



ARL-TR-9329 • OCT 2021



Shareable Components Architecture Design

by Chris Schlesiger

Approved for public release: distribution unlimited.

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.



Shareable Components Architecture Design

Chris Schlesiger

*Computational and Information Sciences Directorate,
DEVCOM Army Research Laboratory*

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.					
1. REPORT DATE (DD-MM-YYYY) October 2021		2. REPORT TYPE Technical Report		3. DATES COVERED (From - To) March 2021–September 2021	
4. TITLE AND SUBTITLE Shareable Components Architecture Design				5a. CONTRACT NUMBER	
				5b. GRANT NUMBER	
				5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR(S) Chris Schlesiger				5d. PROJECT NUMBER	
				5e. TASK NUMBER	
				5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) DEVCOM Army Research Laboratory ATTN: FCDD-RLC-IB Adelphi, MD 20783-1138				8. PERFORMING ORGANIZATION REPORT NUMBER ARL-TR-9329	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)	
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release: distribution unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Shareable Components are discrete pieces of language technology software that can be treated as distinct entities that perform a singular task. Taken together, or even considered on their own, these Shareable Components can be assembled in novel ways to create workflows to perform a function in an application. This report describes the underlying architecture of the Shareable Components system, how the services have been created, and how one can create new components and add additional foreign languages to the existing components.					
15. SUBJECT TERMS software, Android, foreign language, speech, translation, recognition, synthesis, programming, design					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 82	19a. NAME OF RESPONSIBLE PERSON Chris Schlesiger
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified			19b. TELEPHONE NUMBER (include area code) (301) 394-2473

Contents

List of Figures	v
1. Introduction	1
2. Service-Oriented Architecture	2
3. Using a Shareable Component Service	3
4. Client–Server Communication	4
4.1 Actions	5
4.2 Configuration Classes	5
4.3 Parameters	5
4.4 Status	5
5. Creating a Service	6
5.1 ServiceBase	6
5.2 MessageHandlerBase	7
5.3 Managing Configurations	7
5.3.1 ASR Configuration Sample	8
5.3.2 MT Configuration Sample	8
5.3.3 TTS Configuration Sample	11
5.4 Anatomy of an <i>Execute</i> Request	12
6. Installing a Service	16
7. Shareable ASR Kaldi Service	16
8. Shareable MT OpenNMT TF-Lite Service	17
9. Shareable TTS FastSpeech2 MultiBand MelGan TF-Lite Service	18
10. Creating a Client	19

11. ULTRA Application Client	20
11.1 Configuration Files	21
11.2 Workflow	25
11.3 Translation History	26
12. Conclusion	28
13. References	29
Appendix. Libraries of Application Programming Interfaces (APIs)	30
List of Symbols, Abbreviations, and Acronyms	74
Distribution List	75

List of Figures

Fig. 1	Shareable Component service elements.....	2
Fig. 2	Client–server bindings are created in both directions.....	3
Fig. 3	UML class diagram of base classes in use.....	6
Fig. 4	Flow chart depicting the processing of an <i>execute</i> request.....	15
Fig. 5	ULTRA client uses Shareable Components services.....	20
Fig. 6	ULTRA client workflow for speech-to-speech translation.....	21

1. Introduction

Shareable Components are discrete pieces of language technology software that can be treated as distinct entities that perform a singular task. Taken together, or even considered on their own, these shareable components can be assembled in novel ways to create workflows to perform a function in an application.

The system described in this report has been designed to operate on small portable devices running the Android operating system. These devices have become ubiquitous in today's society, found in the hands and pockets of most people throughout the world. Their lightweight footprint makes them ideal for field operations that do not encumber the Soldier, and they have become increasingly powerful as newer models are released, capable of running sophisticated, computationally complex operations.

The Android operating system runs on nearly 70% of all small portable devices globally. Applications created for Android generally use the Java programming language, which is a mature and well-known programming language with the support of a large community of open-source products. Many open-source projects can be found in the area of language technology including automatic speech recognition (ASR), machine translation (MT), and text-to-speech (TTS).

It is these three technologies that have been assembled into the Shareable Components system on the Android operating system. Chaining them together into a workflow has led to the creation of a Speech-to-Speech application called the Universal Language Translator (ULTRA). With this application, a Soldier can speak into the device in English, where it is recorded, recognized into text, translated into a foreign language, and then synthesized into speech that is played out loud. A foreign national interviewed by the Soldier can also use the device to answer questions in his native tongue and have that device translate it back to English for the Soldier to understand.

The Shareable Components system is designed to allow for new language technologies to be created and assembled into new and novel workflows to provide new capabilities and applications. They are also designed to easily allow new language support to be created and added to the existing Shareable Components.

This document describes the underlying architecture of the Shareable Components system, how the services have been created, and how one can create new components and add additional foreign languages to the existing components.

2. Service-Oriented Architecture

Each of the Shareable Components has been wrapped inside an Android service. As a service, it is available for binding by any number of different client applications. These clients call the service to perform a discrete unit of work and await asynchronously for its completion. The services run asynchronously because these language technologies are long and computationally intense. It can take several seconds to perform an ASR, MT, or TTS operation, which is significantly long in computer terms. The advantage to having these wrapped in discrete services is that they can run in the background on their own thread until completion while the client offers feedback to the user or allows that user to perform other operations while waiting.

In addition to being a service, the Shareable Component is also broken down into an adapter and Language Set(s), as depicted in Fig. 1.

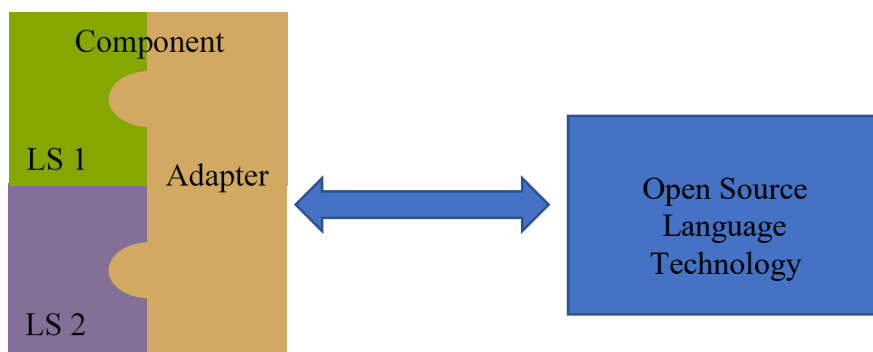


Fig. 1 Shareable Component service elements

The Adapter is the code necessary to run the language technology the component offers and is ideally completely agnostic as to which foreign language(s) is being considered. The intent of the adapter code is to launch and interface with a given piece of typically open-source language technology while also providing a consistent application programming interface (API) for the service that is understood by the overall system.

A Language Set (LS) is a set of pretrained model and configuration files that specify a foreign-language capability for that component. Multiple LSs can be installed on a system with the component to increase its capability. For example, in the ASR Kaldi Shareable Component, LS1 is French and LS2 is Arabic. They both use the same adapter code that connects to the Kaldi Open Source Toolkit.

This service approach and API design can be extended to other service-oriented architectures, such as a REST-API running on larger backend servers, should the system be extended further. (The Appendix contains libraries of APIs.)

3. Using a Shareable Component Service

On an Android system, a client application binds to a service when it wants to use it. This binding opens a channel of communication from the client to the service. Shareable Components services run asynchronously; so, when a service has completed its work, it needs to communicate back to the client (Fig. 2). Thus, another binding operation is performed from the service to the client. It is through this channel that the result of the service's work is returned when it is completed.

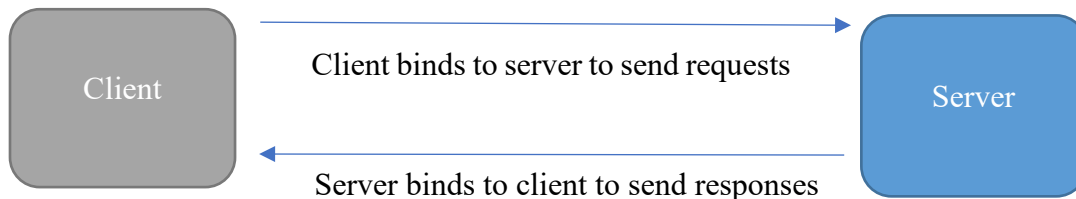


Fig. 2 Client–server bindings are created in both directions

When a Shareable Component service is initially bound by a client, it first searches for installed LSs. These sets are in subfolders in a predefined folder located on the Android device that is specific to that service. The service locates these sets, reads their configurations (stored as YAML files), and checks that all the required model files are present. These LS configurations are kept in a list in the service and last for the lifetime of the service. When the last client application bound to the service exits, the service also exits, clearing this list, so the service is ready to be launched again to start the process all over.

With this design, it becomes simple to add new LSs to the service. When all clients to the service are closed and the service is not running, a new LS can simply be copied into a new folder in the correct location and it will be available to the service the next time it is launched. This also allows for simple modification to existing LSs, if necessary, in the same manner.

The client application can initiate a call to the service to return this list of available language-set configurations. Thus, the client needs no a priori knowledge of languages unless there is some other design requirement for it to do so. It can simply query the service to get what foreign languages are supported. This can change from system to system depending on what foreign LSs are installed on each of those systems.

Each language-set configuration contains information that the client needs to launch a unit of work with the service as well as information that can be displayed to the user in readable format. A client could potentially populate drop-down selections (or other means of selection) using this list that the user can choose from to indicate what language to process.

Included in the configuration information is a universally unique identifier (UUID) for that foreign-language set. This UUID must be sent to the service along with any other required parameters particular to that Shareable Component service and the desired input data when requesting a unit of work.

When the service receives an *execute* request to perform a unit of work from a client, it retrieves the unique identifier given to it, any parameters required, and the input data and starts processing. It finds the matching LS, sends the model files to the adapter, and executes the work and waits for it to finish.

On successful completion it sends the output back to the client along with additional metadata information that describes the output and its format. A status code is also returned to notify the client how that processing finished.

On an error, the service sends an appropriate status code to the client along with appropriate error messages that can be displayed to the user if desired.

4. Client–Server Communication

As stated in the previous section, communication between the client application and a Shareable Component server is through a two-channel binding using Messengers. This type of communication is described in Android documentation.¹

The Message class that is transmitted between the client and server via the bound messenger contains a bundle of key-value pairs of data, typically stored as a JavaScript Object Notation (JSON) object, and an integer value that defines what type of action it is. The Shareable Component architecture uses a set of predefined keys that both sides of the client–server relationship understand as data parameters and actions definitions. Together, these defined terms contain instructions to the server and describe the data that is being exchanged.

To make it easier to standardize these keys, a Java library called *shareable-configuration* was created that both the client and server can reference. This library contains many useful classes, as described in the following sections:

4.1 Actions

This class contains static property definitions defining the action identifiers for the four types messages that exist in the system:

- **Configuration request**—a request from the client to the server to return all language configurations it supports
- **Configuration reply**—the reply from the server to the client with all configurations contained in a list
- **Execution request**—a request from the client to the server to execute its main function on a set of data
- **Execution reply**—the reply with the result of the execution request including all appropriate metadata

4.2 Configuration Classes

There are a number of configuration classes in the library that are specific to ASR, MT, and TTS. These classes are designed to be serialized and deserialized as YAML configuration files. This can be accomplished through the use of a third-party YAML library such as SnakeYAML.

The configuration classes contain all of the information necessary to describe a language capability of the service. The service can simply open the configuration file, deserialize it into one of these classes, assemble them into a list with the other languages, and then return that list when a client requests it.

There are a number of generic base configuration and language classes in the library that are designed to standardize common properties across similar configuration types.

4.3 Parameters

This class in the library contains static property definitions that describe the data and metadata to be exchanged. This data contains definitions that are general for all three current types of servers (ASR, MT, and TTS) and some that are specific to each. These parameters can be extended if new types of services are added to the system.

4.4 Status

An enumeration of status codes that are sent back with replies to indicate whether the execution succeeded or not:

- **Success**—execution completed normally and all results are returned in the message
- **Error**—execution failed and error messages are contained in the message
- **Warning**—execution completed, but there were some problems; both the resulting data and warning messages are included in the message

5. Creating a Service

The Java library *shareable-android-base-classes* was created to assist with the creation of Shareable Component services. It contains two abstract base classes that can be inherited that provide common methods to perform general functions of the service and to provide a framework for methods that can be overridden. The two classes are designed to work together; if one class is used, they should both be used.

