Example BRIAR Python Service

Generated by Doxygen 1.8.17

Chapter 1

BRIAR API

Introduction

This document is intended to describe how to integrate a performer's implementation with the necessary code to allow it to communicate to the Briar API.

The Briar API is designed to create a unified method of interfacing with different algorithms created across different teams of performers, with the gRPC communication layer acting as a cross-platform, cross-language API to aid the design of algorithms and the services which implement them. The client offers a number of functions, such as detect, extract, enroll, enhance, etc. which allow users to construct requests to run said functions on a gRPC service which may run both locally or on a network-connected system.

The face/body detection/identification algorithms can be written in any language supported by gRPC `https://grpc.io/docs/languages/` using BRIARService in 'briar_service.proto'. This 'proto file' is compiled into stubs for other languages, allowing abstract implementation of services to be written in said languages and hook into the client using gRPC messages. An example of one of these services is a python implementation in 'service.py' which illustrates a very simple, working example of a BRIAR service, whose workings can be extrapolated to whatever language implementation which is desired. As this an API and not an implementation of an algorithm, the provided example python service does not implement the detect, extract, etc. functions but does provide a lightly documented example to aid performers in creating their own services to run their algorithms.

If the performer wants a template for a python service then they can create a service class which inherits from BRIARService() in service.py, or they can use the class BRIARServiceServicer in srvc_pb2_grpc.py (a compiled proto-file) if a blank slate is desired.

Using the API and Creating your algorithms

The gRPC stubs and server/client implementation makes BRIAR unique as an API implementation and may not be intuitive at first glance, hence the need of this section.

Supported languages

Because the API uses gRPC as an interface between the client which implements the command line and some other base functions and the service which shall implement your code, your service can be implemented in whatever languages which are supported by gRPC. Here is a link to the supported languages `https://grpc.io/docs/languages/`

While python is used extensively in the documentation to illustrate functionality, any language supported by gRPC can be used in its place.

Protobuf files

Before creating your own implementations, it will be helpful to understand how gRPC is being leveraged for cross-platform, cross-language communication between the BRIAR API and projects which are derived from it.

Protobuf files (found Here) are files ending in '.proto' which outline the functions accessible to the API. They are easily modifiable, cross language files formatted similarly to C++ which define gRPC objects and may be compiled into gRPC stubs with the script `build-proto-stubs.sh`. The compilation creates stubs, which are importable source files, in a number of languages, allowing programs written in said languages to be called from BRIAR clients. Effectively, the gRPC layer can be thought of a cross-language API, allowing any of the languages supported by gRPC to be used as part of the larger BRIAR ecosystem, so long as the messages passed between client and service are the ones defined within the BRIAR protobuf files.

An important section of the proto files, the BRIARService, is shown in part below.

```
service BRIARService{
    rpc status(StatusRequest) returns (StatusReply){};
    rpc detect(stream DetectRequest) returns (stream DetectReply){};
    rpc extract(stream ExtractRequest) returns (stream ExtractReply){};
    ... more definitions
}
```

This is where the api functions (technically service methods) are defined. `rpc function_name(request↩ _type)` defines the function, and what kind of request it expects, and `returns(reply_type)` defines what kind of reply is expected. These request and reply messages, defined within `briar_service.proto`, define the data that is passed to and returned by the service's functions. When compiled, two abstract service classes are created: `BRIARServiceStub` for the client and `BRIARServiceServicer` for the service.

`BRIARServiceServicer` should be inherited by your service class so you can write your own method implementations for it. When initialized, it will listen on the specified port for connections from clients and run the methods defined in the protobuf as it gets requests

`BRIARServiceStub`, referred to as 'stub', mirrors the methods of the service. I.e. the stub has stub.detect, stub.status, etc. which, when called, expect an appropriate request message such as a 'DetectRequest' or 'Status↩ Request' which it will send to the connected service, and will return or yield the reply type defined in the 'returns' field in the protobuf definition.

Protobuf Files and gRPC - a Python Example

It is important to lead this section saying that the Briar API supports every language supported by gRPC and the python implementation of service.py and this section's examples do not limit the actual implementation of your programs to python.

