BRIAR

Generated by Doxygen 1.8.17

Chapter 1

BRIAR API

Authors

Name	Organization
David Bolme	Oak Ridge National Laboratory
Joel Brogan	Oak Ridge National Laboratory
Ian Shelley	Oak Ridge National Laboratory
Bob Zhang	Oak Ridge National Laboratory

Table of Contents

[[*TOC*]]

Overview

The BRIAR Application Programming Interface (API) defines a unified protocol that allows performer teams on BR↩ IAR to conform their research algorithms to a small set of high-level functions. Performers will implement a particular set of high-level functions that take specific input and output formats. These high level functions include:

- Detection
- Tracking
- Extraction
- Verification
- Enrollment
- Search
- Enhancement

This unified interface will allow for uniform and fair testing and evaluation of each algorithmic solutions across different BRIAR performers, while also providing a simple and straight-forward way of interacting with algorithms both programmatically and through a Command Line Interface (CLI).

The BRIAR API is implemented using the google Remote Procedure Call (gRPC) framework (https://grpc.io/). The BRIAR API uses this paradigm to interact with performer implementations via a **Client-Server Interface**, where performers' implementations of algorithms comprise a **Server**, while a pre-implemented API comprises the **Client**. In this way, the BRIAR Client can seamlessly interact with algorithms written in many different languages. To accomplish this, a set of protobuf messages are defined that can be passed between the client and server as requests and responses to different function calls. *All performers must do is implement code for each function that accept input and generate output in the form of these messages.*

This Repository contains 3 main components:

- lib/python -> a python implementation of a fully functional CLI client to interface with BRIAR performer algorithms
  - lib/python/briar_client.py -> a fully functional Python API that allows for calling functions in BRIAR performer algorithms
  - lib/python/cli/* -> function call implementations using the API that provide interaction via the command line
- proto/briar/briar_grpc -> protobuf files that define the messages and services that comprise the BRIAR API

gRPC Interface

Main Components

gRPC is a proto-language that defines a **service** which contains multiple **functions** that each pass strictly defined **messages** back and forth. **Messages** can be thought of as class definitions, and are defined to have strongly-typed **fields**. These **fields** can either be primitive types, or other messages themselves. The files that define the proto structure for BRIAR are located in proto/briar/briar_grpc

Stubs and services

A special compiler called **Protoc** can compile gRPC **Services**, along with their subsequent **functions** and **messages**, down into a large selection of different languages. Once the language is chosen, the Protoc compiler provides an **abstract service implementation** and its **unimplemented functions** in that language, that can be subclassed by performers as they build their service. The Protoc compiler also provides a **client stub** of that given language which connects to an instance of the **service implementation** to allow you to make function calls. The BRIAR API and CLI utilizes a compiled Python stub to make calls to one of these services.

Protobuf Documentation

We have compiled auto-generated documentation based on protobuf doc strings. These can be located for each of our supported languages at proto/briar/briar_grpc/doxyfile-stubs-*

To learn more about gRPC, you can read the documentation at this link: https://grpc.↩io/docs/what-is-grpc/introduction/

Language Support

While the client and command line tools are written in python, as well as the example service, the API is designed to work with any language supported by gRPC. This is done through the use of language agnostic gRPC messages which are used to communicate between the **client** (where commands are invoked) and the **service** (where commands are processed). This communication layer is cross-platform, cross-language, and can be used to communicate across networks so the client and service need not reside on the same system.

A list of supported gRPC languages can be found here: https://grpc.io/docs/languages/

As stated above, because communication between the client and service relies on the gRPC communication protocol, so long as you conform to using said protocol and the messages designated with the protobuf files, the service can be written in any supported language. Furthermore, multiple clients and services could theoretically all be written in different languages, such as having four services written each in Python, C#, C++, and PHP and all four would be able to communicate within the larger 'BRIAR ecosystem'.

While a mostly bare python example service is given, it is expected that performers be able to implement their detection/identification algorithms within their own service in whatever supported language they choose using the gRPC communication layer as an API. A fully functioning python client is provided to act as an interface with any service which a performer may create; however, while unnecessary because the python client is intended to be feature complete and extendable, there is nothing stopping a potential developer from writing their own client in a language of their choice using the provided gRPC stubs.