Each of the three existing Shareable Components libraries inherit from these base classes: ASR Kaldi Service, MT Open Neural Machine Translation (OpenNMT) TensorFlow Lite (TF-Lite), and TTS FastSpeech2 MBM TF-Lite. It is encouraged that future services also inherit from these base classes. Using the Kaldi Service as an example, here is a UML diagram (Fig. 3) showing these classes in use.

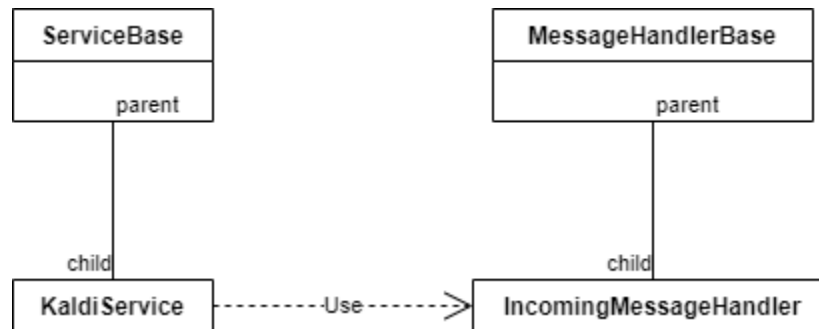


Fig. 3 UML class diagram of base classes in use

5.1 ServiceBase

This class inherits from the `android.app.Service` class and has code necessary to perform a number of functions:

- Instantiate the service
- Search for and read language configurations
- Maintain the list of configurations
- Instantiate the message handler class

Some of the methods on the class are abstract and must be overridden on a child class to perform the function it is designed to do. These are documented in the code.

5.2 MessageHandlerBase

This class inherits from the `android.os.Handler` class and has the code framework necessary to handle requests from and reply to the client. However, most of the important functions are abstract and ready to be overridden by child classes with code specific to that service that is being instantiated. One method that is not abstract is the code to handle configuration requests. Inheriting from this class allows that requirement to be satisfied.

5.3 Managing Configurations

Configuration files for the service are located in a folder the service can easily find. In the Shareable Components Android architecture, these folders are all located under

Phone\Shareable Components

Each service has its own folder dedicated to it in the above folder, and in each of those are further subfolders, one for each LS present. In a particular language-set folder are a YAML configuration file and the model files associated with the language that is installed. As mentioned previously, the `ServiceBase` class already has the code in place to manage the list of LS configurations and return them to the client as a JSON object array when requested. However, a good understanding of these configuration files is necessary to building a service.

The Shareable Components architecture contains three main services already in place: ASR, MT, and TTS. They share common properties in their configurations, and each has specific requirements. The following three sections contain sample configuration files from each of those services with comments for every property.

For all of these configurations, the most important piece of information is the UUID that identifies the LS the service should use. This UUID should be different for every single LS created. There are many online utilities for generating these identifiers quickly and easily, such as <http://uuidgenerator.net>.

This identifier is necessary for the client to send *execute* requests to the service.

5.3.1 ASR Configuration Sample

ASR configurations for Kaldi must indicate which decoder to use. The most recent version of the system uses NNET3 exclusively for chain and chain2 models, but the legacy tri2b and nnet2 decoders can still be chosen in the configuration.

The ASR Kaldi Service stores its language-set configurations in folders under the following location:

```
Phone\Shareable Components\asr-kaldi
```

This configuration has the unique identifier that a client must send to the service to identify that it wants to use this LS to do the recognition.

```
# Configuration file for Shareable Component for ASR

# This is the main language of concern for Automatic Speech
  Recognition (ASR)
# The language name is meant to be a human-readable string
  for display to a user
name: Libyan Arabic
# Unique UUID to identify this ASR for use by clients via
  code
uuid: d1e059c0-8bc6-4149-b8d8-1c2f23e7672f
# Name of the Shareable Component service that these model
  files are intended to be used by
intendedService: asr-kaldi
# ISO-639-1 two-letter language code.
languageCode2: ar
# ISO-639-3 three-letter language code.
languageCode3: ayl
# ISO-3166 country codes for regional identification of the
  language variant.
# Include both the two-letter and three-letter codes
countryCode2: LY
countryCode3: LBY
# Decoder type can be tri2b, nnet2, or nnet3
decoderType: nnet3
# Description contains text that can be passed on to a user
  describing the language or how the ASR was developed.
description: >
  This is the Automatic Speech Recognition service for
    recognizing speech in Libyan-Arabic.
  These are Kaldi chain models intended to be decoded using
    the nnet3 decoder.
# Version information about the ASR model files
version: 1.0.0
```

5.3.2 MT Configuration Sample

MT configurations have language pairs instead of a single language and thus are more complicated. For example, the Arabic language can be translated to and from English, but in the future, it could also be translated to and from Spanish. This is a

case where the design was meant to be open-ended even though the current system does not take advantage of this possibility at this time. They are included in the same LS instead of being separated because they could potentially share model files such as using the same Arabic dictionaries. Model and vocabulary files can be potentially very large, so avoiding duplication is desirable.

For nomenclature and clarity, the languages in the pair are assigned as “A” and “B”. The “A” language is usually the foreign language that is the main concern for the LS, while the “B” language is usually English.

The “A” language is the Arabic in the following example, while English and Spanish are two different “B” languages. The configuration needs to know which files to load up for an A-to-B translation as well as a B-to-A translation and this is all defined as YAML classes in the file. Each of those directions has a unique identifier associated with it; thus, there is more than one per configuration file.

The OpenNMT Tensorflow Lite service stores its language-set configurations in folders under the following location:

```
Phone\Shareable Components\mt-opennmttf
```

The YAML configuration makes extensive use of nested classes and arrays to achieve this architecture:

```
# Configuration file for Shareable Component for MT

# The root is the A language that is the main language of
  concern for Machine Translation (MT)
# The language name is meant to be a human-readable string
  for display to a user
name: Libyan Arabic
# Name of the Shareable Component service that these model
  files are intended to be used by
intendedService: mt-opennmttf
# ISO-639-1 two-letter language code.
languageCode2: ar
# ISO-639-3 three-letter language code.
languageCode3: ayl
# ISO-3166 country codes for regional identification of the
  language variant.
# Include both the two-letter and three-letter codes
countryCode2: LY
countryCode3: LBY
# The other languages that translate to and from the A
  language are the B languages
translationLanguages:
  # Most clients will use the names of the A and B languages
    to create a language pair string such as
  # "English to Libyan-Arabic" and "Libyan-Arabic to English"
  - name: English
```

```

# Similarly language and country codes can be combined
# into pairs: "ar2en" "eng-USA2ayl-LBY"
languageCode2: en
languageCode3: eng
countryCode2: US
countryCode3: USA
# List of model file keys and actual file names required
# by the MT service.
# The key names will be dictated by the MT service. See
# the documentation for the MT service.
# This list will vary depending on the MT service.
modelFiles:
  # Identify which file located in this directory to use
  # for translating A to B.
  - modelFileAtoB: ar-en-transformer-p2-v80-cc-wm-mu-os-
    geo-step230k-tf2.5.0.tflite
  # Identify which file located in this directory to use
  # for translating B to A.
  - modelFileBtoA: en-ar-transformer-p2-v80-cc-wm-mu-os-
    geo-step220k-tf2.5.0.tflite
  # Vocabulary file for Libyan
  - vocabularyA: p2CC-WM-MU-OS-Geo-en-ar80Vocab.ar
  # Vocabulary file for English
  - vocabularyB: p2CC-WM-MU-OS-Geo-en-ar80Vocab.en
# Unique UUID to identify this MT translating A to B for
# use by clients via code
uuidAtoB: a887d6e8-e3d2-4a1a-a14c-e36795544298
# Unique UUID to identify this MT translating B to A for
# use by clients via code
uuidBtoA: 43d3769d-f107-41ff-beea-d99f2945ba45 # Unique
# UUID to identify this MT translating A to B for use by
# clients via code.
# Description contains text that can be passed on to a
# user describing the language pairs,
# how the models were developed, or anything else you
# want to pass on to a user.
description: >
  This is the machine translation service for translating
  Libyan-Arabic to and from English.
  The models are for OpenNMT Torch (lua).
# Version information about the MT model files in this
# directory
version: 1.0.0
- name: Spanish
  languageCode2: es
  languageCode3: esp
  countryCode2: ES
  countryCode3: ESP
  modelFiles:
    - modelFileAtoB: ar-es-transformer-p2-v80-cc-wm-mu-os-
      geo-step230k-tf2.5.0.tflite
    - modelFileBtoA: es-ar-transformer-p2-v80-cc-wm-mu-os-
      geo-step220k-tf2.5.0.tflite
    - vocabularyA: p2CC-WM-MU-OS-Geo-en-ar80Vocab.ar
    - vocabularyB: p2CC-WM-MU-OS-Geo-en-ar80Vocab.es
  uuidAtoB: d5fadaed-4fc5-470d-8b47-613b80af8927
  uuidBtoA: 37705682-45e2-4299-ade2-cf5769e81d4a

```

```

description: This is the machine translation service for
  translating Libyan-Arabic to and from Spanish.
version: 1.0.0
- name: French
  languageCode2: fr
  languageCode3: fra
  countryCode2: FR
  countryCode3: FRA
  modelFiles:
    - modelFileAtoB: ar-fr-transformer-p2-v80-cc-wm-mu-os-
      geo-step230k-tf2.5.0.tflite
    - modelFileBtoA: fr-ar-transformer-p2-v80-cc-wm-mu-os-
      geo-step220k-tf2.5.0.tflite
    - vocabularyA: p2CC-WM-MU-OS-Geo-en-ar80Vocab.ar
    - vocabularyB: p2CC-WM-MU-OS-Geo-en-ar80Vocab.fr
  uuidAtoB: 2e199a57-d18a-45d4-8ece-bd8bf653f42b
  uuidBtoA: 37adf73e-549c-4748-826d-6ba457797f40
description: This is the machine translation service for
  translating Libyan-Arabic to and from French.
version: 1.0.0

```

5.3.3 TTS Configuration Sample

TTS configurations specify a number of model files that must be present for the service to operate.

The FastSpeech2 MultiBand MelGan TTS service stores its language-set configurations in folders under the following location:

```
Phone\Shareable Components\tts-fs2mbm
```

Like an ASR configuration, a TTS configuration contains a single unique identifier for the LS, as shown in the following:

```

# Configuration file for Shareable Component for TTS

# This is the main language of concern for Text to Speech
  (TTS)
# The language name is meant to be a human-readable string
  for display to a user
name: Libyan Arabic
# Unique UUID to identify this TTS for use by clients via
  code
uuid: f0e11d8c-7be4-4247-ac94-044e035c4fca
# Name of the Shareable Component service that these model
  files are intended to be used by
intendedService: tts-fs2-mbm-tf
# ISO-639-1 two-letter language code.
languageCode2: ar
# ISO-639-3 three-letter language code.
languageCode3: ayl
# ISO-3166 country codes for regional identification of the
  language variant.
# Include both the two-letter and three-letter codes
countryCode2: LY

```

```

countryCode3: LBY
# List of model file keys and actual file names required by
  the TTS service.
# The key names will be dictated by the TTS service. See the
  documentation for the TTS service.
# This list will vary depending on the TTS service.
modelFiles:
  # The TensorFlow Lite file containing the model for
    synthesizing the waveform.
  - fastspeech2: fastspeech2.tflite
  # The mapping file for tokens to ids.
  - mapper: mapper.json
  # The TensorFlow Lite file containing the model for the
    vocoder to produce the speech.
  - melgan: melgan.tflite
# Description contains text that can be passed on to a user
  describing the language or how the TTS was developed.
description: >
  This is the Text to Speech service for creating speech in
    Libyan-Arabic.
  The models are FastSpeech2 with Multi-Band MelGAN Vocoder
    using TensorFlow Lite.
# Version information about the TTS model files
version: 1.0.0

```

5.4 Anatomy of an *Execute* Request

Once the service has been built, it needs to handle service requests from any type of client. The request comes in as a message object with key-value pairs of data as noted previously. The service's job is to parse out this request, check it for required data field values that it specifically needs, execute the task the service was designed to perform, check to make sure the output was generated correctly, create a new message object, populate the key-value pairs of data as appropriate, and then send the reply message back to the client through the appropriate channel.