Within the existing BRIAR code, the gRPC stubs are used to define the format and contents of the messages being passed between BRIAR clients and services. These gRPC messages can be conceptually divided into three types - requests, replies, and objects. Requests are sent by clients to services to invoke functions on the service side, replies are sent from services to clients to ferry back the results from the invoked method, and the objects are classes/structures suction as "Detection" or "BriarRect" which are used to hold data in a meaningful way as it is passed one way or the other. Which functions in the service accept what methods is defined within the BRIAR↩ Service service in the briar_service.proto file

For an example of how a client invokes a request and gets a reply, look at the status function in briar_client. Within the function, there are these lines:

```
# Establishing a connection to the service
self.channel = grpc.insecure_channel(port,options=channel_options)
# Creating the stub
self.stub = srvc_pb2_grpc.BRIARServiceStub(self.channel)
```

and

```
# Getting the service's status
reply = self.stub.status(srvc_pb2.StatusRequest())
print(reply.developer_name, reply.service_name, reply.version, reply.status)
```

As stated before, BRIARService in the briar_service.proto defines the service, and BRIARServiceStub contains 'network calls' to methods on the service side. The line with `self.stub.status(srvc_pb2.Status↩ Request())` initializes a status request message and passes it to the stub which then calls the network code and passes the message on to the connected service. The service will receive the status request and, because `self.stub.status` was called, the service will call its 'status' function when it handles the incoming network message. Below is the code where that happens

```
def status(self, request, context):
    return srvc_pb2.StatusReply(developer_name="[devname-here]",
                                service_name="This is a service",
                                version=briar_pb2.APIVersion(major=1,
                                                             minor=2,
                                                             patch=3),
                                status=briar_pb2.BriarServiceStatus.READY)
```

The service code above constructs a StatusReply message. The returned values will pass back through the gRPC backend and be sent over the network back to the client and be put into the variable 'reply'. From there, it can be accessed like any other class.

```
print(reply.developer_name, reply.service_name, reply.version, reply.status)
```

The official documentation for how services implement requests, methods, and replies can be found here `https://grpc.io/docs/what-is-grpc/core-concepts/`

Creating a Python Service Example

This section will illustrate how to implement your own algorithms with the API using python, but the steps here can be applied to other languages as well. For convenience, let's assume you are working on a project named **BARB** which uses the BRIAR API.

You will want to add your algorithms in a service file which will inherit from the BRIARServiceServicer. Below are the steps to take.

1. Create the file barb_service.py and add the imports
```
from concurrent import futures
import grpc
import time
from briar import media_converters, Rect
from briar.functions import new_uuid
from briar.service import BRIARService
from briar.briar_grpc.briar_pb2 import Attribute, BriarDurations, BriarRect, Detection
from briar.briar_grpc.briar_service_pb2 import DetectReply
from briar.briar_grpc.briar_service_pb2_grpc import add_BRIARServiceServicer_to_server
```
2. Create a new class which inherits from BRIARService
```
class BARBService(BRIARService):
    def __init__(self, options=None, database_path="databases"):
        super(BARBService, self).__init__()
```
3. Implement one or more of the methods defined in BRIARService. 'detect' will be used as an example
```
def detect(self, request_iter, context):
    # Iterate over requests as the client sends them
    for detect_request in request_iter:
        t0 = time.time()
        # break out the request's attributes for clarity
        protobuf_media = detect_request.media
        frame_num = detect_request.frame
        subject_id = detect_request.subject_id
        subject_name = detect_request.subject_name
        detect_options = detect_request.detect_options
        # Convert the protobuf byte vector back into a numpy array
        numpy_img = media_converters.image_proto2cv(protobuf_media)
        # run the detection algorithm
        roi, det_class, score = self.detect_worker(numpy_img)
        # populate the gRPC detection class
        t1 = time.time()
        loc = BriarRect(x=roi.x, y=roi.y, width=roi.width, height=roi.height)
        detection = Detection(confidence=score, location=loc, frame=frame_num, detection_id=1,
                              detection_class=det_class)
        t2 = time.time()
        # Create an arbitrary attribute to use
```

```
            attrib1 = Attribute(key="Attribute1",
                                text="AttributeText")
            attrib2 = Attribute(key="Attribute2",
                                fvalue=0.25)
            attrib3 = Attribute(key="Attribute3",
                                buffer="ByteArray".encode("utf-16"))
            # python GRPC doesn't like direct assignment of iterables. Use CopyFrom instead.
            detection.attributes.MergeFrom([attrib1, attrib2, attrib3])
            detect_reply = DetectReply()
            detect_reply.detections.append(detection)
            detect_reply.frame_id = frame_num
            # Briar has a 'durations' grpc_object to easily store and return timing metrics
            # durs = BriarDurations()
            detect_reply.durations.durations["detection_time"] = (t2-t1)*1e6 # Durations are tracked in
     microseconds
            detect_reply.durations.total_duration = (time.time()-t0)*1e6
            print("Yielding")
            yield detect_reply
    def detect_worker(self, numpy_img):
        # ... do detection stuff
        roi = Rect(5, 5, 25, 25)
        det_class = "FACE"
        score = 0.99
        return roi, det_class, score
```

