();
...
ctx->StopProfile();
...
```

Note ProfileContext::Create methods are thread-safe. StartProfiling() and StopProfiling() are not thread-safe. For a given *ProfileContext*, calls to these methods must be serialized.

Subclassed by *nvidia::inferenceserver::client::ProfileGrpcContext*, *nvidia::inferenceserver::client::ProfileHttpContext*

Public Functions

Error **`StartProfile`**`()`

Start profiling on the inference server.

Return *Error* object indicating success or failure.

Error **`StopProfile`**()
Stop profiling on the inference server.

Protected Functions

`ProfileContext`(bool)

`virtual` *Error* **`SendCommand`**(**`const`** std::string &*cmd_str*) = 0

Protected Attributes

`const` bool **`verbose_`**

Class ProfileGrpcContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::ProfileContext` (*Class ProfileContext*)

Class Documentation

`class ProfileGrpcContext` : **`public`** nvidia::inferenceserver::client::*ProfileContext*

Public Static Functions

`static` *Error* **`Create`**(std::unique_ptr<*ProfileContext*> **ctx*, **`const`** std::string &*server_url*, bool *verbose* = false)
Create context that controls profiling on a server using GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns the new *ProfileContext* object.
- `server_url`: The inference server name and port.
- `verbose`: If true generate verbose output when contacting the inference server.

Class ProfileHttpContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::ProfileContext` (*Class ProfileContext*)

Class Documentation

class ProfileHttpContext : **public** nvidia::inferenceserver::client::*ProfileContext*

ProfileHttpContext is the HTTP instantiation of *ProfileContext*.

Public Static Functions

static *Error* **Create**(std::unique_ptr<*ProfileContext*> **ctx*, **const** std::string &*server_url*, bool *verbose* = false)

Create context that controls profiling on a server using HTTP protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns the new *ProfileContext* object.
- `server_url`: The inference server name and port.
- `verbose`: If true generate verbose output when contacting the inference server.

Class ServerHealthContext

- Defined in *File request.h*

Inheritance Relationships

Derived Types

- `public nvidia::inferenceserver::client::ServerHealthGrpcContext` (*Class ServerHealthGrpcContext*)
- `public nvidia::inferenceserver::client::ServerHealthHttpContext` (*Class ServerHealthHttpContext*)

Class Documentation

class ServerHealthContext

A *ServerHealthContext* object is used to query an inference server for health information.

Once created a *ServerHealthContext* object can be used repeatedly to get health from the server. A *ServerHealthContext* object can use either HTTP protocol or GRPC protocol depending on the Create function (*ServerHealthHttpContext::Create* or *ServerHealthGrpcContext::Create*). For example:

```
std::unique_ptr<ServerHealthContext> ctx;
  ServerHealthHttpContext::Create(&ctx, "localhost:8000");
  bool ready;
  ctx->GetReady(&ready);
  ...
  bool live;
  ctx->GetLive(&live);
  ...
```

Note ServerHealthContext::Create methods are thread-safe. *GetReady()* and *GetLive()* are not thread-safe. For a given *ServerHealthContext*, calls to *GetReady()* and *GetLive()* must be serialized.

Subclassed by *nvidia::inferenceserver::client::ServerHealthGrpcContext*, *nvidia::inferenceserver::client::ServerHealthHttpConte*

Public Functions

virtual *Error* **GetReady**(bool **ready*) = 0
Contact the inference server and get readiness state.

Return *Error* object indicating success or failure of the request.

Parameters

- `ready`: Returns the readiness state of the server.

virtual *Error* **GetLive**(bool **live*) = 0
Contact the inference server and get liveness state.

Return *Error* object indicating success or failure of the request.

Parameters

- `ready`: Returns the liveness state of the server.

Protected Functions

ServerHealthContext(bool)

Protected Attributes

const bool **verbose_**

Class ServerHealthGrpcContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::ServerHealthContext` (*Class ServerHealthContext*)

Class Documentation

`class ServerHealthGrpcContext` : **`public`** nvidia::inferenceserver::client::*ServerHealthContext*

ServerHealthGrpcContext is the GRPC instantiation of *ServerHealthContext*.

Public Functions

Error **`GetReady`**(bool **ready*)

Contact the inference server and get readiness state.

Return *Error* object indicating success or failure of the request.

Parameters

- `ready`: Returns the readiness state of the server.

Error **`GetLive`**(bool **live*)

Contact the inference server and get liveness state.

Return *Error* object indicating success or failure of the request.

Parameters

- `ready`: Returns the liveness state of the server.

Public Static Functions

`static` *Error* **`Create`**(std::unique_ptr<*ServerHealthContext*> **ctx*, **`const`** std::string &*server_url*, bool *verbose* = false)

Create a context that returns health information about server.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *ServerHealthGrpcContext* object.
- `server_url`: The inference server name and port.
- `verbose`: If true generate verbose output when contacting the inference server.

Class ServerHealthHttpContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::ServerHealthContext` (*Class ServerHealthContext*)

Class Documentation

`class ServerHealthHttpContext` : **`public`** nvidia::inferenceserver::client::*ServerHealthContext*

ServerHealthHttpContext is the HTTP instantiation of *ServerHealthContext*.

Public Functions

Error **`GetReady`**(bool **ready*)

Contact the inference server and get readiness state.

Return *Error* object indicating success or failure of the request.

Parameters

- `ready`: Returns the readiness state of the server.

Error **`GetLive`**(bool **live*)

Contact the inference server and get liveness state.

Return *Error* object indicating success or failure of the request.

Parameters

- `ready`: Returns the liveness state of the server.

Public Static Functions

`static` *Error* **`Create`**(std::unique_ptr<*ServerHealthContext*> **ctx*, **`const`** std::string &*server_url*, bool *verbose* = false)

Create a context that returns health information.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *ServerHealthHttpContext* object.
- `server_url`: The inference server name and port.
- `verbose`: If true generate verbose output when contacting the inference server.

Class ServerStatusContext

- Defined in *File request.h*

Inheritance Relationships

Derived Types

- `public nvidia::inferenceserver::client::ServerStatusGrpcContext` (*Class ServerStatusGrpcContext*)
- `public nvidia::inferenceserver::client::ServerStatusHttpContext` (*Class ServerStatusHttpContext*)

Class Documentation

class ServerStatusContext

A *ServerStatusContext* object is used to query an inference server for status information, including information about the models available on that server.

Once created a *ServerStatusContext* object can be used repeatedly to get status from the server. A *ServerStatusContext* object can use either HTTP protocol or GRPC protocol depending on the Create function (*ServerStatusHttpContext::Create* or *ServerStatusGrpcContext::Create*). For example:

```
std::unique_ptr<ServerStatusContext> ctx;
ServerStatusHttpContext::Create(&ctx, "localhost:8000");
ServerStatus status;
ctx->GetServerStatus(&status);
...
ctx->GetServerStatus(&status);
...
```

Note ServerStatusContext::Create methods are thread-safe. *GetServerStatus()* is not thread-safe. For a given *ServerStatusContext*, calls to *GetServerStatus()* must be serialized.

Subclassed by *nvidia::inferenceserver::client::ServerStatusGrpcContext*, *nvidia::inferenceserver::client::ServerStatusHttpContex*

Public Functions

virtual *Error* **GetServerStatus**(ServerStatus **status*) = 0

Contact the inference server and get status.

Return *Error* object indicating success or failure of the request.

Parameters

- `status`: Returns the status.

Protected Functions

ServerStatusContext(bool)

Protected Attributes

const bool **verbose_**

Class ServerStatusGrpcContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::ServerStatusContext` (*Class ServerStatusContext*)

Class Documentation

`class ServerStatusGrpcContext` : **`public`** nvidia::inferenceserver::client::*ServerStatusContext*

ServerStatusGrpcContext is the GRPC instantiation of *ServerStatusContext*.

Public Functions

Error **`GetServerStatus`** (ServerStatus **status*)

Contact the inference server and get status.

Return *Error* object indicating success or failure.

Parameters

- `status`: Returns the status.

Public Static Functions

`static` *Error* **`Create`** (std::unique_ptr<*ServerStatusContext*> **ctx*, **`const`** std::string &*server_url*, bool *verbose* = false)

Create a context that returns information about an inference server and all models on the server using GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *ServerStatusGrpcContext* object.
- `server_url`: The inference server name and port.
- `verbose`: If true generate verbose output when contacting the inference server.

`static` *Error* **`Create`** (std::unique_ptr<*ServerStatusContext*> **ctx*, **`const`** std::string &*server_url*, **`const`** std::string &*model_name*, bool *verbose* = false)

Create a context that returns information about an inference server and one model on the sever using GRPC protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *ServerStatusGrpcContext* object.

- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `verbose`: If true generate verbose output when contacting the inference server.

Class ServerStatusHttpContext

- Defined in *File request.h*

Inheritance Relationships

Base Type

- `public nvidia::inferenceserver::client::ServerStatusContext` (*Class ServerStatusContext*)

Class Documentation

`class ServerStatusHttpContext` : **`public`** nvidia::inferenceserver::client::*ServerStatusContext*

ServerStatusHttpContext is the HTTP instantiation of *ServerStatusContext*.

Public Functions

Error **`GetServerStatus`**(ServerStatus **status*)

Contact the inference server and get status.

Return *Error* object indicating success or failure.

Parameters

- `status`: Returns the status.

Public Static Functions

`static` *Error* **`Create`**(std::unique_ptr<*ServerStatusContext*> **ctx*, **`const`** std::string &*server_url*, bool *verbose* = false)

Create a context that returns information about an inference server and all models on the server using HTTP protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *ServerStatusHttpContext* object.
- `server_url`: The inference server name and port.
- `verbose`: If true generate verbose output when contacting the inference server.

static *Error* **Create**(std::unique_ptr<*ServerStatusContext*> **ctx*, **const** std::string &*server_url*, **const** std::string &*model_name*, bool *verbose* = false)

Create a context that returns information about an inference server and one model on the sever using HTTP protocol.

Return *Error* object indicating success or failure.

Parameters

- `ctx`: Returns a new *ServerStatusHttpContext* object.
- `server_url`: The inference server name and port.
- `model_name`: The name of the model to get status for.
- `verbose`: If true generate verbose output when contacting the inference server.

14.3.3 Functions

Function CustomErrorString

- Defined in *File custom.h*

Function Documentation

const char ***CustomErrorString**(void **custom_context*, int *errcode*)

Get the string for an error code.

Return The error code string, or nullptr if the error code has no string representation.

Parameters

- `custom_context`: The custom state associated with the error code. Can be nullptr if no custom state.
- `errcode`: The error code.

Function CustomExecute

- Defined in *File custom.h*

Function Documentation

int **CustomExecute**(void **custom_context*, uint32_t *payload_cnt*, *CustomPayload* **payloads*, *CustomGetNextInputFn_t input_fn*, *CustomGetOutputFn_t output_fn*)

Execute the custom model.

Return An error code. Zero indicates success, all other values indicate failure. Use CustomErrorString to get the error string for an error code.

Parameters

- `custom_context`: The custom state associated with the context that should execute. Can be nullptr if no custom state.
- `payload_cnt`: The number of payloads to execute.

- `payloads`: The payloads to execute.
- `input_fn`: The callback function to get tensor input (see CustomGetNextInputFn_t).
- `output_fn`: The callback function to get buffer for tensor output (see CustomGetOutputFn_t).

Function CustomFinalize

- Defined in *File custom.h*

Function Documentation

int **`CustomFinalize`**(void **custom_context*)

Finalize a custom context.

All state associated with the context should be freed.

Return An error code. Zero indicates success, all other values indicate failure. Use CustomErrorString to get the error string for an error code.

Parameters