Incoming execute requests have two ways of passing the data to process. In the first way, the client can specify that the input data resides in a file on the device by specifying its full-qualified name and path. The format of that file is particular to what that service requires and information about that format may be included in the metadata key-value pairs. For example, an ASR service requires an audio wave file. That file may be saved on the device, but the file's sampling rate and encoding should be sent as part of the request's metadata. Similarly, the requesting client may specify that the output from the service also be saved in a file, in which case the client should specify that output file location as part of the request.

The second way of passing data is by directly putting it in the message object as one of the key-value pairs. However, there is a limit to the size of a message object, which is 500 kB. Putting too much data in the message object will result in errors

in the system. This is easily reached when audio data is transmitted, for example. To get around this limitation, the Shareable Components architecture supports data-chunking in either direction from client to server or server to client. The data is broken into smaller chunks and sent sequentially as separate message objects, which are then reassembled into the original data after the final chunk is received.

To indicate the data has been chunked, the message should contain the key “CHUNK”, with a value that can be one of three choices:

- FIRST:N—where N is the total number of chunks to expect including this first one
- N—the chunk number, which ranges from 2 to the second-to-last
- LAST:N—indicates this is the last chunk, and N is the total number that should be received; this should be the same number that was sent with the first chunk

Since data is usually sent as byte arrays, splitting and concatenating the results is a simple operation.

If the “CHUNK” key-value pair is present in the message, then the “INPUT” key-value pair contains the chunk fragment; otherwise, the entire data object is assumed to be contained in the “INPUT” key-value pair. If it is not the last chunk, then it is saved and execution ends for that request. It is expected that subsequent requests to the service will contain further chunks, and when the last chunk is received it is reassembled and the data finally processed.

All chunking and input are ignored if the “INPUT_FILENAME” key exists, since external files take precedence. Using files to transfer large data is the preferred method of transferring data from client to service since both are colocated on the same device. For small data, such as a single sentence of text sent to an MT service, it is preferable to send it directly in the message object since that eliminates file reading–writing overhead.

The reverse process should take place when dealing with the output of the service. The service is responsible for checking to see how it should send it back to the client. If the client’s initial request contained an “OUTPUT_FILENAME” key-value pair, it should place the result in that location specified and include any metadata about that file in the reply message.

If the client did not specify an output file location, the service should then check to see if the size of the output is too large to send in a single message and then chunk it as previously, except with the “OUTPUT” key-value pair. Again, other metadata about the data should be sent with the message. For example, if the service is for

TTS, it should specify what the output wave data array's sampling rate and encoding are.

In the case of any errors or warnings generated by the service, it should similarly populate the reply message in the appropriate error key-value pairs and send that back to the client.

All requests sent to the service must have a unique identifier generated by the client attached. This is different than the identifier used to identify the LS. This is a request identifier that is used by the client to synchronize requests and replies and also allows for easy grouping of chunks. In the case an input filename was sent as the request, that input filename should also be returned in the reply to assist the client in further synchronization.

All Shareable Components services are expected to time their execution and return that value with the reply. Figure 4 depicts an *execute* request via flow chart.

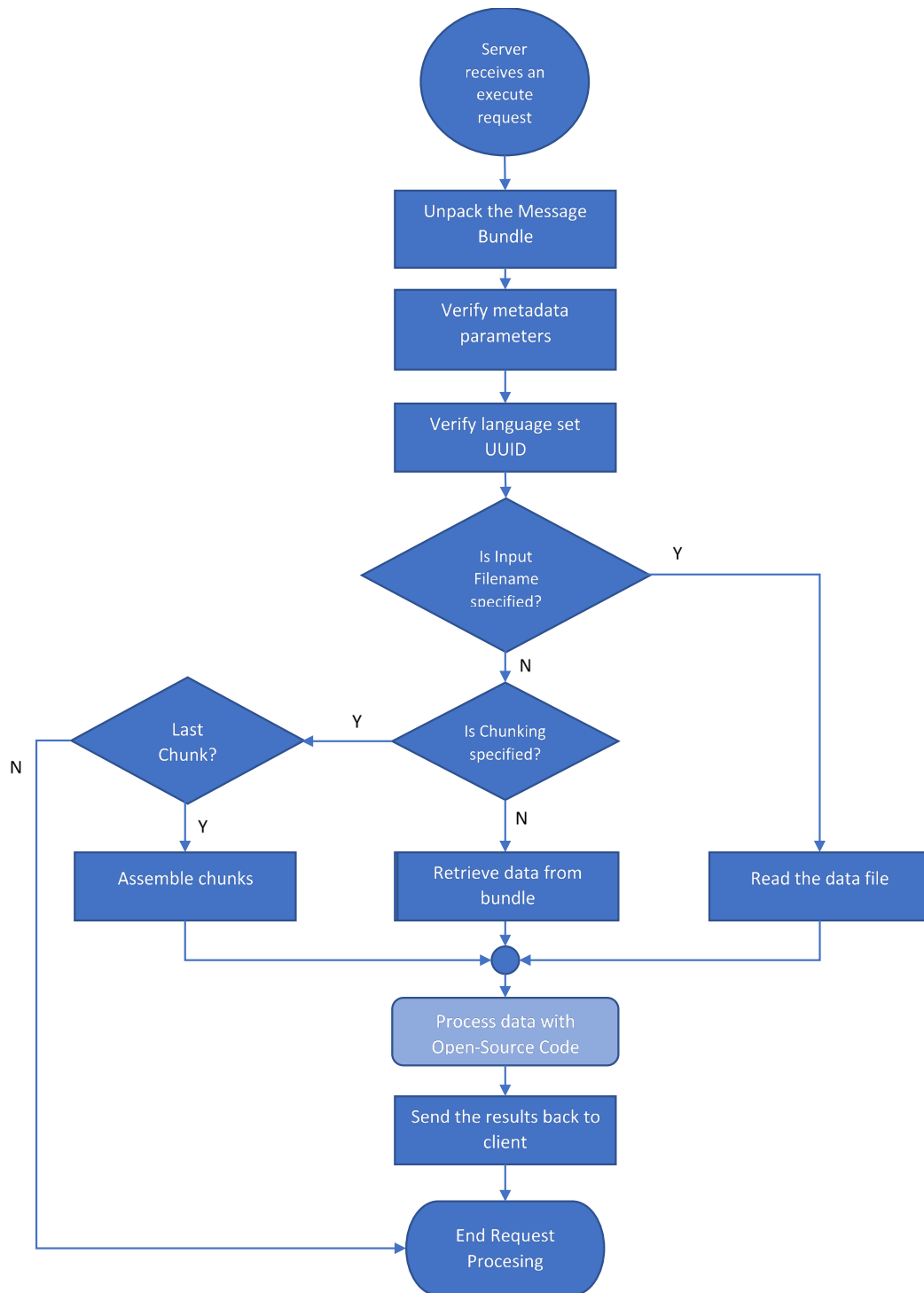


Fig. 4 Flow chart depicting the processing of an *execute* request

6. Installing a Service

The Shareable Components architecture is designed around the idea of a service having a single adapter and a number (that can be one, or more) of LSs installed. These LSs include a number of model files that must be read by the adapter to perform its function. In the Android operating system, specific permission must be granted to the service to read these externally installed files. This permission is separate from what a client may request when the application is launched for the first time.

After installing a service on an Android phone, it is a good practice to verify the appropriate “Storage” permission has been granted to the service.

To install LSs on the phone, one need simply to copy over all of the appropriate model files and configurations to the correct location for that service.

The Shareable Components system comes with installation scripts and instructions in a separate document that automate this process and set the correct permissions.

7. Shareable ASR Kaldi Service

This service performs ASR using the Kaldi open-source toolkit. When the service receives an *execute* request, it looks for the following parameters:

- Request ID—the client’s identifier for this job
- Language UUID—which language models to use for recognition, taken from the configuration
- Sample Rate—for Kaldi Tri2b, this value must be 16000. For the other decoders, it will down-sample as needed.
- Encoding—this is PCM 16 bit
- Input Filename—to specify what file to recognize

Once all these parameters are collected and verified as valid, the service instantiates a Kaldi Bridge class. This class connects the service code to the adapter code that directly calls the Kaldi Software Development Kit methods that have been compiled in C++ code. This code runs in another background process to protect the service from unhandled exceptions.

When Kaldi finishes, the service packages up the reply message with the following parameters:

- Request ID—taken from the request message so the reply matches

- Execution Time—the time the service took to run the recognition
- Output—typically the output text of the ASR is small enough to include in the reply, but the service checks the size; if it is too big, it will chunk it automatically
- Chunk—if necessary, this contains the chunk information
- Status—success or error
- Error Message—if an error occurred, messages describing it appear here.

After returning the reply, the service has completed its function.

8. Shareable MT OpenNMT TF-Lite Service

This service performs MT using the OpenNMT TF-Lite system. TensorFlow models are trained in the OpenNMT system and then converted for use with TF-Lite that can run on Android. TF-Lite has an implementation in Java, with AndroidStudio tie-ins, that can be called directly from the Android service's code, making it really simple to implement.

When the service receives an execute request, it looks for the following parameters:

- Request ID—the client's identifier for this job
- Language UUID—which language pair models to use for translation, taken from the configuration
- Input—text strings are generally small enough to not reach the 500-kB limit, but the service can handle input chunking if necessary

Once the parameters are collected and verified as valid, the service instantiates the OpenNMTBridge class that separates the service adapter code with the code specific to running MT. The bridge code then launches TF-Lite, loading the specified model and vocabulary files as needed, preprocessing the text, sending it through the decoder, and then postprocessing the result. This execution runs in a separate thread to insulate the service from unhandled exceptions.

When OpenNMT finishes, the service packages up the reply message with the following parameters:

- Request ID—taken from the request message so the reply matches
- Execution Time—the time that the service took to run the translation

- Output—typically the output text of the MT is small enough to include in the reply; but, the service checks the size and if it is too big, it will chunk it automatically
- Chunk—if necessary, this contains the chunk information
- Status—success or error
- Error Message—if an error occurred, messages describing it appear here.

After returning the reply, the service has completed its function.

9. Shareable TTS FastSpeech2 MultiBand MelGan TF-Lite Service

This service performs TTS using FastSpeech2 and the MultiBand MelGan vocoder. The process runs in TF-Lite, which runs natively on the Android system just as outlined in Section 8.

When the service receives an execute request, it looks for the following parameters:

- Request ID—the client’s identifier for this job
- Language UUID—which language model to use for translation, taken from the configuration
- Input—text strings are generally small enough to not reach the 500-kB limit, but the service can handle input chunking if necessary
- Output Filename—since the output of the TTS is usually a large wave file, it is preferred that the client specify where it would like the service to save the result; also, it is possible to send it back through the reply message with chunking

Once all the parameters are collected and verified as valid, the service instantiates an FS2MBMBridge class. This class connects the service to the adapter code that runs the decoder. The bridge class first sends the text through a preprocessor that has code specific to the language requested to normalize the text, convert numbers to text, and then convert them to the IDs that were used to train the TensorFlow models. It runs the model, sends the output to the vocoder to create the array of binary data that represents the wave form, then saves that to a wave file in the location specified by the request.

This all runs in a separate thread to insulate the service from unhandled exceptions as with the previous classes. When the synthesis is complete, the service packages up the reply message with the following parameters:

- Request ID—taken from the request message so the reply matches
- Execution Time—the time the service took to run the synthesis
- Output Filename—as specified in the request
- Output—if no file location is specified, this contains the array of binary data
- Chunk—if necessary, this contains the chunk information
- Sample Rate—the sample rate this TTS generates is 22050 Hz
- Encoding—the TTS produces PCM Float encoded data
- Status—success or error
- Error Message—if an error occurred, messages describing it appear here

Since the TTS generates its wave files in PCM Float, which can get large, it is preferred to have it write out the wave file with all of the appropriate headers by using the Output Filename request parameter.

After returning the reply, the service has completed its function.