4. Add __main__

```
    if __name__ == "__main__":
        server = grpc.server(futures.ThreadPoolExecutor(max_workers=10),
                             options=[('grpc.max_send_message_length', 20000000),
                                      ('grpc.max_receive_message_length', 20000000)])
        add_BRIARServiceServicer_to_server(BARBService(), server)
        server.add_insecure_port("0.0.0.0:50051")
        server.start()
        # server.wait_for_termination()
        print("Service Started.  ")
        while True:
            time.sleep(0.1)
```

5. Run `python barb_service.py`
   (a) You should see `Service Started.`
6. In another terminal, run `python -m briar detect /some/path/to/an/image/file.jpg`
   (a) The client will automatically read the image file and send a detect request to the example BARB service (defaults to connecting to 127.0.0.1:50051 if no connection parameters given)
   (b) The service will call self.detect, enter into the main iterator, and iterate once (if you gave a single file) and yield a single detect reply which will return to the client, which has its own iterator.
   (c) You can provide multiple image files with the command line
      - `python -m briar detect img1.jpg img2.jpg img_dir`
7. A stacktrace related to gRPC will likely occur during integration of algorithms with the BRIAR-API. Because gRPC is a message passing language, an error in a performer's implementation will result in a error propagated through the BRIAR client. Error messages from gRPC can be opaque as shown by the following example:

```
  File "/home/ii1/anaconda3/envs/insightface/lib/python3.8/site-packages/grpc/_channel.py", line 426, in
      __next__
    return self._next()
  File "/home/ii1/anaconda3/envs/insightface/lib/python3.8/site-packages/grpc/_channel.py", line 809, in
      _next
    raise self
grpc._channel._MultiThreadedRendezvous: <_MultiThreadedRendezvous of RPC that terminated with:
        status = StatusCode.UNKNOWN
        details = "Exception iterating requests!"
        debug_error_string = "None"
```

1. Seeing this on the client end most likely means there was an issue with the service implementation, and server side debugging may be required. You will see this message anytime the implemented service fails to return a valid reply message.

Chapter 2

Namespace Index

2.1 Packages

Here are the packages with brief descriptions (if available):

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 4

Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

Chapter 6

Namespace Documentation

6.1 briar Namespace Reference

Namespaces

- service

6.2 briar.service Namespace Reference

Classes

- class BRIARService

  This service, when run, starts a server which awaits connections and messages from briar clients, running methods defined in briar_service.proto in "BRIARService".

Functions

- def serve ()

  Initialize and run the BRIARService.

6.2.1 Detailed Description

Systems based off the BRIAR API will come in two parts: the BRIAR client (which shouldn't need to be extended and is the api itself since it is the 'hooks' which connect into gRPC) and the Service which runs wherever you want the image processing to be performed. Any machine learning, neural networks, image processing, etc should take place within performer designed services and this file, containing the BRIARService, exists as a basic framework for creating these other services and does nothing significant on is own.

The Service itself runs as a server which can accept gRPC calls on the specified port. These calls are defined within briar_service.proto under in the "BRIARService" service. Each line will look like:

- rpc status(StatusRequest) returns (StatusReply){};

or

- rpc detect(stream DetectRequest) returns (stream DetectReply){}

where the name after 'rpc' will define which method to call and 'stream' prefixing a request will define said request to be an iterable on client side, server side, or both. Whenever the service gets receives a request matching one which is defined, it will invoke the associated method defined within the class, (i.e. the line "rpc detect(...." will cause BRIARService.detect to run with the Detect Request acting as the argument.

In the case of a stream, the client is expected to yield when "stream" prefixes the reply, and the server is expected to yield when "stream" prefixes the request within the rpc decleration. For example, when the server is streaming, on the server side the code will need to yield replies, and on client-side you will put the service_stub.method_name in a loop.

- for reply in service_stub.method_name(DetectRequest):

When the client is streaming the client will yield requests and on the server-side you will put the request iterator in a loop.

- for request in request_iterator:

In case both are streaming, then both will yield, and both will iterate in a ping-pong fashion, yielding back and forth to each-other.

6.2.2 Function Documentation

6.2.2.1 serve()

```
def briar.service.serve ( )
```

Initialize and run the BRIARService.