- `custom_context`: The custom state associated with context that should be freed. Can be nullptr if no custom state.

Function CustomInitialize

- Defined in *File custom.h*

Function Documentation

int **`CustomInitialize`**(**`const`** char **serialized_model_config*, size_t *serialized_model_config_size*, int *gpu_device_id*, void ***custom_context*)

Initialize the custom shared library for a given model configuration and get the associated custom context.

Return An error code. Zero indicates success, all other values indicate failure. Use CustomErrorString to get the error string for an error code.

Parameters

- `serialized_model_config`: Serialized representation of the model configuration to use for initialization. This serialization is owned by the caller and so must be copied if a persistent copy of required by the shared library.
- `serialized_model_config_size`: The size of serialized_model_config, in bytes.
- `gpu_device_id`: The GPU device ID to initialize for, or CUSTOM_NO_GPU_DEVICE if should initialize for CPU.
- `custom_context`: Returns the opaque handle to the custom state associated with this initialization. Returns nullptr if no context associated with the initialization.

Function nvidia::inferenceserver::client::operator<<

- Defined in *File request.h*

Function Documentation

std::ostream &`nvidia::inferenceserver::client::`**`operator<<`**(std::ostream&, **`const`** *Error*&)

14.3.4 Defines

Define CUSTOM_NO_GPU_DEVICE

- Defined in *File custom.h*

Define Documentation

`CUSTOM_NO_GPU_DEVICE`

GPU device number that indicates that no GPU is available for a context.

In CustomInitialize this value is used for 'gpu_device_id' to indicate that the model must execute on the CPU.

14.3.5 Typedefs

Typedef CustomErrorStringFn_t

- Defined in *File custom.h*

Typedef Documentation

`typedef` char *(***`CustomErrorStringFn_t`**)(void *, int)

Type for the CustomErrorString function.

Typedef CustomExecuteFn_t

- Defined in *File custom.h*

Typedef Documentation

`typedef` int (***`CustomExecuteFn_t`**)(void *, uint32_t, *CustomPayload* *, *CustomGetNextInputFn_t*, *CustomGetOutputFn_t*)

Type for the CustomExecute function.

Typedef CustomFinalizeFn_t

- Defined in *File custom.h*

Typedef Documentation

`typedef` int (***`CustomFinalizeFn_t`**)(void *)

Type for the CustomFinalize function.

Typedef CustomGetNextInputFn_t

- Defined in *File custom.h*

Typedef Documentation

typedef bool (***CustomGetNextInputFn_t**)(void *input_context, **const** char *name, **const** void **content, uint64_t *content_byte_size)

Type for the CustomGetNextInput callback function.

This callback function is provided in the call to ComputeExecute and is used to get the value of the input tensors. Each call to this function returns a contiguous block of the input tensor value. The entire tensor value may be in multiple non-contiguous blocks and so this function must be called multiple times until 'content' returns nullptr.

Return false if error, true if success.

Parameters

- `input_context`: The input context provided in call to CustomExecute.
- `name`: The name of the input tensor.
- `content`: Returns a pointer to the next contiguous block of content for the named input. Returns nullptr if there is no more content for the input.
- `content_byte_size`: Acts as both input and output. On input gives the maximum size expected for 'content'. Returns the actual size, in bytes, of 'content'.

Typedef CustomGetOutputFn_t

- Defined in *File custom.h*

Typedef Documentation

typedef bool (***CustomGetOutputFn_t**)(void *output_context, **const** char *name, size_t shape_dim_cnt, int64_t *shape_dims, uint64_t content_byte_size, void **content)

Type for the CustomGetOutput callback function.

This callback function is provided in the call to ComputeExecute and is used to report the shape of an output and to get the buffers to store the output tensor values.

Return false if error, true if success.

Parameters

- `output_context`: The output context provided in call to CustomExecute.
- `name`: The name of the output tensor.
- `shape_dim_cnt`: The number of dimensions in the output shape.
- `shape_dims`: The dimensions of the output shape.
- `content_byte_size`: The size, in bytes, of the output tensor.

    - `content`: Returns a pointer to a buffer where the output for the tensor should be copied. If nullptr and function returns true (no error), then the output should not be written and the backend should continue to the next output. If non-nullptr, the size of the buffer will be large enough to hold 'content_byte_size' bytes.

Typedef CustomInitializeFn_t

- Defined in *File custom.h*

Typedef Documentation

typedef int (***CustomInitializeFn_t**)(**const** char *, size_t, int, void **)
    Type for the CustomInitialize function.

Typedef CustomPayload

- Defined in *File custom.h*

Typedef Documentation

typedef struct *custom_payload_struct* **CustomPayload**

14.3.6 Directories

Directory src

Directory path: `src`

Subdirectories

- *Directory clients*
- *Directory servables*

Directory clients

Parent directory (`src`)

Directory path: `src/clients`

Subdirectories

- *Directory c++*

Directory c++

Parent directory (`src/clients`)

Directory path: `src/clients/c++`

Files

- *File request.h*

Directory servables

Parent directory (`src`)

Directory path: `src/servables`

Subdirectories

- *Directory custom*

Directory custom

Parent directory (`src/servables`)

Directory path: `src/servables/custom`

Files

- *File custom.h*

14.3.7 Files

File custom.h

Parent directory (`src/servables/custom`)

Contents

Definition (`src/servables/custom/custom.h`)

Program Listing for File custom.h

Return to documentation for file (`src/servables/custom/custom.h`)