10. Creating a Client

The client application is where one creates a workflow chaining together the Shareable Component services into a useful and desired outcome. The client application's job is to provide an interface for a user to interact with to get work done.

As such, the client application must be aware of or discover by some means what services are available to it that are installed on the system. It can then query those services to determine what languages are supported and then display choices to the user based on those results.

When the user starts some work that uses those services, the client application must then interact with each service in turn according to its workflow, await the results of each, and then put the final result together in a manner the user can understand and use.

The client application needs to be aware of what each service requires and send and receive the data in the correct format, whether by sending it with the request messages (dealing with chunking if the data is large) or by managing some sort of file system.

The client also should have sophisticated error handling in the event the service is unable to process the data given to it, which can then give appropriate feedback to the user.

11. ULTRA Application Client

The ULTRA application is an example of a complex client to the Shareable Components services. In this client, the workflow uses all three of the services—ASR, MT, and TTS—to create a Speech-to-Speech translation system in both directions (Fig. 5). By this design, the ULTRA application can be used by a nonlinguist to communicate with foreign nationals while in the field using only a cell phone.

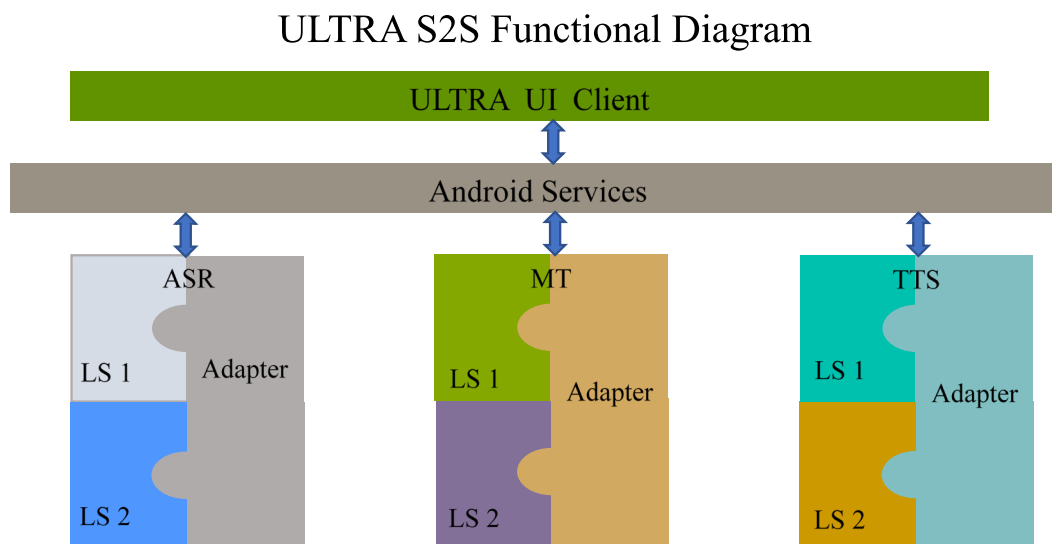


Fig. 5 ULTRA client uses Shareable Components services

The ULTRA User Manual² covers how the application can be used. In this section, the underlying architecture will be described in terms of how the client was built and how the ULTRA application works under the hood.

The ULTRA client first records speech spoken by the user (Fig. 6) and saves it in a wave file. Then it sends that file to be processed by the ASR service, binding to it through the Android service. After it receives the recognized text, it then sends that text to the bound MT service and waits for that response. Once the translation is received back through the response channel, the client calls the TTS service and requests that a file be generated in the file system for the spoken translated speech. When that is complete, the client then plays back the final file through the phone's speaker. All files generated by the processing are saved in its translation history for later analysis and review.

ULTRA Client Workflow

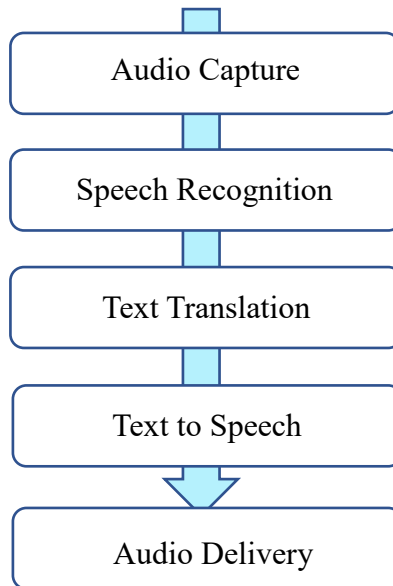


Fig. 6 ULTRA client workflow for speech-to-speech translation

ULTRA was built with specific language pairs defined that call particular LSs on each of the three main services. These workflows of were preassembled by the development team and set in configuration files that are described in the next section. New language-pair combinations can be created simply by creating new configuration files for ULTRA, but they must still follow the same basic workflow of ASR–MT–TTS.

The ULTRA client application is a React-Native application written mostly in Javascript with native Java code in place to do the client–server communication.

11.1 Configuration Files

When ULTRA is launched, the application looks for YAML configuration files in the following location:

Phone\Ultra\Languages

Any configuration it finds is opened and parsed and then made available to the application. In this way, new configurations can be created or existing ones modified between launches of the application. The configuration file contains information specific to which ASR, MT, and TTS engine and language should be used when performing a Speech-to-Speech operation for a given language pair. Each configuration is specific to a single language pair and contains information for translating in both directions, to and from one language to the other. Also

included in the configuration are prompts in both languages that the system uses to display at certain points during its operation, as well as human-readable text that can be displayed to describe the language pair in the configuration so that the user can pick it from a selection of available language pairs.

This configuration file represents a priori knowledge of the languages installed on the system. A configuration file that contains references to languages not installed with the Shareable Components services should not be included with ULTRA and will cause errors if it is.

The following are two examples of a configuration file. The first is for Libyan Arabic:

```
# The location for which this language pack is designed
location: Libya
# Readable description used for display in Ultra
description: Libyan Arabic to English - Chain2 and TF-Lite MT
# Ultra is designed to be Speech-Speech for two languages,
#   requiring ASR, MT, and TTS for each direction of
#   translation
# A Language - Libyan Arabic
# B Language - English
#
# Libyan Arabic to English
AtoB:
  sourceName: Arabic
  sourceNative: عربي
  sourceAbbreviation: AR
  targetName: English
  targetNative: انجليزي
  targetAbbreviation: EN
  # Libyan Arabic ASR
  ASR:
    intent: mil.army.arl.shareable.asr.KaldiService
    uuid: 869c1822-4b64-4746-953c-afb25fade82c
  # Libyan Arabic to English MT
  MT:
    intent: mil.army.arl.shareable.mt.OpenNMTTFSservice
    uuid: a887d6e8-e3d2-4a1a-a14c-e36795544298
  # English TTS
  TTS:
    intent: androidTTS
    uuid: en-US-language
  # Prompts displayed by the application when starting with
  #   language A as the source for a speaker of A to see
  prompts:
    recording: جاري التسجيل...
    sentence: تكلم جملة كي يتم ترجمتها...
    cancel: إلغاء!
    translate: ترجم
    recognizing: جاري التعرف...
    translating: جاري الترجمة...
    synthesizing: جاري توليف الكلام...
```

```

        deleting: مسح الترجمة القديمة...
# English to Libyan Arabic
BtoA:
    sourceName: English
    sourceNative: انجليزي
    sourceAbbreviation: EN
    targetName: Arabic
    targetNative: عربي
    targetAbbreviation: AR
# English ASR
ASR:
    intent: mil.army.arl.shareable.asr.KaldiService
    uuid: 2d7a1b8a-45cd-4b8c-81b8-063f78632492
# English to Libyan Arabic MT
MT:
    intent: mil.army.arl.shareable.mt.OpenNMTTFSservice
    uuid: 43d3769d-f107-41ff-beea-d99f2945ba45
# Libyan Arabic TTS
TTS:
    intent: mil.army.arl.shareable.tts.FS2MBMService
    uuid: 1a47a2a3-910e-4b5d-8e94-c6b8bdc437b1
# Prompts displayed by the application when starting with
    language B as the source for a speaker of B to see
prompts:
    recording: Recording...
    sentence: Speak a sentence to translate...
    cancel: Cancel
    translate: Translate
    recognizing: Recognizing...
    translating: Translating...
    synthesizing: Synthesizing speech...
    deleting: Deleting...

```

The second configuration file is for African French:

```

# The location for which this language pack is designed
location: Africa
# Readable description used for display in Ultra
description: French to English - Chain2 and TF-Lite MT
# Ultra is designed to be Speech-Speech for two languages,
    requiring ASR, MT, and TTS for each direction of
    translation
# A Language - African French
# B Language - English
#
# African French to English
AtoB:
    sourceName: French
    sourceNative: Français
    sourceAbbreviation: FR
    targetName: English
    targetNative: Anglais
    targetAbbreviation: EN
# African French ASR
ASR:
    intent: mil.army.arl.shareable.asr.KaldiService
    uuid: 9cb56977-7c60-45c5-8091-e767bef35483

```

```

# African French to English MT
MT:
    intent: mil.army.arl.shareable.mt.OpenNMTTFSservice
    uuid: cf316a4d-332d-4dae-a5bc-cb674841f663
# English TTS
TTS:
    intent: androidTTS
    uuid: en-US-language
# Prompts displayed by the application when starting with
    language A as the source for a speaker of A to see
prompts:
    recording: Enregistrement...
    sentence: Énoncez une phrase à traduire...
    cancel: Annuler
    translate: Traduire
    recognizing: Retrouvant...
    translating: Traduisant...
    synthesizing: Synthétiser l'énoncé...
    deleting: Suppression...
# English to African French
BtoA:
    sourceName: English
    sourceNative: Anglais
    sourceAbbreviation: EN
    targetName: French
    targetNative: Français
    targetAbbreviation: FR
# English ASR
ASR:
    intent: mil.army.arl.shareable.asr.KaldiService
    uuid: 2d7a1b8a-45cd-4b8c-81b8-063f78632492
# English to African French MT
MT:
    intent: mil.army.arl.shareable.mt.OpenNMTTFSservice
    uuid: 312b517c-8053-4db6-8fc4-2bab016074bf
# African French TTS
TTS:
    intent: androidTTS
    uuid: fr-FR-language
# Prompts displayed by the application when starting with
    language B as the source for a speaker of B to see
prompts:
    recording: Recording...
    sentence: Speak a sentence to translate...
    cancel: Cancel
    translate: Translate
    recognizing: Recognizing...
    translating: Translating...
    synthesizing: Synthesizing speech...
    deleting: Deleting...

```

For each of ASR, MT, and TTS, the intent describes which Shareable Components service should be invoked to perform that action for that direction of translation. These intents are used by the client code to identify the Android service to bind to. The UUID given corresponds to a given language and is taken directly from the

configuration file for that language as found in the Shareable Components folder on the phone. This is passed as a parameter when the service is invoked to perform its work.

For TTS, there is a special case when the language is using the Google Speech Services found as part of the Android system. In that case, the TTS intent is specified as “androidTTS” and the UUID corresponds to a name in the list of languages available in the Google Speech Services. This name is found through the programming API and must match exactly.