Runs until killed

Returns

:

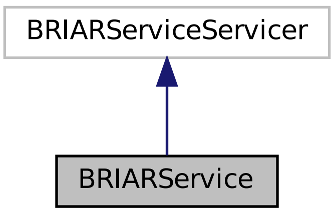
Chapter 7

Class Documentation

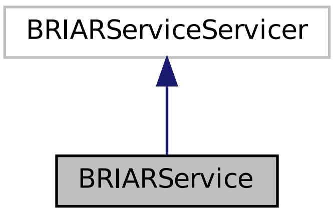
7.1 BRIARService Class Reference

This service, when run, starts a server which awaits connections and messages from briar clients, running methods defined in briar_service.proto in "BRIARService".

Inheritance diagram for BRIARService:



Collaboration diagram for BRIARService:



Public Member Functions

- def __init__ (self, options=None, database_path="databases")
- def cluster (self, request, context)

  Takes a set of templates and clusters them, matching according to subject similarity.
- def database_create (self, request, context)

  Create a new database and populate it with templates.
- def database_finalize (self, request, context)

  : Finalizes the database and saves it to the disk
- def database_insert (self, request, context)

  Inserts the templates contained in the request into a specified database.
- def database_list_templates (self, request, context)

  Produces a list of templates contained in the specified database.
- def database_load (self, request, context)

  Load the database specified in the load request.
- def database_names (self, request, context)

  Produces a list of database names contained in the specified database.
- def database_remove_templates (self, request, context)

  Takes the ids in 'request' and removes them from the database.
- def database_retrieve (self, request, context)

  Retrieves the templates contained in the database matching the provided names.
- def detect (self, req_iter, context)

  Streams image data in the form of a extract request iterator.
- def enhance (self, request, context)

  Run an enhancement on a provided image.
- def enroll (self, req_iter, context)

  Streams images or templates in the form of an enroll request iterator.
- def extract (self, req_iter, context)

  Streams image data in the form of a extract request iterator.
- def get_api_version (self, request, context)
- def search (self, request, context)

  Search database for templates matching the provided probe.
- def status (self, request, context)
- def verify (self, request, context)

  : Calculate how similar sets of templates are

Public Attributes

- options
- predictor_path
- rec_model_path

Static Public Attributes

- string DEFAULT_PREDICTOR_NAME = "predictor_model.dat"
- string DEFAULT_REC_MODEL_NAME = "recognition_model.dat"
- string DEFAULT_WEIGHTS_DIR = "weights"

7.1.1 Detailed Description

This service, when run, starts a server which awaits connections and messages from briar clients, running methods defined in briar_service.proto in "BRIARService".

The methods which are invoked by incoming grpc messages are not implemented and are present to act as a framework to assist performers developing their own services.

7.1.2 Constructor & Destructor Documentation

7.1.2.1 __init__()

```
def __init__ (
              self,
              options = None,
              database_path = "databases" )
```

Parameters

options	optparse.Values: Command line options to control the service
database_path	str: Path to where to save generated data

7.1.3 Member Function Documentation

7.1.3.1 cluster()

```
def cluster (
              self,
              request,
              context )
```

Takes a set of templates and clusters them, matching according to subject similarity.

Parameters

request	briar_service_pb2.ClusterRequest: Templates name to cluster
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.ClusterReply

7.1.3.2 database_create()

```
def database_create (
              self,
              request,
              context )
```

Create a new database and populate it with templates.

Parameters

request	briar_service_pb2.DatabaseCreateRequest: Request with database name and optionally templates
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.DatabaseCreateReply

7.1.3.3 database_finalize()

```
def database_finalize (
              self,
              request,
              context )
```

: Finalizes the database and saves it to the disk

Parameters

request	briar_service_pb2.DatabaseFinalizeRequest: Request.

Returns

: briar_service_pb2.DatabaseFinalizeReply

7.1.3.4 database_insert()

```
def database_insert (
              self,
              request,
              context )
```

Inserts the templates contained in the request into a specified database.

Parameters

request	briar_service_pb2.DatabaseInsertRequest: Request containing database name and templates to insert
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.DatabaseInsertReply

7.1.3.5 database_list_templates()

```
def database_list_templates (
              self,
              request,
              context )
```

Produces a list of templates contained in the specified database.

- request: briar_service_pb2.DatabaseListRequest

Parameters

request	Request containing database name to list
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.DatabaseListReply

7.1.3.6 database_load()

```
def database_load (
              self,
              request,
              context )
```

Load the database specified in the load request.

Parameters

request	briar_service_pb2.DatabaseLoadRequest: Request containing database name to load
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

@reply: briar_service_pb2.DatabaseLoadReply

7.1.3.7 database_names()

```
def database_names (
              self,
              request,
              context )
```

Produces a list of database names contained in the specified database.