```
// Copyright (c) 2018-2019, NVIDIA CORPORATION. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//  * Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
//  * Redistributions in binary form must reproduce the above copyright
//    notice, this list of conditions and the following disclaimer in the
//    documentation and/or other materials provided with the distribution.
//  * Neither the name of NVIDIA CORPORATION nor the names of its
//    contributors may be used to endorse or promote products derived
//    from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
// PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
// CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
// EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
// OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#pragma once

#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

#define CUSTOM_NO_GPU_DEVICE -1

// A payload represents the input tensors and the required output
// needed for execution in the backend.
typedef struct custom_payload_struct {
  // The size of the batch represented by this payload.
  uint32_t batch_size;

  // The number of inputs included in this payload.
  uint32_t input_cnt;

  // The 'input_cnt' names of the inputs included in this payload.
  const char** input_names;

  // For each of the 'input_cnt' inputs, the number of dimensions in
  // the input's shape, not including the batch dimension.
  const size_t* input_shape_dim_cnts;
```

(continues on next page)

(continued from previous page)

```
  // For each of the 'input_cnt' inputs, the shape of the input, not
  // including the batch dimension.
  const int64_t** input_shape_dims;

  // The number of outputs that must be computed for this payload. Can
  // be 0 to indicate that no outputs are required from the backend.
  uint32_t output_cnt;

  // The 'output_cnt' names of the outputs that must be computed for
  // this payload. Each name must be one of the names from the model
  // configuration, but all outputs do not need to be computed.
  const char** required_output_names;

  // The context to use with CustomGetNextInput callback function to
  // get the input tensor values for this payload.
  void* input_context;

  // The context to use with CustomGetOutput callback function to get
  // the buffer for output tensor values for this payload.
  void* output_context;

  // The error code indicating success or failure from execution. A
  // value of 0 (zero) indicates success, all other values indicate
  // failure and are backend defined.
  int error_code;
} CustomPayload;

typedef bool (*CustomGetNextInputFn_t)(
    void* input_context, const char* name, const void** content,
    uint64_t* content_byte_size);

typedef bool (*CustomGetOutputFn_t)(
    void* output_context, const char* name, size_t shape_dim_cnt,
    int64_t* shape_dims, uint64_t content_byte_size, void** content);

typedef int (*CustomInitializeFn_t)(const char*, size_t, int, void**);

typedef int (*CustomFinalizeFn_t)(void*);

typedef char* (*CustomErrorStringFn_t)(void*, int);

typedef int (*CustomExecuteFn_t)(
    void*, uint32_t, CustomPayload*, CustomGetNextInputFn_t,
    CustomGetOutputFn_t);

int CustomInitialize(
    const char* serialized_model_config, size_t serialized_model_config_size,
    int gpu_device_id, void** custom_context);

int CustomFinalize(void* custom_context);

const char* CustomErrorString(void* custom_context, int errcode);

int CustomExecute(
    void* custom_context, uint32_t payload_cnt, CustomPayload* payloads,
    CustomGetNextInputFn_t input_fn, CustomGetOutputFn_t output_fn);
```

(continues on next page)

(continued from previous page)

```
#ifdef __cplusplus
}
#endif
```

Includes

- `stddef.h`
- `stdint.h`

Classes

- *Struct custom_payload_struct*

Functions

- *Function CustomErrorString*
- *Function CustomExecute*
- *Function CustomFinalize*
- *Function CustomInitialize*

Defines

- *Define CUSTOM_NO_GPU_DEVICE*

Typedefs

- *Typedef CustomErrorStringFn_t*
- *Typedef CustomExecuteFn_t*
- *Typedef CustomFinalizeFn_t*
- *Typedef CustomGetNextInputFn_t*
- *Typedef CustomGetOutputFn_t*
- *Typedef CustomInitializeFn_t*
- *Typedef CustomPayload*

File request.h

Parent directory (`src/clients/c++`)

Contents

Definition (`src/clients/c++/request.h`)

Program Listing for File request.h

Return to documentation for file (`src/clients/c++/request.h`)