Installation

These instructions will set up the API and command line and will establish a python development environment for using the API. If you are using another language, additional steps may be required to add the pre-generated gRPC stubs (found in lib/[language_name]) to incorporate them into your development environment

1. (Optional) Use a conda environment
   (a) Create a conda environment with `conda create --name briar`
   (b) Activate the environment with `conda activate briar`

If you do not wish to create an anaconda environment, start here:

1. Clone the repo at https://code.ornl.gov/briar1/briar-api
   (a) If you do not have permission to view the repository, contact the repository's maintainer at bolmeds@ornl.gov
2. Navigate to the repository python lib folder in a terminal at: `briar-api/lib/python/briar/`
3. Install the python dependencies: `pip install -r requirements.txt`
4. Return to the repository's root directory
5. Build the protobuf stubs: `./build-proto-stubs.sh`
   (a) May need to make the script executable : `chmod +x ./build-proto-stubs.sh`
6. Set the environment variables for BRIAR: `./briar_env.sh`
7. Install the python files: `python setup.py install`

Developing a BRIAR Service that communicates with the API

For instructions on how to develop BRIAR services that host your algorithms with the BRIAR API, please see this document.

Using the Command Line Interface (Python Client)

The BRIAR CLI is implemented in python, and utilizes the Protoc-compiled python stubs to communicate with performer algorithms. This is the method of sending commands to the service. As shown in the overview figure of the BRIAR API, you can see this is done in two ways - through the Command Line Interface (CLI), or through the python client (which the CLI uses).

To use the CLI, In a terminal, run `python -m briar`. You will be provided with a selection of commands:

```
Commands:
    status - Connects to the server and displays version and status information.
    detect - Only run detection.
    extract - Run feature extraction.
    enroll - Extract and enroll biometrics in a database.
    search - Search database for media and IDs that contain biometric matches to the query.
    finalize - Finalize a database.
    sigset-stats - Convert a sigset to a csv file.
    sigset-enroll - Enroll a sigset in a database.
```

Run `python -m briar status` to check that you can establish a connection to the service. If you can, you will see a message similar to the following:

```
==================== STATUS ====================
  Developer:   Oak Ridge National Laboratory
  Name:        Briar.Exmaple
  Alg Version: 0.1.0
  API Version: 0.0.0
  Status:      READY
================================================
```

Note that the other commands representing BRIAR tasks have been intentionally left unimplemented.

This command line functionality is processed through the briar_client, which uses python to compose and send gRPC messages to services at specified network locations (default is localhost:50051 and can be specified with -p). If you wish to extend this functionality, the client can be used like any python module or else can be used as an example for creating your own in whatever language you choose.

Usage

Before continuing, startup a BRIAR service implementation, otherwise the examples below will not work. If you do not have a BRIAR service implementation, please refer to `The BRIAR API Example`. The Briar API example is a basic service which implements the BRIAR SDK and runs face and body detection and identification algorithms using the BRIAR API as a front end.

In a different terminal, run `python -m briar`. You will be provided with a selection of commands:

```
Commands:
    status - Connects to the server and displays version and status information.
    detect - Only run detection.
    extract - Run feature extraction.
    enroll - Extract and enroll biometrics in a database.
    search - Search database for media and IDs that contain biometric matches to the query.
    finalize - Finalize a database.
    sigset-stats - Convert a sigset to a csv file.
    sigset-enroll - Enroll a sigset in a database.
```

Run `python -m briar status` to check that you can establish a connection to the service. If you can, you will see a message similar to the following:

```
==================== STATUS ====================
  Developer:   Oak Ridge National Laboratory
```

```
  Name:        Briar.Exmaple
  Alg Version: 0.1.0
  API Version: 0.0.0
  Status:      READY
================================================
```

To run detections, run `python -m detect -v [image_filename]`, replacing [image_filename] with the path to an image file. You should see something like this:

```
Scanning for videos and images...
    Found 1 images.
    Found 0 videos.
Running Detect on 1 Images, 0 Videos
Detected 3 detections in 4.407133102416992e-06 seconds
Detected test_image.jpg in 4.407133102416992e-05s
Finished detect.
Finished 1 files in 4.552372455596924 seconds
```

If you need any help with any of the commands, each one is fully documented with help text which can be accessed with the '–help' flag. The help text can also be viewed here.