11.2 Workflow

The ULTRA workflow is initiated when the user records speech in one of the languages. When the recording is completed, and the user taps the button to continue, the following operations occur:

- 1) Saves the recorded speech into a file located in the History folder
- 2) Determines which language is currently selected
- 3) Binds to the ASR service indicated by the configuration intent
- 4) Sends a request to the ASR service to recognize the file; this request contains the location of the input speech file, its sampling rate and encoding (ULTRA always uses 16 kHz and PCM 16), and the UUID of LS the service should use
- 5) Awaits the ASR reply, displaying a progress screen to the user while it does so
- 6) Upon reply receipt, looks at the Status field for success or error
- 7) On a success, the recognized text is included in the “OUTPUT” field and ULTRA continues; on an error, the error message is logged, feedback is given to the user, the recorded file is deleted, and the workflow ends
- 8) Binds to the MT service indicated by the configuration intent for the language-pair direction
- 9) Sends a request to the MT service using the ASR recognition output as input
- 10) Awaits the MT reply, updating the progress screen to indicate that processing is moving forward
- 11) Upon reply receipt, looks at the Status field for success or error

- 12) On a success, the translated text is included in the “OUTPUT” field and ULTRA continues; on an error, the error message is logged, feedback is given to the user, the recorded file is deleted, and the workflow ends
- 13) Checks the TTS configuration to see if a Shareable Components Service or a Google Speech Service is specified
- 14) If it is Google Speech Service, invokes the TTS engine directly from the client application and passes in the translated text, saving the result to a wave file in the History folder
- 15) If it is a Shareable Components Service, binds to the TTS service
- 16) Sends a request to the TTS service to synthesize the file; this request contains an “OUTPUT_FILENAME” parameter so the service will save the resulting wave file in the History folder with a name given by the client
- 17) Awaits the TTS reply, updating the progress screen to indicate that processing is moving forward
- 18) Upon reply receipt, looks at the Status field for success or error
- 19) On an error, the error message is logged, feedback is given to the user, the recorded file is deleted, and the workflow ends
- 20) On a success, builds a translation-history YML file that contains the information generated throughout this workflow for this particular unit of Speech-to-Speech work (see Section 11.3 for its contents)
- 21) Adds a translation-history entry in the list for display that contains links to the original speech, the synthesized speech, the translated text, and all of the execution times.
- 22) Plays the synthesized speech through the speaker of the phone
- 23) Sets the system ready to perform another translation

11.3 Translation History

The ULTRA application saves each translation that it performs in the following location by default:

Phone\Ultra\History

This location can be changed through the application’s settings, allowing the user complete control of where translations are saved. Through the course of using the application, a user has the flexibility to save translations created on a specific location at a specific date; for example, to create a log of interviews using the folder system.

Each translation is contained in a set of three files that all start with the same unique number. This number is an integer representation of the time the original speech was recorded. Sorting the folder by name will naturally order the translations in the same order they were created.

- 1) The original speech is saved as a 16-kHz sampled PCM16 wave file.
- 2) The synthesized speech in the target translation language will also be saved as a wave file, but will have the suffix MT-TTS.
- 3) A YML file will accompany the two wave files that contain all of the information about the translation.

Here is an example YML file that is produced by ULTRA during a translation:

```
key: 1628275393498
startTime: Fri Aug 06 2021 14:43:16 GMT-0400 (EDT)
sourceFile: 1628275393498.wav
recognition: 'where is the gas station '
recognitionTime: 5.167267237
translation: 'الغاز؟ محطة أين '
translationTime: 1.111796041
targetFile: 1628275393498-mt-tts.wav
synthesisTime: 0.624946197
overallTime: 7.18
serviceDescription: Libyan Arabic to English - Chain2 and TF-
Lite MT
sourceName: English
sourceNative: انجليزي
sourceAbbreviation: EN
targetName: Arabic
targetNative: عربي
targetAbbreviation: AR
ASR:
  intent: mil.army.arl.shareable.asr.KaldiService
  uuid: 2d7a1b8a-45cd-4b8c-81b8-063f78632492
MT:
  intent: mil.army.arl.shareable.mt.OpenNMTTFSservice
  uuid: b78007de-7646-49b9-98f3-474e954edbab
TTS:
  intent: mil.army.arl.shareable.tts.FS2MBMSservice
  uuid: 1a47a2a3-910e-4b5d-8e94-c6b8bdc437b1
```

Contained in the file is the date and time it was created, the time it took to process, and all of the intermediate steps produced during the translation. It also specifies which service and engine were used to create the translation.

12. Conclusion

The Shareable Components architecture was designed to be a flexible means of encapsulating and packaging language technologies for use in sophisticated portable systems. As new and novel language technologies emerge and evolve, it should be a simple matter to incorporate them as new Shareable Components that can augment existing systems or lead to the creation of new applications.

Furthermore, as new languages sets are created it is a simple matter to add them with existing Shareable Components installations because of the language-set-to-adaptor design.

It is the goal of this architecture to be able to incorporate any language technology that proves to be useful, whether through the rich open-source community or other sources. This should lead to the creation of applications with a wide variety of workflows that can be used to satisfy any user's needs.

13. References

1. Bound services overview. Google Developers; 2020.
<https://developer.android.com/guide/components/bound-services>.
2. Schlesiger C. Universal Language Translator (ULTRA) user manual. DEVCOM Army Research Laboratory; 2021 Sep. Report No.: ARL-TN-1082.

Appendix. Libraries of Application Programming Interfaces (APIs)

The appendix that follows contains descriptions of each of the libraries comprising the Shareable Components architecture. The purpose of each class is included along with subclasses, methods, and relevant public or protected properties that need to be known when using the API.

A.1 Shareable Configuration

This library contains base classes and static classes to standardize communication between the client and service.

A.1.1 Actions

Contains constants that should be used as the “what” parameter in Android messages sent to and from the Shareable Components android service and client.

A.1.2 Field Details

- CONFIGURATIONS_REPLY

```
public static final int CONFIGURATIONS_REPLY
```

The service responds with this action with a JavaScript Object Notation (JSON) serialized list of valid configurations.

- CONFIGURATIONS_REQUEST

```
public static final int CONFIGURATIONS_REQUEST
```

A client uses this action get a list of configurations the service supports. Clients should request the list of configurations the service supports to get a valid selection for execution. These configurations refer to languages or language pairs and associated decoder types, and the like. Each of these configurations contains a unique engine identifier that must be passed with any execution request.

- EXECUTE_REPLY

```
public static final int EXECUTE_REPLY
```

Service response to the main execution with the results. The service will provide the output of its execution along with execution times, status, and other parameters relevant to describing the output.

- EXECUTE_REQUEST

```
public static final int EXECUTE_REQUEST
```

Client's request to the server to perform its main function. The client needs to send a unique configuration identifier to the service to specify which action to perform, as well as information about the data input and how the output will arrive. For most services this main function is recognition for automatic speech recognition (ASR), translation for machine translation (MT), and synthesis for text-to-speech (TTS).

A.1.3 ConfigurationBase

This class serves as a base configuration object. It contains properties common to most configurations used in the Shareable Components Android services.

A.1.4 Method Details

- `getDescription`

```
public java.lang.String getDescription()
```

Gets the description of the configuration. The description contains text that can be passed on to a user describing the language or how the language set (LS) and models were developed.

Returns:

The description of the configuration

- `setDescription`

```
public void setDescription(java.lang.String description)
```

Sets the description of the configuration. The description contains text that can be passed on to a user describing the language or how the LS and models were developed.

Parameters:

`description` - The description of the configuration

- `getModelFile`

```
public java.lang.String getModelFile  
(java.lang.String key)
```

Gets the filename specified by the key found in the model files list. The filename should point to an actual file in the folder with this configuration. The key is a unique identifier for a file this service needs in order to run.

Parameters:

`key` - The unique identifier of the file needed

Returns:

The name of the file found in the folder

- `setModelFile`

```
public void setModelFile(java.lang.String key,  
java.lang.String fileName)
```

Adds a new key and filename pair to the list of model files. The filename should point to an actual file in the folder with this configuration. The key is a unique identifier for a file this service needs in order to run.

Parameters:

key - The unique identifier of the file needed

location - The name of the file found in the folder

- **getModelFiles**

```
public java.util.List<java.util.LinkedHashMap<java.lang.S  
tring, java.lang.String>> getModelFiles()
```

Gets the list of key-value pairs representing a particular model file and its actual name in the folder. The key is a unique name for a file the service requires to be present. The value is the actual filename found in the folder located with this configuration file.

Returns:

The list of linked key-value pairs

- **setModelFiles**

```
public void setModelFiles  
(java.util.List<java.util.LinkedHashMap<java.lang.String,  
java.lang.String>> modelFiles)
```

Sets the list of key-value pairs representing a particular model file and its actual name in the folder. The key is a unique name for a file the service requires to be present. The value is the actual filename found in the folder located with this configuration file.

Parameters:

modelFiles - A list of linked key-value pairs

- **getVersion**

```
public java.lang.String getVersion()
```

Gets version information about the language-set model files.

Returns:

Version information about the language-set model files

- **setVersion**

```
public void setVersion(java.lang.String version)
```

Sets version information about the language-set model files.

Parameters:

`version` - Version information about the language-set model files

A.1.5 LanguageBase

This class serves as a base language configuration for use in a Shareable Component Android service. It contains properties to hold its name and standard abbreviations for the language and country most associated with the language variant.

A.1.6 Method Details

- `getCountryCode2`

```
public java.lang.String getCountryCode2()
```

Gets the ISO-3166* two-letter country code for regional identification of the language variant.

- `setCountryCode2`

```
public void setCountryCode2  
(java.lang.String countryCode2)
```

Sets the ISO-3166 two-letter country code for regional identification of the language variant.

Parameters:

`countryCode2` - The ISO-3166 two-letter code

- `getCountryCode3`

```
public java.lang.String getCountryCode3()
```

Gets the ISO-3166 three-letter country code for regional identification of the language variant.

- `setCountryCode3`

```
public void setCountryCode3  
(java.lang.String countryCode3)
```

Sets the ISO-3166 three-letter country code for regional identification of the language variant.

Parameters:

`countryCode3` - The ISO-3166 three-letter code

* Standard published by the International Organization for Standardization (ISO) that defines codes for names of countries, dependent territories, and special areas of geographical interest.

- `getIntendedService`

```
public java.lang.String getIntendedService()
```

Gets the name of the service for which this configuration is intended.

Returns:

The name of the service this configuration is intended for

- `setIntendedService`

```
public void setIntendedService  
(java.lang.String intendedService)
```

Sets the name of the service for which this configuration is intended.

Parameters:

`intendedService` - The name of the service this configuration is intended for

- `getLanguageCode2`

```
public java.lang.String getLanguageCode2()
```

Gets the ISO-639-1 standard's* two-letter language code.

Returns:

The ISO-639-1 two-letter language code

- `setLanguageCode2`

```
public void setLanguageCode2  
(java.lang.String languageCode2)
```

Sets the ISO-639-1 two-letter language code.

Parameters:

`languageCode2` - The ISO-639-1 two-letter language code

- `getLanguageCode3`

```
public java.lang.String getLanguageCode3()
```

Gets the ISO-639-3 three-letter language code.

Returns:

The ISO-639-3 three-letter language code

- `setLanguageCode3`

* Standard published by the International Organization for Standardization (ISO) that is concerned with representation of names for languages and language groups.

```
public void setLanguageCode3  
(java.lang.String languageCode3)
```

Sets the ISO-639-3 three-letter language code.

Parameters:

languageCode3 - the ISO-639-3 three-letter language code

- **getName**

```
public java.lang.String getName()
```

Gets the language name. The language name is meant to be a human-readable string for display to a user.

Returns:

The name of the language

- **setName**

```
public void setName(java.lang.String name)
```

Sets the language name. The language name is meant to be a human-readable string for display to a user.

Parameters:

name - The name of the language

- **toString**

```
public java.lang.String toString()
```

Converts this object to a string containing the value in the name property.

Overrides:

toString in class java.lang.Object

A.1.7 Parameters

Contains constants that are used as keys in message bundles sent to and from the Shared Components Service and clients that use it. Using these constants as keys will ensure the parameters are sent correctly and consistently and breaking changes will be minimized across versions.

Each of the static properties contains a key that can be passed into a bundle parcel. The “Actions” section describes which types of messages will contain these key-value pairs.

A.1.8 Field Details

- **CHUNK**

```
public static final java.lang.String CHUNK
```

Indicates the input parameter is just a piece of the entire input. If this parameter is present, then the array in either the INPUT or OUTPUT parameter is only a piece of the final array and must be reassembled by the client or server. Multiple requests/replies will be sent with each of the pieces of the array. The value of this parameter will indicate the ordering.

FIRST:N - indicates this is the first chunk and notifies the client/server that further chunks are forthcoming. The integer N after the colon indicates total number of chunks to expect including this one.

N - a numeral from 2 upward to the second to last indicating the chunk in the series

LAST:N - indicates this is the last chunk and final concatenation and processing should occur. The same count number as from the FIRST chunk is present after the colon to confirm the final count.

Actions:

EXECUTE_REQUEST, EXECUTE_REPLY

Bundle value type:

String

- **CONFIGURATIONS**

```
public static final java.lang.String CONFIGURATIONS
```

Constains a JSON encoded array of Configuration objects supported by the service. The type of configuration object is specific to each service and is in this library.