```
// Copyright (c) 2018-2019, NVIDIA CORPORATION. All rights reserved.
//
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
//  * Redistributions of source code must retain the above copyright
//    notice, this list of conditions and the following disclaimer.
//  * Redistributions in binary form must reproduce the above copyright
//    notice, this list of conditions and the following disclaimer in the
//    documentation and/or other materials provided with the distribution.
//  * Neither the name of NVIDIA CORPORATION nor the names of its
//    contributors may be used to endorse or promote products derived
//    from this software without specific prior written permission.
//
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS ``AS IS'' AND ANY
// EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
// PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
// CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
// EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
// OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
// (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
// OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#pragma once


#include <curl/curl.h>
#include <grpcpp/grpcpp.h>
#include <condition_variable>
#include <memory>
#include <mutex>
#include <string>
#include <thread>
#include <vector>
#include "src/core/api.pb.h"
#include "src/core/grpc_service.grpc.pb.h"
```

(continues on next page)

(continued from previous page)

```
#include "src/core/grpc_service.pb.h"
#include "src/core/model_config.h"
#include "src/core/model_config.pb.h"
#include "src/core/request_status.pb.h"
#include "src/core/server_status.pb.h"

namespace nvidia { namespace inferenceserver { namespace client {

//==============================================================================
class Error {
 public:
  explicit Error(const RequestStatus& status);

  explicit Error(RequestStatusCode code = RequestStatusCode::SUCCESS);

  explicit Error(RequestStatusCode code, const std::string& msg);

  RequestStatusCode Code() const { return code_; }

  const std::string& Message() const { return msg_; }

  const std::string& ServerId() const { return server_id_; }

  uint64_t RequestId() const { return request_id_; }

  bool IsOk() const { return code_ == RequestStatusCode::SUCCESS; }

  static const Error Success;

 private:
  friend std::ostream& operator<<(std::ostream&, const Error&);
  RequestStatusCode code_;
  std::string msg_;
  std::string server_id_;
  uint64_t request_id_;
};

//==============================================================================
class ServerHealthContext {
 public:
  virtual Error GetReady(bool* ready) = 0;

  virtual Error GetLive(bool* live) = 0;

 protected:
  ServerHealthContext(bool);

  // If true print verbose output
  const bool verbose_;
};

//==============================================================================
class ServerStatusContext {
 public:
  virtual Error GetServerStatus(ServerStatus* status) = 0;

 protected:
```

(continues on next page)

(continued from previous page)

```
  ServerStatusContext(bool);

  // If true print verbose output
  const bool verbose_;
};

//==============================================================================
class InferContext {
 public:
  //==============
  class Input {
   public:
    virtual ~Input(){};

    virtual const std::string& Name() const = 0;

    virtual int64_t ByteSize() const = 0;

    virtual size_t TotalByteSize() const = 0;

    virtual DataType DType() const = 0;

    virtual ModelInput::Format Format() const = 0;

    virtual const DimsList& Dims() const = 0;

    virtual Error Reset() = 0;

    virtual const std::vector<int64_t>& Shape() const = 0;

    virtual Error SetShape(const std::vector<int64_t>& dims) = 0;

    virtual Error SetRaw(const uint8_t* input, size_t input_byte_size) = 0;

    virtual Error SetRaw(const std::vector<uint8_t>& input) = 0;

    virtual Error SetFromString(const std::vector<std::string>& input) = 0;
  };

  //==============
  class Output {
   public:
    virtual ~Output(){};

    virtual const std::string& Name() const = 0;

    virtual DataType DType() const = 0;

    virtual const DimsList& Dims() const = 0;
  };

  //==============
  class Result {
   public:
    virtual ~Result(){};

    enum ResultFormat {
```

(continues on next page)

(continued from previous page)

```
    RAW = 0,

    CLASS = 1
  };

  virtual const std::string& ModelName() const = 0;

  virtual int64_t ModelVersion() const = 0;

  virtual const std::shared_ptr<Output> GetOutput() const = 0;

  virtual Error GetRawShape(std::vector<int64_t>* shape) const = 0;

  virtual Error GetRaw(
      size_t batch_idx, const std::vector<uint8_t>** buf) const = 0;

  virtual Error GetRawAtCursor(
      size_t batch_idx, const uint8_t** buf, size_t adv_byte_size) = 0;

  template <typename T>
  Error GetRawAtCursor(size_t batch_idx, T* out);

  struct ClassResult {
    size_t idx;
    float value;
    std::string label;
  };

  virtual Error GetClassCount(size_t batch_idx, size_t* cnt) const = 0;

  virtual Error GetClassAtCursor(size_t batch_idx, ClassResult* result) = 0;

  virtual Error ResetCursors() = 0;

  virtual Error ResetCursor(size_t batch_idx) = 0;
};