The Briar API Example service can be run on any network enabled computer and can be accessed with the BRIAR command line tools over the network by specifying the IP and port as an argument like so `-p 192.168.1.↩ 100:50051`. Leaving the argument blank defaults to using localhost and port 50051.

Run multiple copies of the BRIAR API Example on a compute resource. Make sure to use a different port. `$ screen python ./briar-example/service.py -w 1 -g 5 --port=127.0.0.1:50063`

1.0.1 Sample Commands with the BRIAR API

1. Start the docker service in one line (example from the BRIAR API exmaple algorithm; each service may vary):
   - `docker run -e NVIDIA_DRIVER_CAPABILITIES=all --gpus all --shm-size=4g --net host -v storage:/briar/briar_storage --rm -it briar-example python ./briar-example/service.py -p 127.0.0.1:50051 -w 2 -g 0 --max-message-siz`
     - Change `-p` to match the IP address set by the BRIAR Example service. `--max-message-size=-1` denotes no maximum message size, which is useful for large images.
2. Detection on a video
   - `python -m briar detect -p 127.0.0.1:50051 <path_to_media>`
3. Enrolling an image
   - `python -m briar enroll -p 127.0.0.1:50051 --max-message-size=-1 --name = <subject-name> --entry = <subject-id|media-id> --entry-type = <subject|media> --database = <database-name> <path-to-media>`
4. Finalize (save) a database to disk
   - `python -m briar finalize -p 127.0.0.1:50051 --database=<database-name>`
5. Search against a database
   - `python -m briar search -p 127.0.0.1:50051 --database=<database-name> <path-to-image>`
6. Whole-body tracking and enroll all detections along a track
   - `python -m briar enroll -p 127.0.0.1:50051 --name=<subject-name> --entry=<numeric-id> --track --database=<database-name> <path-to-video>`

R&D Team Deliverables and Evaluation

Teams will submit software deliverables as docker and singularity container images. Additional documentation can be found in How To Develop a BRIAR Service.

Further Information

The documentation above is intended to provide a high level understanding of the service and client. However, this document does not go into fine detail, particularly pertaining to the service, how it works, and the details of creating your own. For more information, refer to the stubs and protobuf documentation in this repository's `doc` folder.

Chapter 2

How to develop a BRIAR Service

Authors

Name	Organization
Deniz Aykac	Oak Ridge National Laboratory
David Bolme	Oak Ridge National Laboratory
Joel Brogan	Oak Ridge National Laboratory
Ian Shelley	Oak Ridge National Laboratory
Bob Zhang	Oak Ridge National Laboratory

Table of Contents

[[*TOC*]]

Introduction

This document will give you a quick guide on how to take the algorithms you've developed for BRIAR and make them conform and communicate with the BRIAR API. Throughout this document we will use examples from the BRIAR API Example code which is developed in the Python language.

How the gRPC works in a nutshell (And also the BRIAR API)

The BRIAR API is implemented using the `google Remote Procedure Call (gRPC) framework`. The BRIAR API uses this paradigm to interact with performer implementations via a **Client-Server Interface**, where performers' implementations of algorithms comprise a **Server**, while a already implemented API comprises the **Client**. In this way, the BRIAR Client can seamlessly interact with algorithms written in many different languages. To accomplish this, a set of protobuf messages are defined that can be passed between the client and server as requests and responses to different function calls. *All performers must do is implement code for each function that accept input and generate output in the form of these messages.*

gRPC is a proto-language that defines a **service** which contains multiple **functions** that each pass strictly defined **messages** back and forth. **Messages** can be thought of as class definitions, and are defined to have strongly-typed

fields. These **fields** can either be primitive types, or other messages themselves. The files that define the proto structure for BRIAR are located in `proto/briar/briar_grpc`

As a performer, you must implement the functions defined by the BRIAR API protocol:

```
Commands:
    status - Connects to the server and displays version and status information.
    detect - Only run detection.
    extract - Run feature extraction.
    enroll - Extract and enroll biometrics in a database.
    search - Search database for media and IDs that contain biometric matches to the query.
    finalize - Finalize a database.
    sigset-stats - Convert a sigset to a csv file.
    sigset-enroll - Enroll a sigset in a database.
```

To wrap your algorithms in the correct code that can communicate over gRPC to the BRIAR client, we will need the compiled stub files, service files, and message files for your specific language to continue. `Jump to this section` to learn how to run the protoc compiler to automatically generate these stubs.