Parameters

request	briar_service_pb2.DatabaseNamesRequest: Request. No additional options
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.DatabaseNamesReply

7.1.3.8 database_remove_templates()

```
def database_remove_templates (
              self,
              request,
              context )
```

Takes the ids in 'request' and removes them from the database.

Parameters

request	briar_service_pb2.DatabaseRemoveTmplsRequest: Request containing database name and template ids to remove
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.DatabaseRemoveTmplsReply

7.1.3.9 database_retrieve()

```
def database_retrieve (
              self,
              request,
              context )
```

Retrieves the templates contained in the database matching the provided names.

Parameters

request	briar_service_pb2.DatabaseRetrieveRequest: Request containing the database name to pull templates from
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.DatabaseRetrieveReply

7.1.3.10 detect()

```
def detect (
              self,
              req_iter,
              context )
```

Streams image data in the form of a extract request iterator.

Takes images, detects contents, and creates detections using provided detections.

Parameters

req_iter	Generator(briar_service_pb2.DetectRequest): Detections happen as a part of an iteration part of a for loop. Will yield a DetectRequest containing a detect and other info.
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: Each iteration should yield a DetectReply

7.1.3.11 enhance()

```
def enhance (
              self,
              request,
              context )
```

Run an enhancement on a provided image.

Parameters

request	briar_service_pb2.EnhanceRequest: Contains image(s), enhancement options, and type of enhancement to run
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.EnhanceReply

7.1.3.12 enroll()

```
def enroll (
              self,
              req_iter,
              context )
```

Streams images or templates in the form of an enroll request iterator.

Takes images, optionally detects, and extracts them to create templates using provided detections, auto detection, or the full image as specified by the extract flag. Enrolls templates into the database

Parameters

req_iter	Generator(briar_service_pb2.EnrollRequest): Enrolls (and optional detects and extracts) happen as a part of an iteration request_iter will need to be part of a for loop. Will yield a EnrollRequest containing info resulting from enrolls.
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: Each iteration should yield a EnrollReply

7.1.3.13 extract()

```
def extract (
              self,
              req_iter,
              context )
```

Streams image data in the form of a extract request iterator.

Takes images, extracts faces, and creates templates using provided detections, auto detection, or the full image as specified by the extract flag. Returns templates representing faces in the media

Parameters

req_iter	Generator(briar_service_pb2.ExtractRequest): Extracts happen as a part of an iteration part of a for loop. Will yield a ExtractRequest containing templates and other info resulting from extractions
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: Each iteration yields a ExtractReply

7.1.3.14 get_api_version()

```
def get_api_version (
              self,
              request,
              context )
```

7.1.3.15 search()

```
def search (
              self,
              request,
              context )
```

Search database for templates matching the provided probe.

Parameters

request	briar_service_pb2.SearchRequest: Request to search containing template and name of database to search
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.SearchReply

7.1.3.16 status()

```
def status (
              self,
              request,
              context )
```

Parameters

request	briar_service_pb2.StatusRequest: Request containing options for the get status request
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service_pb2.StatusReply

7.1.3.17 verify()

```
def verify (
              self,
              request,
              context )
```

: Calculate how similar sets of templates are

Parameters

request	briar_service_pb2.VerifyRequest: Request to verify containing templates/media to compare
context	grpc.ServicerContext: object that provides RPC-specific information such as timeout limits.

Returns

: briar_service.VerifyReply

7.1.4 Member Data Documentation

7.1.4.1 DEFAULT_PREDICTOR_NAME

```
string DEFAULT_PREDICTOR_NAME = "predictor_model.dat"  [static]
```

7.1.4.2 DEFAULT_REC_MODEL_NAME

```
string DEFAULT_REC_MODEL_NAME = "recognition_model.dat"  [static]
```

7.1.4.3 DEFAULT_WEIGHTS_DIR

```
string DEFAULT_WEIGHTS_DIR = "weights"  [static]
```

7.1.4.4 options

```
options
```

7.1.4.5 predictor_path

```
predictor_path
```

7.1.4.6 rec_model_path

```
rec_model_path
```

The documentation for this class was generated from the following file:

- briar-api/lib/python/briar/service.py

Chapter 8

File Documentation

8.1 briar-api/doc/readme-python-service.md File Reference

8.2 briar-api/lib/python/briar/service.py File Reference

Classes

- class BRIARService

  This service, when run, starts a server which awaits connections and messages from briar clients, running methods defined in briar_service.proto in "BRIARService".

Namespaces

- briar.service

Functions

- def serve ()

  Initialize and run the BRIARService.

Index