//==============
class Options {
 public:
  virtual ~Options(){};

  static Error Create(std::unique_ptr<Options>* options);

  virtual bool Flag(InferRequestHeader::Flag flag) const = 0;

  virtual void SetFlag(InferRequestHeader::Flag flag, bool value) = 0;

  virtual uint32_t Flags() const = 0;

  virtual void SetFlags(uint32_t flags) = 0;

  virtual size_t BatchSize() const = 0;

  virtual void SetBatchSize(size_t batch_size) = 0;

  virtual Error AddRawResult(
```

(continues on next page)

(continued from previous page)

```
        const std::shared_ptr<InferContext::Output>& output) = 0;

    virtual Error AddClassResult(
        const std::shared_ptr<InferContext::Output>& output, uint64_t k) = 0;
  };

  //==============
  class Request {
   public:
    virtual ~Request() = default;

    virtual uint64_t Id() const = 0;
  };

  //==============
  struct Stat {
    size_t completed_request_count;

    uint64_t cumulative_total_request_time_ns;

    uint64_t cumulative_send_time_ns;

    uint64_t cumulative_receive_time_ns;

    Stat()
        : completed_request_count(0), cumulative_total_request_time_ns(0),
          cumulative_send_time_ns(0), cumulative_receive_time_ns(0)
    {
    }
  };

  //==============
  class RequestTimers {
   public:
    enum Kind {
      REQUEST_START,
      REQUEST_END,
      SEND_START,
      SEND_END,
      RECEIVE_START,
      RECEIVE_END
    };

    RequestTimers();

    Error Reset();

    Error Record(Kind kind);

   private:
    friend class InferContext;
    friend class InferHttpContext;
    friend class InferGrpcContext;
    friend class InferGrpcStreamContext;
    struct timespec request_start_;
    struct timespec request_end_;
    struct timespec send_start_;
```

(continues on next page)

(continued from previous page)

```
    struct timespec send_end_;
    struct timespec receive_start_;
    struct timespec receive_end_;
  };

 public:
  using ResultMap = std::map<std::string, std::unique_ptr<Result>>;

  virtual ~InferContext() = default;

  const std::string& ModelName() const { return model_name_; }

  int64_t ModelVersion() const { return model_version_; }

  uint64_t MaxBatchSize() const { return max_batch_size_; }

  const std::vector<std::shared_ptr<Input>>& Inputs() const { return inputs_; }

  const std::vector<std::shared_ptr<Output>>& Outputs() const
  {
    return outputs_;
  }

  Error GetInput(const std::string& name, std::shared_ptr<Input>* input) const;

  Error GetOutput(
      const std::string& name, std::shared_ptr<Output>* output) const;

  Error SetRunOptions(const Options& options);

  Error GetStat(Stat* stat);

  virtual Error Run(ResultMap* results) = 0;

  virtual Error AsyncRun(std::shared_ptr<Request>* async_request) = 0;

  virtual Error GetAsyncRunResults(
      ResultMap* results, const std::shared_ptr<Request>& async_request,
      bool wait) = 0;

  Error GetReadyAsyncRequest(
      std::shared_ptr<Request>* async_request, bool wait);

 protected:
  InferContext(const std::string&, int64_t, CorrelationID, bool);

  // Function for worker thread to proceed the data transfer for all requests
  virtual void AsyncTransfer() = 0;

  // Helper function called before inference to prepare 'request'
  virtual Error PreRunProcessing(std::shared_ptr<Request>& request) = 0;

  // Helper function called by GetAsyncRunResults() to check if the request
  // is ready. If the request is valid and wait == true,
  // the function will block until request is ready.
  Error IsRequestReady(
      const std::shared_ptr<Request>& async_request, bool wait);
```

(continues on next page)

(continued from previous page)

```
// Update the context stat with the given timer
Error UpdateStat(const RequestTimers& timer);

using AsyncReqMap = std::map<uintptr_t, std::shared_ptr<Request>>;

// map to record ongoing asynchronous requests with pointer to easy handle
// as key
AsyncReqMap ongoing_async_requests_;

// Model name
const std::string model_name_;

// Model version
const int64_t model_version_;

// The correlation ID to use with all inference requests using this
// context. A value of 0 (zero) indicates no correlation ID.
const CorrelationID correlation_id_;

// If true print verbose output
const bool verbose_;

// Maximum batch size supported by this context. A maximum batch
// size indicates that the context does not support batching and so
// only a single inference at a time can be performed.
uint64_t max_batch_size_;

// Requested batch size for inference request
uint64_t batch_size_;

// Use to assign unique identifier for each asynchronous request
uint64_t async_request_id_;

// The inputs and outputs
std::vector<std::shared_ptr<Input>> inputs_;
std::vector<std::shared_ptr<Output>> outputs_;

// Settings generated by current option
// InferRequestHeader protobuf describing the request
InferRequestHeader infer_request_;

// Standalone request context used for synchronous request
std::shared_ptr<Request> sync_request_;

// The statistic of the current context
Stat context_stat_;

// worker thread that will perform the asynchronous transfer
std::thread worker_;

// Avoid race condition between main thread and worker thread
std::mutex mutex_;

// Condition variable used for waiting on asynchronous request
std::condition_variable cv_;
```

(continues on next page)

(continued from previous page)

```
  // signal for worker thread to stop
  bool exiting_;
};

//==============================================================================
class ProfileContext {
 public:
  Error StartProfile();