Actions:

CONFIGURATIONS_REPLY

Bundle value type:

String

- **ENCODING**

```
public static final java.lang.String ENCODING
```

The encoding for the audio binary byte array format. This will be PCM 16 or FLOAT and is the value that matches the static properties on the class "android.media.AudioFormat".

For ASR, this will identify the encoding of the incoming byte array in INPUT.

For TTS, this will identify be the encoding of the synthesized float array in OUTPUT.

Actions:

EXECUTE_REQUEST, EXECUTE_REPLY

Bundle value type:

int

- **ERROR_MESSAGE**

```
public static final java.lang.String ERROR_MESSAGE
```

A message describing what went wrong with the request; only included when STATUS is ERROR.

Actions:

CONFIGURATIONS_REPLY, EXECUTE_REPLY

Bundle value type:

String

- **EXECUTION_TIME**

```
public static final java.lang.String EXECUTION_TIME
```

The time in seconds it took to execute the request.

Actions:

EXECUTE_REPLY

Bundle value type:

double

- **INPUT**

```
public static final java.lang.String INPUT
```

The input the service should process.

For MT or TTS, this will be a string.

For ASR, it will be a byte array. See the individual service documentation.

This parameter will be ignored if INPUT_FILENAME is present in the request.

Actions:

EXECUTE_REQUEST

Bundle value type:

varies

- **INPUT_FILENAME**

```
public static final java.lang.String INPUT_FILENAME
```

The fully qualified path and name of a file the service should use as input. If this parameter is present in an EXECUTE_REQUEST it will take precedence over any value in the INPUT parameter.

On an EXECUTE_REPLY, this parameter will also be present as a reference to where the input data that was processed originated from.

Actions:

EXECUTE_REQUEST, EXECUTE_REPLY

Bundle value type:

String

- LANGUAGE_UUID

```
public static final java.lang.String LANGUAGE_UUID
```

The unique identifier (UUID) associated with the desired language configuration to use for execution. For MT, this will be one of uuidAtoB or uuidBtoA; for ASR and TTS it will be the uuid value.

Actions:

EXECUTE_REQUEST

Bundle value type:

String

- OUTPUT

```
public static final java.lang.String OUTPUT
```

The output generated by the service.

For MT or ASR, this will be a string.

For TTS, it will be a byte array. See the individual service documentation.

This parameter will not be present if OUTPUT_FILENAME was specified in the request.

Actions:

EXECUTE_REPLY

Bundle value type:

varies

- OUTPUT_FILENAME

```
public static final java.lang.String OUTPUT_FILENAME
```

The fully qualified path and name of a file the service should create as its output. This parameter should be specified by the client in the EXECUTE_REQUEST, and will likewise be present in the EXECUTE_REPLY as reference.

If this parameter is present in the request, the service will not send data back to the client in the reply in the OUTPUT parameter. It will only create the output file.

Actions:

EXECUTE_REQUEST, EXECUTE_REPLY

Bundle value type:

String

- REQUEST_ID

```
public static final java.lang.String REQUEST_ID
```

A unique identifier provided by the client to identify the execution request. The service will send this same request ID back for identification and matching purposes. This is essential since requests and replies are asynchronous.

Actions:

EXECUTE_REQUEST, EXECUTE_REPLY

Bundle value type:

String

- SAMPLE_RATE

```
public static final java.lang.String SAMPLE_RATE
```

The sample rate in Hz for the audio binary byte array (e.g., 16000 and 22050).

For ASR, this will identify the sample rate of the incoming byte array in INPUT.

For TTS, this will identify the sample rate of the synthesized float array in OUTPUT.

Actions:

EXECUTE_REQUEST, EXECUTE_REPLY

Bundle value type:

int

- STATUS

```
public static final java.lang.String STATUS
```

The status of the request, whether it succeeded or had an error.

SUCCESS - the request succeeded and output is returned

WARNING - the request succeeded and output is returned, but warnings were issued in WARNING_MESSAGE

ERROR - the request did not succeed, no output is returned, and the ERROR_MESSAGE parameter has details

The `Status` enum is provided in this library, which can be converted to a string.

Actions:

CONFIGURATIONS_REPLY, EXECUTE_REPLY

Bundle value type:

String

- **SYNTHESIS_SPEED**

```
public static final java.lang.String SYNTHESIS_SPEED
```

Optional. The desired speed of the synthesized speech.

A value of 1.0 is the default when this option is not present and represents the normal speed of the given Text-to-Speech synthesizer.

A value of < 1.0 will result in slower speech.

A value of > 1.0 will result in faster speech.

A value of ≤ 0 is not permitted and will result in the default value being used instead.

Actions:

EXECUTE_REQUEST

Bundle value type:

float

- **WARNING_MESSAGE**

```
public static final java.lang.String WARNING_MESSAGE
```

A message describing what problems or warnings arose during execution. Only included when STATUS is WARNING.

Actions:

CONFIGURATIONS_REPLY, EXECUTE_REPLY

Bundle value type:

String

A.1.9 Status

This enumeration contains values that will be returned by the service to indicate whether the request succeeded or failed.

- Enum Constants
 - **ERROR**
 - **SUCCESS**
 - **WARNING**

A.2 Shareable Android Base Classes

This library contains two base classes for creating a Shareable Components language service.

A.2.1 MessageHandlerBase

This class implements some basic methods and properties that are common among Shareable Components Android handlers. Creation of a new Shareable Components message handler is simplified by inheriting from this base class. A service that descends from ServiceBase will also need to be created, which is found in this same package.

A.2.2 Field Details

- service

```
protected final ServiceBase service
```

A.2.3 Constructor Details

- MessageHandlerBase

```
public MessageHandlerBase(ServiceBase service)
```

Constructor for this class. A reference to the service class is passed so methods can be called on it by this handler.

Parameters:

service - A descendant of ServiceBase

A.2.4 Method Details

- handleConfigurationRequest

```
protected void handleConfigurationRequest  
(android.os.Message msg)
```

Handles returning a list of JSON serialized Configuration objects. This method is called from handleMessage and returns appropriate errors as necessary.

Parameters:

msg - The incoming message object

- **handleMessage**

```
public void handleMessage(android.os.Message msg)
```

Incoming message handler. Required by all descendants of Handler. This method calls either handleConfigurationRequest or handleExecuteRequest as appropriate, or passes on to the super-class Handler.

Overrides:

handleMessage in class android.os.Handler

Parameters:

msg - The incoming message object

- **handleExecuteRequest**

```
protected abstract void handleExecuteRequest  
(android.os.Message msg)
```

Handles incoming requests to execute the main function of the service. Classes that descend from this base class must override and implement their logic here.

Parameters:

msg - The incoming message object

- **sendExecuteError**

```
protected void sendExecuteError  
(android.os.Messenger messenger, android.os.Bundle bundle,  
java.lang.String message)
```

Sends a reply to the client with an error. Convenience method that inserts the appropriate error indicators to the message bundle and then sends the message back to the client. Implementors should insert any other data into the bundle prior to calling this method to return the message.

Parameters:

messenger - The messenger object to reply to

bundle - The data bundle with other key-value pairs inserted

message - The error message this message should insert before sending

- **sendReply**

```
protected void sendReply(android.os.Messenger messenger,  
android.os.Bundle bundle,int what)
```

Creates a message and sends the data bundle back to the client. This convenience method sends the data back to the client. The *what* should be one of the Actions specified in the shareable-configuration library: Actions.CONFIGURATION_REPLY or Actions.EXECUTE_REPLY.

Parameters:

messenger - The reply messenger specified on the original incoming message

bundle - The data bundle of key-value pairs to send

what - The action to indicate the type of reply

A.2.5 ServiceBase

This class implements some basic methods and properties that are common among Shareable Components Android services. Creation of a new Shareable Components service is simplified by inheriting from this base class. A handler that descends from MessageHandlerBase will also need to be created, which is found in this same package.

A.2.6 Field Details

- *messenger*

```
protected android.os.Messenger messenger
```

A.2.7 Constructor Details

- ServiceBase

```
public ServiceBase()
```

A.2.8 Method Details

- addConfiguration

```
protected abstract void addConfiguration  
(java.io.InputStream stream, java.lang.String directory,  
java.lang.StringBuilder sbWarnings)
```

Adds a configuration of whatever type this services manages. Many Configuration classes for ASR, MT, and TTS are stored in the shareable-configurations library and are used by those services. New services may re-use those or create new ones and add appropriate management here.

Parameters:

stream - A stream to an open YAML-formatted configuration file that can be deserialized

directory - The folder that the config.yml file is currently located

sbWarnings - Any problems parsing entries in the configuration file should be logged by appending to this object

- **addUuid**

```
protected java.lang.String addUuid(java.lang.String uuid,  
java.lang.String modelFolder)
```

Add an association with an engine uuid and the folder its model files can be found. This is stored in a hashtable named configMap on this class and this method is used to add entries with duplicate checking. When an EXECUTE_REQUEST is received, this lookup can be used to quickly find the correct folder for the engine to find LS files to work with.

Parameters:

uuid - The unique identifier of the engine

modelFolder - The directory where the engine's model files are located

Returns:

Null on a success, an error string in a failure

- **clearConfigurations**

```
protected abstract void clearConfigurations()
```

Clear the locally managed store of Configuration objects, called prior to getting new configurations from the service.

- **getConfigurations**

```
public abstract java.lang.String getConfigurations()
```

Get the locally managed list of Configuration objects, serialized as a JSON string to be sent to a client on a CONFIGURATION_REQUEST.

Returns:

A serialized JSON string of Configuration objects

- **getConfiguration**

```
public abstract java.lang.Object getConfiguration  
(java.lang.String uuid)
```

Return a Configuration object from the local store for the supplied UUID. Configuration objects are uniquely matched by a UUID to identify what language and technology to process. Implementors that override this method should change the return type to a specific Configuration class.

Parameters:

uuid - A unique identifier for a configuration

Returns:

A Configuration class specific to this service

- **getMessenger**

```
protected abstract android.os.Messenger getMessenger()
```

Returns the Messenger that this service uses. The message handler created by this class passes the service as a parameter so the handler has access to a reference here in order to call methods.

Returns:

A Messenger created with an instance of a class descending from `MessageHandlerBase`

- **getModelFolder**

```
public java.lang.String getModelFolder  
(java.lang.String uuid)
```

Returns the model folder associated with the uuid. This method simply accesses the internal configMap to return the folder that was added with the addUuid method.

Parameters:

uuid - A unique identifier for an engine

Returns:

The directory associated with the uuid stored in configMap

- **getName**

```
public abstract java.lang.String getName()
```

A unique name for this service. This name is inserted in error messages returned to the client. The word “Service” will automatically be appended.

Returns:

The name of this service

- **getRootFolder**

```
protected abstract java.lang.String getRootFolder()
```

Gets the root folder for this service where LSs are stored. The service’s readConfigurations method will search for subfolders under the root for installed LSs.

Returns:

A fully qualified folder path on the android phone

- **onBind**

```
public android.os.IBinder onBind  
(android.content.Intent intent)
```

Called when a client binds to this service. This method returns a binder using the `MessageHandlerBase`. For convenience `readConfigurations` is called to load the list of installed LSs for this service in anticipation of a `CONFIGURATION_REQUEST`.

Specified by:

`onBind` in class `android.app.Service`

Parameters:

`intent` – The intent used to bind to this service

- `readConfigurations`

```
protected void readConfigurations()
```

Reads the installed LSs for this service. Opens the folder returned by `getRootFolder` and looks at all subfolders for configurations and LSs and builds a list of `Configuration` objects by calling `addConfiguration` with a reference to found `config.yml` files.

A.3 Shareable Utilities

This library contains a static class with convenience methods that client and services can use in their functioning.

A.3.1 Utils

This class contains static utility methods for the Shareable Components system of Android clients and services.

A.3.2 Method Details

- `concatArrays`

```
public static float[] concatArrays(float[] first,
float[] second)
```

Concatenates two float arrays into a single float array.

Parameters:

`first` - The array to be placed first in the concatenated array

`second` - The array to be placed second in the concatenated array

Returns:

The concatenated float array

- `concatArrays`

```
public static byte[] concatArrays(byte[] first,
byte[] second) throws java.lang.Exception
```

Concatenates two byte arrays into a single byte array.