Compiled protobuf code provides an **abstract service implementation** as a class, filled with **unimplemented functions** in any given language desired by the performer. These abstract classes should be subclassed by performers, with unimplemented functions subsequently implemented to wrap performer code to comply with the specific **BRIAR input and output messages types** provided by the Protoc compiler.

Building Stubs

To install protoc, use the installation instructions provided here: `https://grpc.io/docs/protoc-installation/`

In the root directory of the repository, there is a script to handle stub compilation for you. To build the stubs from the .proto files, run `./build-proto-stubs.sh`

Stubs will be generated in the directory `lib` directory for a variety of popular languages including C++, CSharp, JavaScript, Objective-C, PHP, Python, and Ruby. This stubs will allow developers to produce both clients and services in each of these languages.

Submitting Containers

For evaluation the these containers will be run using `singularity` or `podman` due to data security requirements for ORNL computer systems. Containers will be run in user space and will not have root permissions. Each container will expose a port on the localhost interface that will be used for gRPC communications to the BRIAR API. The container will also be provided with a data directory that will be used for persistent storage across runs.

The container should contain everything needed to run the BRIAR service including software dependencies, trained machine learning models, configuration files, etc. The solution is expected to be automatic and easy to use with appropriate default parameters. On start up the container will boot the software solution, initialize software and models as needed. The software can initialize and load any data from previous runs in the persistent storage. The software will also expose the gRPC service to the external network port 50051 and will be ready to accept client connections.

All R&D teams will be provided with the same testing client as well as supporting files and scripts used by the BRIAR T&E team. Performers should conduct internal testing of their solutions before submission to insure the container operates correctly. Experiments will typically consist three steps:

1. enrolling subject media into a gallery database
2. enrolling field videos and images into a probe database

3. conducting searches and verifications of the probe entries against the gallery entries to generate search results or score matrices

To save computational time teams will need to implement gRPC calls to merge, split, and reorganize the templates in probe and gallery databases to support a variety of tests without re-enrolling the media. Starting in Phase 2, performers may also be expected to implement more specialized template fusion methods.

Most evaluations on the deliverables will treat the software as a black box and will only interface through the gRPC calls. This means that teams should have complete freedom to implement solutions as long as they conform to the public gRPC interface. Creativity is encouraged. Many gRPC function calls and data field are marked as OPTIONAL or REQUIRED. Teams are responsible for implementing REQUIRED items, however some of the OPTIONAL interfaces may transition to REQUIRED as the program and the software matures. The API is designed to be flexible and has many ways to pass additional data structures, options, and parameters to support and encourage teams to implement additional features beyond the core required capabilities.

Software will be submitted to ORNL using containers.

- Containers should include everything need to run and evaluate the software solution including: dependencies, compiled software, ML models, configuration files, etc.
  Source code is not needed and can be delivered separately.
- Software will be delivered in two container types.
  - Docker - is the gold standard for containerization and so this deliverable will allow government agencies an easy method to start up and run BRIAR software.
  - Singularity - is a container solution that addresses security issues with docker.
    Converting containers from docker to singularity is relatively easy. However, instead of running with root permission singularity will require user level permissions.
- Simple commands will be used to start and stop containers. These must respect certain environment variables which tell the container which ports to start up on and which CPU, GPU, and Memory Limits are allocated to the container. The start and stop commands for BRIAR example services are shown in the following sections.
- A data directory will be mounted in the container that can be used for persistent storage. This is primarily to store databases of templates but can be used for other configuration data as well. Data stored in other directories in the container will be wiped between runs.

Additional details to follow

Converting from docker to singularity

The conversion from docker needs to take place on a machine with root permissions.