  // \return Error object indicating success or failure.
  Error StopProfile();

 protected:
  ProfileContext(bool);
  virtual Error SendCommand(const std::string& cmd_str) = 0;

  // If true print verbose output
  const bool verbose_;
};

//==============================================================================
class ServerHealthHttpContext : public ServerHealthContext {
 public:
  static Error Create(
      std::unique_ptr<ServerHealthContext>* ctx, const std::string& server_url,
      bool verbose = false);

  Error GetReady(bool* ready) override;
  Error GetLive(bool* live) override;

 private:
  ServerHealthHttpContext(const std::string&, bool);
  Error GetHealth(const std::string& url, bool* health);

  // URL for health endpoint on inference server.
  const std::string url_;
};

//==============================================================================
class ServerStatusHttpContext : public ServerStatusContext {
 public:
  static Error Create(
      std::unique_ptr<ServerStatusContext>* ctx, const std::string& server_url,
      bool verbose = false);

  static Error Create(
      std::unique_ptr<ServerStatusContext>* ctx, const std::string& server_url,
      const std::string& model_name, bool verbose = false);

  Error GetServerStatus(ServerStatus* status) override;

 private:
  static size_t ResponseHeaderHandler(void*, size_t, size_t, void*);
  static size_t ResponseHandler(void*, size_t, size_t, void*);

  ServerStatusHttpContext(const std::string&, bool);
  ServerStatusHttpContext(const std::string&, const std::string&, bool);
```

(continues on next page)

(continued from previous page)

```
  // URL for status endpoint on inference server.
  const std::string url_;

  // RequestStatus received in server response
  RequestStatus request_status_;

  // Serialized ServerStatus response from server.
  std::string response_;
};

//==============================================================================
class InferHttpContext : public InferContext {
 public:
  ~InferHttpContext() override;

  static Error Create(
      std::unique_ptr<InferContext>* ctx, const std::string& server_url,
      const std::string& model_name, int64_t model_version = -1,
      bool verbose = false);

  static Error Create(
      std::unique_ptr<InferContext>* ctx, CorrelationID correlation_id,
      const std::string& server_url, const std::string& model_name,
      int64_t model_version = -1, bool verbose = false);

  Error Run(ResultMap* results) override;
  Error AsyncRun(std::shared_ptr<Request>* async_request) override;
  Error GetAsyncRunResults(
      ResultMap* results, const std::shared_ptr<Request>& async_request,
      bool wait) override;

 private:
  static size_t RequestProvider(void*, size_t, size_t, void*);
  static size_t ResponseHeaderHandler(void*, size_t, size_t, void*);
  static size_t ResponseHandler(void*, size_t, size_t, void*);

  InferHttpContext(
      const std::string&, const std::string&, int64_t, CorrelationID, bool);

  // @see InferContext.AsyncTransfer()
  void AsyncTransfer() override;

  // @see InferContext.PreRunProcessing()
  Error PreRunProcessing(std::shared_ptr<Request>& request) override;

  // curl multi handle for processing asynchronous requests
  CURLM* multi_handle_;

  // URL to POST to
  std::string url_;

  // Serialized InferRequestHeader
  std::string infer_request_str_;

  // Keep an easy handle alive to reuse the connection
  CURL* curl_;
```

(continues on next page)

(continued from previous page)

```
};

//==============================================================================
class ProfileHttpContext : public ProfileContext {
 public:
  static Error Create(
      std::unique_ptr<ProfileContext>* ctx, const std::string& server_url,
      bool verbose = false);

 private:
  static size_t ResponseHeaderHandler(void*, size_t, size_t, void*);

  ProfileHttpContext(const std::string&, bool);
  Error SendCommand(const std::string& cmd_str) override;

  // URL for status endpoint on inference server.
  const std::string url_;

  // RequestStatus received in server response
  RequestStatus request_status_;
};

//==============================================================================
class ServerHealthGrpcContext : public ServerHealthContext {
 public:
  static Error Create(
      std::unique_ptr<ServerHealthContext>* ctx, const std::string& server_url,
      bool verbose = false);

  Error GetReady(bool* ready) override;
  Error GetLive(bool* live) override;

 private:
  ServerHealthGrpcContext(const std::string&, bool);
  Error GetHealth(const std::string& mode, bool* health);

  // GRPC end point.
  std::unique_ptr<GRPCService::Stub> stub_;
};

//==============================================================================
class ServerStatusGrpcContext : public ServerStatusContext {
 public:
  static Error Create(
      std::unique_ptr<ServerStatusContext>* ctx, const std::string& server_url,
      bool verbose = false);

  static Error Create(
      std::unique_ptr<ServerStatusContext>* ctx, const std::string& server_url,
      const std::string& model_name, bool verbose = false);

  Error GetServerStatus(ServerStatus* status) override;

 private:
  ServerStatusGrpcContext(const std::string&, bool);
  ServerStatusGrpcContext(const std::string&, const std::string&, bool);
```

(continues on next page)

(continued from previous page)

```
  // Model name
  const std::string model_name_;

  // GRPC end point.
  std::unique_ptr<GRPCService::Stub> stub_;
};