Parameters:

`first` - The array to be placed first in the concatenated array

`second` - The array to be placed second in the concatenated array

Returns:

The concatenated byte array

Throws:

`java.lang.Exception` - Implementors should catch exceptions thrown by this method

- **convertToPCM16**

```
public static byte[] convertToPCM16(float[] input)
```

Converts a float array to a 16-bit integer byte array. This method is used to convert audio data before saving to a WAV file. The byte array is little-endian, typical of what is in a WAV file.

Parameters:

`input` - The float array to convert with ranges -1 to 1

Returns:

The 16-bit integer byte array

- **createChunks**

```
public
static java.util.ArrayList<java.lang.String> createChunks
(java.lang.String text, int chunkSize)
```

Divides a large string into a smaller pieces. When data objects are to be sent through a bound service, there are data-size restrictions. The most common strategy is to break up the data into smaller pieces, send them sequentially, and have the other side of the service reassemble them. This helper method does the work of breaking up the data and returning an ArrayList of items.

Parameters:

`text` - The large string that should be checked for breaking up into chunks

`chunkSize` - The size of chunk to create

Returns:

A list of string chunks

- **createChunks**

```
public    static java.util.ArrayList<float[]> createChunks
(float[] array, int chunkSize)
```

Divides a large float array into a smaller pieces. When data objects are to be sent through a bound service, there are data-size restrictions. The most common strategy is to break up the data into smaller pieces, send them sequentially, and

have the other side of the service reassemble them. This helper method does the work of breaking up the data and returning an ArrayList of items.

Parameters:

`array` - The large float array that should be checked for breaking up into chunks

`chunkSize` - The size of chunk to create

Returns:

A list of float array chunks

- **createChunks**

```
public static java.util.ArrayList<byte[]> createChunks  
(byte[] array, int chunkSize)
```

Divides a large byte array into a smaller pieces. When data objects are to be sent through a bound service, there are data-size restrictions. The most common strategy is to break up the data into smaller pieces, send them sequentially, and have the other side of the service reassemble them. This helper method does the work of breaking up the data and returning an ArrayList of items.

Parameters:

`text` - The large byte array that should be checked for breaking up into chunks

`chunkSize` - The size of chunk to create

Returns:

A list of byte array chunks

- **resample**

```
public static float[] resample(float[] input,  
int sampleRate, int newSampleRate)
```

Changes the sample rate of an audio float array. Downsampling always has a better result than upsampling.

Parameters:

`input` - The float array to resample with ranges -1 to 1

`sampleRate` - The current sample rate of the array

`newSampleRate` - The desired sample rate of the array

Returns:

A new float array with the resampled data

- **saveAsWav**

```
public static void saveAsWav(float[] data, int sampleRate,  
java.lang.String fileName) throws  
java.io.FileNotFoundException, java.io.IOException
```

Saves a float array as a WAV file. The audio array is assumed to have 1 channel (mono). Creates the appropriate WAV header.

Parameters:

`data` - The float array of audio data

`sampleRate` - The sample rate of the audio data

`fileName` - The desired location and filename to save the data

Throws:

`java.lang.Exception` - Implementors should catch exceptions thrown by this method: `FileNotFoundException`, `IOException`

- **saveAsWav**

```
public static void saveAsWav(byte[] data, int sampleRate,  
    java.lang.String fileName) throws  
    java.io.FileNotFoundException, java.io.IOException
```

Saves a 16-bit integer byte array as a WAV file. The audio array is assumed to have 1 channel (mono) and encoded as 16-bit PCM. Creates the appropriate WAV header.

Parameters:

`data` - The 16-bit integer array of audio data

`sampleRate` - The sample rate of the audio data

`fileName` - The desired location and filename to save the data

Throws:

`java.lang.Exception` - Implementors should catch exceptions thrown by this method: `FileNotFoundException`, `IOException`

A.4 Shareable Android ASR Service for Kaldi

This is the service that clients can use to perform ASR on recorded speech using the Kaldi engine.

A.4.1 Decoder

This abstract base class for decoders contains properties common to all three types of Kaldi decoders this system can process.

- **Field Detail**

- **utterance**

- `protected volatile java.lang.String utterance`

- **languagePath**

- `protected java.lang.String languagePath`

- **inputFilename**

```
protected java.lang.String inputFilename
```

- **Constructor Detail**

- **Decoder**

```
public Decoder(java.lang.String languagePath,  
java.lang.String inputFilename)
```

Creates a new instance of the decoder requiring the path to the language-set models and the fully qualified name of the wave file to process.

Parameters:

languagePath - The path to the LS model folder to use

inputFilename - The fully qualified path and name of the wave file to recognize

- **Method Detail**

- **getUtterance**

```
public java.lang.String getUtterance()
```

Gets the text of the speech that was recognized.

Returns:

The recognized text

- **run**

```
public abstract void run()
```

Required by Runnable, this method is invoked when the thread starts to start the Kaldi recognition process.

Specified by:

run in interface `java.lang.Runnable`

A.4.2 IncomingMessageHandler

This class handles all requests to the Shareable Components Kaldi Service sent by client applications. It inherits from a common Shareable Components base class for message handlers, which sets up the link to the parent service and does some of the processing for configurations.

This class has code to process execute requests to run the Kaldi ASR on an incoming byte array representing speech. This byte array must be sampled at 16 kHz and encoding in PCM16 integers if the configured language uses the Tri2b decoder. If it uses the nnet3 decoder (for chain models), the input can be other than that; however, use of that sample rate and encoding is encouraged in all cases.

- **Constructor Detail**

- IncomingMessageHandler

```
public IncomingMessageHandler(KaldiService service)
```

Creates the handler to deal with incoming request messages. Takes a reference to the service so it can call methods on it to get configurations.

Parameters:

service - The KaldiService object that holds language configurations

- Method Detail

- handleExecuteRequest

```
protected void handleExecuteRequest  
(android.os.Message msg)
```

Does the main work of handling incoming ASR requests. The message received by the service will contain a bundle of key-value pairs of data. The keys this handler expects are outlined in the shareable-configuration library found in the Actions and Parameters objects as static properties.

The Kaldi engine operates on a WAV file saved on the local file system, so this method will save the input byte array to a temporary file for processing if the original incoming request did not already specify the name of a file already on the system to process.

For input byte arrays, if they are too large (500 kB), the client should send them in a series of smaller chunks. This method is equipped to deal with reassembling the chunks before sending it on to Kaldi.

Specified by:

handleExecuteRequest in
class mil.army.arl.shareable.MessageHandlerBase

Parameters:

msg - The incoming request to execute the ASR

See Also:

Actions, Parameters

A.4.3 KaldiBridge

This class acts as a bridge between the Shareable Components Android service and the Kaldi ASR. This class loads the C++ libraries for Kaldi, constructs the thread under which it should run, executes the ASR and waits for the result, and then returns the resulting text string.

- Constructor Detail

- KaldiBridge

```
public KaldiBridge(java.lang.String modelDir)
```

The constructor requires a reference to the folder on the Android device that contains the LS models that the Kaldi ASR should use to recognize the waveform.

Parameters:

`modelDir` - A folder on the Android device containing language-set models

- **Method Detail**

- **getErrorMessage**

```
public java.lang.String getErrorMessage()
```

Gets any error messages that were generated by Kaldi while processing. Exceptions are caught by the bridge and then saved as messages on the bridge that can be retrieved by the caller to return meaningful information back to the client.

Returns:

An error message that occurred during Kaldi processing

- **recognize**

```
public java.lang.String recognize  
(java.lang.String decoderType,  
java.lang.String inputFilename)
```

Does the actual work of recognizing the specified wave file using the specified decoder. Constructs a new thread and waits for it to complete running the Kaldi ASR.

Parameters:

`decoderType` - The name of the decoder type, one of `nnet2`, `nnet3`, or `tri2b`

`inputFilename` - The fully qualified path and name of the file to process

Returns:

The text that was recognized from the wave file speech

A.4.4 KaldiService

This class sets up the Android service for the Kaldi ASR Shareable Components service. It inherits from the Shareable Components base class, which does the bulk of the work for setting up the service. It requires and communicates with the `IncomingMessageHandler` class, which handles all of the service requests.

This class manages a list of configurations that it locates in the folder “Shareable Components/asr-kaldi” on the Android device. Client applications can query this service for a list of configuration languages that this service can process.

Incoming requests to this service from bound clients need to follow specific guidelines. See the Shareable Components library `shareable-configuration` and the

two classes Actions and Parameters for full details, as well as the API documentation.

- Method Detail

- addConfiguration

```
protected void addConfiguration  
(java.io.InputStream stream,  
java.lang.String directory,  
java.lang.StringBuilder sbWarnings)
```

Adds a configuration to the service's list of configurations. The stream passed into this method can be read and deserialized into a Configuration class. Usually this stream is from an opened YAML file. If this method has problems reading the file, error messages can be appended to sbWarning. Since the service reads all folders to find configuration languages, a failure here is only a warning for this language, not a complete failure for all configurations.

This class maintains a Hashtable of configurations matching the uuid with a folder where the language is installed.

Specified by:

addConfiguration in
class mil.army.arl.shareable.ServiceBase

Parameters:

stream - The opened stream to the config.yml file

directory - The directory that the configuration was found

sbWarnings - Messages about failure to parse the config.yml file are appended to this object

clearConfigurations

```
protected void clearConfigurations()
```

Clears the list of configurations.

Specified by:

clearConfigurations in
class mil.army.arl.shareable.ServiceBase

- getConfiguration

```
public mil.army.arl.shareable.asr.Configuration getConf  
iguration(java.lang.String uuid)
```

Returns the Configuration object associated with the specified uuid. The Configuration object returned is specific to ASR and contains information necessary for this service to process the request, such as type of decoder to use.

Specified by:

getConfiguration in
class mil.army.arl.shareable.ServiceBase

Parameters:

uuid - The unique identifier to search for

Returns:

A Configuration object that matches the uuid

- getConfigurations

```
public java.lang.String getConfigurations()
```

Returns the whole list of configurations serialized into a string. On startup, the service reads the folder “Shareable Components/asr-kaldi” for all subfolders that have language configurations for the Kaldi service. These Configurations are saved on this object and returned as a JSON-serialized string to any client that requests it.

Specified by:

getConfigurations in
class mil.army.arl.shareable.ServiceBase

Returns:

A JSON serialized string containing all configurations found by this service

- getMessenger

```
protected android.os.Messenger getMessenger()
```

Returns the Messenger that is used by the service to handle incoming requests.

Specified by:

getMessenger in class mil.army.arl.shareable.ServiceBase

Returns:

The messenger that uses the IncomingMessageHandler

getName

```
public java.lang.String getName()
```

Gets the name of this service for use in error messages and logging. The word “Service” gets appended, so it is not necessary to include it here.

Specified by:

getName in class mil.army.arl.shareable.ServiceBase

Returns:

The name of this service

- getRootFolder

```
protected java.lang.String getRootFolder()
```

Gets the root folder where this service should look for language-set configurations.

Specified by:

`getRootFolder` in class `mil.army.arl.shareable.ServiceBase`

Returns:

A directory

A.4.5 NNet2Decoder

This class invokes the C++ NNET2 decoder and passes to it parameters needed to recognize the speech in the wave file.

- Constructor Detail

- NNet2Decoder

```
public NNet2Decoder(java.lang.String languagePath,  
java.lang.String inputFilename)
```

- Method Detail

- run

```
public void run()
```

Runs the Kaldi NNET2 decoder.

Specified by:

`run` in interface `java.lang.Runnable`

Specified by:

[run](#) in class [Decoder](#)

- nnet2Decode

```
public java.lang.String nnet2Decode(java.lang.String la  
nguageJString, java.lang.String[] configStringArray)
```

The Java Native Interface (JNI) to the C++ function.

A.4.6 NNet3Decoder

This class invokes the C++ NNET3 decoder and passes to it parameters needed to recognize the speech in the wave file.

- Constructor Detail

- NNet3Decoder

```
public NNet3Decoder(java.lang.String languagePath,  
java.lang.String inputFilename)
```

- Method Detail

- run

```
public void run()
```

Runs the Kaldi NNET2 decoder.