1. Install singularity using this guide: https://sylabs.io/guides/3.8/user-guide/quick_start.html#quick-installation-steps
2. Singularity can convert the docker container to a singularity container using a command like this: `sudo singularity build briar-example.sif docker-daemon://briar-example:latest`
3. Move the singularity container to the target machine (root not access needed).
4. Run a shell to test the container `singularity shell --nv --no-home --cleanenv -B $BRIAR_DATA_DIR:/briar/briar_storage -H /briar ./briar-example.sif`

Starting the container

Please double check this section before each deliverable. Details on how solutions are started are subject to change through the course of the BRIAR program.

The evaluation environment will define variables that control the hardware resources allocated to each solution. Multiple versions of BRIAR software may also be run in parallel.

```
$ BRIAR_GRPC_PORT=127.0.0.1:50060 # Start the gRPC service on this port
$ MAX_MESSAGE_SIZE=134217728 # 128Mb
$ BRIAR_WORKERS=2 # Number of worker processes
$ BRIAR_GPUS=2,3 # Use only GPU 2 & 3
$ BRIAR_DATA_DIR=/briar/data/team1/experiment2
```

Performer teams should insure that the BRIAR services start and run correctly on both docker and singularity.

Docker

It is expected that government stake holders will use docker or compatible services to run BRIAR software. Here we show a sample command.

```
docker run -e NVIDIA_DRIVER_CAPABILITIES=all --gpus all --shm-size=4g --net host -v
       $BRIAR_DATA_DIR:/briar/briar_storage --rm -it briar-exmaple python ./briar-exmaple/service.py -p
       $BRIAR_GRPC_PORT -w $BRIAR_WORKERS -g $BRIAR_GPUS --max-message-size=$MAX_MESSAGE_SIZE
```

Singularity

The current plan is to run all evaluations in singularity. There may be some issues converting to singularity since this will not run as a root process and permissions may need to be changed.

```
singularity run --nv --no-home --cleanenv -B $BRIAR_DATA_DIR:/briar/briar_storage -H ./briar-example
       ./briar-example.sif python  ./briar-example/service.py -p $BRIAR_GRPC_PORT -w $BRIAR_WORKERS -g
       $BRIAR_GPUS --max-message-size=$MAX_MESSAGE_SIZE
```

Command Line Tools

Once the service is started the command line tool can be used to test the service. These instructions are found in the `BRIAR API Documentation`

Evaluations

The services will be run using default parameters. Services are expected to be completely automatic where video and images are provided and algorithms will auto-configure themselves and perform detection, tracking, extraction, and enrollment operations to ingest the media with no extra information. Evaluations will be conducted using three steps:

1. Build a gallery database using the commands `python -m briar sigset-enroll --entry-type=subject --best ...` or `python -m briar enroll --entry-type=subject ...` commands. This gallery will contain entries organized by a subject ids (G#####). Each entry may come from multiple media files (e.g. photographs from different angles, gait walking videos, etc.) but should contain just one 'foreground' or 'best' person.
2. Build a probe database using the commands `python -m briar sigset-enroll --entry-type=media ...` or `python -m briar enroll --entry-type=media ...`. These probes will contain entries organized by unique media ids. Each entry will come from a single media file where each file may contain multiple people.
3. The final step is to run search and verification commands to produce results that will be analyzed using statistical tools. These commands are `python -m briar test-verify` or `python -m briar test-search`

Version Requirements

This section will contain what performers will be required to implement for each phase once details are solidified.

Quick Reference

Please check This section regularly to make sure deliverables stay up to date with the evaluation environment.

Parameter	Value	Notes
API Version	v1.0.0	
Default gRPC Port	127.0.0.1:50051	
Evaluation Port Range	50050 - 50150	All deliverables should be able to run on these ports.
Evaluation CPU Count	8	Intel or AMD
Evaluation Memory Limit	32GB	
GPU Count	2	NVIDIA A100
GPU Memory	At least 8GB	Amount of memory per GPU
Singularity Version	##	
CUDA Version	11.4.1	CUDA version on the host machine
Enrollment Speed	5x Realtime	
Verification Speed	TBD	
Search Speed	TBD	
Template Size	$<$ 1Mb	

BRIAR API Versions

This section will contain notes on how the BRIAR API has been developed and how it will be changing in the future. Please check back regularly for updates to make sure any solution delivered stays in compliance.

2021/09/07 v1.0.0 - Initial BRIAR API released.