//==============================================================================
class InferGrpcContext : public InferContext {
 public:
  virtual ~InferGrpcContext() override;

  static Error Create(
      std::unique_ptr<InferContext>* ctx, const std::string& server_url,
      const std::string& model_name, int64_t model_version = -1,
      bool verbose = false);

  static Error Create(
      std::unique_ptr<InferContext>* ctx, CorrelationID correlation_id,
      const std::string& server_url, const std::string& model_name,
      int64_t model_version = -1, bool verbose = false);

  virtual Error Run(ResultMap* results) override;
  virtual Error AsyncRun(std::shared_ptr<Request>* async_request) override;
  Error GetAsyncRunResults(
      ResultMap* results, const std::shared_ptr<Request>& async_request,
      bool wait) override;

 protected:
  InferGrpcContext(
      const std::string&, const std::string&, int64_t, CorrelationID, bool);

  // Helper function to initialize the context
  Error InitHelper(
      const std::string& server_url, const std::string& model_name,
      bool verbose);

  // @see InferContext.AsyncTransfer()
  virtual void AsyncTransfer() override;

  // @see InferContext.PreRunProcessing()
  Error PreRunProcessing(std::shared_ptr<Request>& request) override;

  // The producer-consumer queue used to communicate asynchronously with
  // the GRPC runtime.
  grpc::CompletionQueue async_request_completion_queue_;

  // GRPC end point.
  std::unique_ptr<GRPCService::Stub> stub_;

  // request for GRPC call, one request object can be used for multiple calls
  // since it can be overwritten as soon as the GRPC send finishes.
  InferRequest request_;
};

//==============================================================================
class InferGrpcStreamContext : public InferGrpcContext {
```

(continues on next page)

(continued from previous page)

```
 public:
  ~InferGrpcStreamContext() override;

  static Error Create(
      std::unique_ptr<InferContext>* ctx, const std::string& server_url,
      const std::string& model_name, int64_t model_version = -1,
      bool verbose = false);

  static Error Create(
      std::unique_ptr<InferContext>* ctx, CorrelationID correlation_id,
      const std::string& server_url, const std::string& model_name,
      int64_t model_version = -1, bool verbose = false);

  Error Run(ResultMap* results) override;
  Error AsyncRun(std::shared_ptr<Request>* async_request) override;

 private:
  InferGrpcStreamContext(
      const std::string&, const std::string&, int64_t, CorrelationID, bool);

  // @see InferContext.AsyncTransfer()
  void AsyncTransfer() override;

  // gRPC objects for using the streaming API
  grpc::ClientContext context_;
  std::shared_ptr<grpc::ClientReaderWriter<InferRequest, InferResponse>>
      stream_;
};

//==============================================================================
class ProfileGrpcContext : public ProfileContext {
 public:
  static Error Create(
      std::unique_ptr<ProfileContext>* ctx, const std::string& server_url,
      bool verbose = false);

 private:
  ProfileGrpcContext(const std::string&, bool);
  Error SendCommand(const std::string& cmd_str) override;

  // GRPC end point.
  std::unique_ptr<GRPCService::Stub> stub_;
};

//==============================================================================

std::ostream& operator<<(std::ostream&, const Error&);

template <>
Error InferContext::Result::GetRawAtCursor(size_t batch_idx, std::string* out);

template <typename T>
Error
InferContext::Result::GetRawAtCursor(size_t batch_idx, T* out)
{
  const uint8_t* buf;
  Error err = GetRawAtCursor(batch_idx, &buf, sizeof(T));
```

(continues on next page)

(continued from previous page)

```
  if (!err.IsOk()) {
    return err;
  }

  std::copy(buf, buf + sizeof(T), reinterpret_cast<uint8_t*>(out));
  return Error::Success;
}

}}}  // namespace nvidia::inferenceserver::client
```

Includes

- `condition_variable`
- `curl/curl.h`
- `grpcpp/grpcpp.h`
- `memory`
- `mutex`
- `src/core/api.pb.h`
- `src/core/grpc_service.grpc.pb.h`
- `src/core/grpc_service.pb.h`
- `src/core/model_config.h`
- `src/core/model_config.pb.h`
- `src/core/request_status.pb.h`
- `src/core/server_status.pb.h`
- `string`
- `thread`
- `vector`

Namespaces

- *Namespace nvidia*
- *Namespace nvidia::inferenceserver*
- *Namespace nvidia::inferenceserver::client*

Classes

- *Struct Result::ClassResult*
- *Struct InferContext::Stat*
- *Class Error*
- *Class InferContext*

- *Class InferContext::Input*
- *Class InferContext::Options*
- *Class InferContext::Output*
- *Class InferContext::Request*
- *Class InferContext::RequestTimers*
- *Class InferContext::Result*
- *Class InferGrpcContext*
- *Class InferGrpcStreamContext*
- *Class InferHttpContext*
- *Class ProfileContext*
- *Class ProfileGrpcContext*
- *Class ProfileHttpContext*
- *Class ServerHealthContext*
- *Class ServerHealthGrpcContext*
- *Class ServerHealthHttpContext*
- *Class ServerStatusContext*
- *Class ServerStatusGrpcContext*
- *Class ServerStatusHttpContext*

Functions

- *Function nvidia::inferenceserver::client::operator<<*

CHAPTER

FIFTEEN

PYTHON API

15.1 Client

CHAPTER

SIXTEEN

INDICES AND TABLES

- genindex