Specified by:

run in interface `java.lang.Runnable`

Specified by:

[run](#) in class [Decoder](#)

- o `nnet3Decode`

```
public java.lang.String nnet3Decode(java.lang.String languageJString, java.lang.String[] configStringArray)
```

The Java Native Interface (JNI) to the C++ function.

A.4.7 Tri2bDecoder

This class invokes the C++ Tri2b decoder and passes to it parameters needed to recognize the speech in the wave file.

Note: The wave file input must be sampled at 16 kHz and in PCM16 integer format.

- Constructor Detail

- o `Tri2bDecoder`

```
public Tri2bDecoder(java.lang.String languagePath, java.lang.String inputFilename)
```

- Method Detail

- o `run`

```
public void run()
```

Runs the Kaldi NNET2 decoder.

Specified by:

run in interface `java.lang.Runnable`

Specified by:

[run](#) in class [Decoder](#)

- o `tri2bDecode`

```
public java.lang.String tri2bDecode(java.lang.String[] configStringArray)
```

The Java Native Interface (JNI) to the C++ function.

A.5 Shareable MT Service for OpenNMT TensorFlow Lite (TF-Lite)

This is the service that can perform MT on foreign-language text using OpenNMT TF-Lite.

A.5.1 IncomingMessageHandler

This class handles all requests to the Shareable Components OpenNMT TF-Lite Service sent by client applications. It inherits from a common Shareable Components base class for message handlers, which sets up the link to the parent service and does some of the processing for configurations.

This class has code to process execute requests to run the OpenNMT TF-Lite MT on an incoming string of text in one language and translate it into another language.

- Constructor Detail

- IncomingMessageHandler

```
public IncomingMessageHandler(OpenNMTTFService service)
```

Creates the handler to deal with incoming request messages. Takes a reference to the service so it can call methods on it to get configurations.

Parameters:

service - The `OpenNMTTFService` object that holds language-pair configurations

- Method Detail

- `handleExecuteRequest`

```
protected void handleExecuteRequest  
(android.os.Message msg)
```

Does the main work of handling incoming MT requests. The message received by the service will contain a bundle of key-value pairs of data. The keys this handler expects are outlined in the shareable-configuration library found in the Actions and Parameters objects as static properties.

The OpenNMT engine operates on a string input as a parameter to the method. If the client has specified the name of a file already on the system to process, the service will open it and read all the lines in to send to the engine.

If the input string to translate is too large (500 kB), the client should be send it in a series of smaller chunks. This method is equipped to deal with reassembling the chunks before sending it to OpenNMT.

Specified by:

`handleExecuteRequest` in

class `mil.army.arl.shareable.MessageHandlerBase`

Parameters:

msg - The incoming request to execute the MT

See Also:

Actions, Parameters

A.5.2 ModelConfiguration

This class holds specific file locations taken from the configuration for a single language-pair translation.

A.5.3 MT_TFLite

This class is responsible for loading the models, converting the text to tokens, and invoking TF-Lite to translate text.

- Method Detail

- loadModel

```
@WorkerThread                                public boolean loadModel
(java.io.File modelFile,                        boolean useNNAPI,
int num_lite_threads)
```

Loads the model into the interpreter ready to process. This method can throw an exception by the Interpreter class if it fails to load.

Parameters:

modelFile - The location of the model file

useNNAPI - If true runs using NNAPI hardware accelerator

num_lite_threads - The number of threads to use

Returns:

True if the model was able to load, false if it could not find the file

- loadModelFromAssets

```
@WorkerThread                                public boolean loadModelFromAssets
(android.content.res.AssetFileDescriptor fileDescriptor
, boolean useNNAPI, int num_lite_threads)
```

Load TF-Lite model from assets file

Parameters:

fileDescription - The asset manager file descriptor to load from

useNNAPI - If true runs using NNAPI hardware accelerator

num_lite_threads - The number of threads to use

Returns:

Null if it could not load the model Taken from Tensorflow example <https://github.com/tensorflow/examples/blob/master/lite/examples/bert>

`_qa/android/app/src/main/java/org/tensorflow/lite/examples/bertqa/ml/QaClient.java`

- **readVocabularyFile**

```
public com.google.common.collect.HashBiMap<java.lang.String, java.lang.Integer> readVocabularyFile
(java.io.InputStream file)
```

Reads the vocabulary from a file

Parameters:

`file` - The file it will read, should be the same vocab files used to train the model

Returns:

The vocabulary as HashBiMap, used in the setVocab methods

- **setSourceVocabulary**

```
public void setSourceVocabulary
(com.google.common.collect.HashBiMap<java.lang.String,
java.lang.Integer> sourceVocabulary)
```

Set the source vocabulary file, use the same vocab files that were used to train the OpenNMT model; this vocabulary file will be used in converting the sentence, which will be translated to ids

Parameters:

`sourceVocabulary` - A vocab file can be attained using `readVocab(InputStream)`

- **setTargetVocabulary**

```
public void setTargetVocabulary
(com.google.common.collect.HashBiMap<java.lang.String,
java.lang.Integer> targetVocabulary)
```

Set the target vocabulary file, use the same vocab files that were used to train the OpenNMT model; this vocabulary file will be used in converting the ids to a sentence after translation

Parameters:

`targetVocabulary` - A vocab file can be attained using `readVocab(InputStream)`

- **translate**

```
public java.lang.String translate
(java.lang.String textToTranslate) throws
java.lang.Exception
```

Translates the text given. Model must have successfully loaded, and the vocabulary files must have been set.

Parameters:

`textToTranslate` - The text to translate

Returns:

The translated text

Throws:

`java.lang.InstantiationException` - With varying messages if the needed model files have not been loaded

A.5.4 OpenNMTTFBridge

This class acts as a bridge between the Shareable Components Android service and the OpenNMT TF-Lite MT. This class runs the TensorFlow interpreter, constructs the thread under which it should run, executes the MT and waits for the result, and then returns the resulting text string.

- Constructor Detail

- OpenNMTTFBridge

```
public OpenNMTTFBridge
(ModelConfiguration configuration)
```

The constructor requires a reference to the folder on the Android device that contains the LS models that OpenNMT-TF should use to translate the text.

Parameters:

`configuration` - The configuration object containing paths to files the engine needs

- Method Detail

- `getErrorMessage`

```
public java.lang.String getErrorMessage()
```

Gets any error messages that were generated by OpenNMT-TF while processing. Exceptions are caught by the bridge and then saved as messages on the bridge that can be retrieved by the caller to return meaningful information back to the client.

Returns:

An error message that occurred during OpenNMT-TF processing

- `translate`

```
public java.lang.String translate
(java.lang.String inputText)
```

Does the actual work of translating the text input using the T7 specified when constructing this class. Constructs a new thread and waits for it to complete running OpenNMT.

Parameters:

`inputText` - The text to be translated

Returns:

The translated text

A.5.5 OpenNMTTFService

This class sets up the Android service for the OpenNMT TF-Lite MT Shareable Components service. It inherits from the Shareable Components base class, which does the bulk of the work for setting up the service. It requires and communicates with the IncomingMessageHandler class, which handles all the service requests.

This class manages a list of configurations that it locates in the folder “Shareable Components/mt-opennmttf” on the Android device. Client applications can query this service for a list of configuration languages that this service can process.

Incoming requests to this service from bound clients need to follow specific guidelines. See the Shareable Components library shareable-configuration and the two classes Actions and Parameters for full details, as well as the API documentation.

- Method Detail

- addConfiguration

```
protected void addConfiguration  
(java.io.InputStream stream,  
java.lang.String directory,  
java.lang.StringBuilder sbWarnings)
```

Adds a configuration to the service’s list of configurations. The stream passed into this method can be read and deserialized into a Configuration class. Usually this stream is from an opened YAML file. If this method has problems reading the file, error messages can be appended to sbWarning. Since the service reads all folders to find configuration-language pairs, a failure here is only a warning for this language pair, not a complete failure for all configurations.

This class maintains a Hashtable of configurations matching the uuid with a folder where the language pair is installed.

Specified by:

addConfiguration in
class mil.army.arl.shareable.ServiceBase

Parameters:

stream - The opened stream to the config.yml file

directory - The directory that the configuration was found

sbWarnings - Messages about failure to parse the config.yml file are appended to this object

- clearConfigurations

```
protected void clearConfigurations()
```

Specified by:

clearConfigurations in
class mil.army.arl.shareable.ServiceBase

- o getConfiguration

```
public ModelConfiguration getConfiguration  
(java.lang.String uuid)
```

Returns the ModelConfiguration object associated with the specified uuid. The ModelConfiguration object returned is specific to this MT and contains information necessary for this service to process the request.

Specified by:

getConfiguration in
class mil.army.arl.shareable.ServiceBase

Parameters:

uuid - The unique identifier to search for

Returns:

A ModelConfiguration object that matches the uuid

- o getConfigurations

```
public java.lang.String getConfigurations()
```

Returns the whole list of configurations serialized into a string. On startup, the service reads the folder “Shareable Components/mt-opennmttf” for all subfolders that have language configurations for the OpenNMT service. These Configurations are saved on this object and returned as a JSON-serialized string to any client that requests it.

Specified by:

getConfigurations in
class mil.army.arl.shareable.ServiceBase

Returns:

A JSON-serialized string containing all configurations found by this service

- o getMessenger

```
protected android.os.Messenger getMessenger()
```

Returns the Messenger that is used by the service to handle incoming requests.

Specified by:

getMessenger in class mil.army.arl.shareable.ServiceBase

Returns:

The messenger that uses the IncomingMessageHandler

- o getName

```
public java.lang.String getName()
```

Gets the name of this service for use in error messages and logging. The word “Service” gets appended, so it is not necessary to include it here.

Specified by:

getName in class mil.army.arl.shareable.ServiceBase

Returns:

The name of this service

- getRootFolder

```
protected java.lang.String getRootFolder()
```

Gets the root folder where this service should look for language-set configurations.

Specified by:

getRootFolder in class mil.army.arl.shareable.ServiceBase

Returns:

A directory

A.5.6 Processor

This class contains methods for processing text either before or after it is run through the MT.

- Method Detail

- postProcess

```
public static java.lang.String postProcess  
(java.lang.String sentence)
```

Applies postprocessing to the output of the MT, which includes first-letter capitalization and detokenization of apostrophes and numbers

Parameters:

sentence - The sentence to postprocess

Returns:

The postprocessed text

- preprocessSentence

```
public static java.lang.String preprocessSentence  
(java.lang.String sentence)
```

Preprocesses the string and returns the preprocessed string

Parameters:

sentence - The string it will preprocess

Returns:

String that has been preprocessed

A.6 Shareable TTS Service for FastSpeech2 MBM TF-Lite

This is the service to run TTS using models trained in FastSpeech2 MBM TF-Lite. As of this version, it has been custom built to only run Libyan Arabic TTS; thus, the LS and the Adapter are commingled. In future versions as more languages are added to the service, the separation will be cleaner.

A.6.1 AbstractModule

This abstract class holds an options object for TF-Lite that both the FastSpeech2 and the MBM Vocoder use.

A.6.2 ArDigitsToAlphabet

This class for expanding digits to Arabic Alphabet.

- Constructor Detail

- ArDigitsToAlphabet

```
public ArDigitsToAlphabet(java.lang.String digits)
```

- Method Detail

- getArDigits

```
public java.lang.String getArDigits()
```

Gets the Arabic digits of the string used to construct this class.

- stringToDigit

```
public static java.lang.String stringToDigit  
(long digit)
```

Gets the number of the string used to construct this class.

A.6.3 FastSpeech2

This class is responsible for running the FastSpeech2 module using TF-Lite.

- Constructor Detail

- FastSpeech2

```
public FastSpeech2(java.lang.String modulePath)
```

Constructs this class with the specified TF-Lite model file.

Parameters:

`modulePath` - The fully qualified path and name of the TF-Lite model file

- Method Detail

- `getMelSpectrogram`

```
public org.tensorflow.lite.support.tensorbuffer.TensorBuffer getMelSpectrogram(int[] inputIds, float speed)
```

This method takes the results of the Processor and sends it through TF-Lite to produce the waveform of the speech.

Parameters:

`inputIds` - The array of data resulting from processing the text

`speed` - The TF-Lite parameter to control speed

Returns:

The tensorbuffer ready for the vocoder to process

A.6.4 FS2MBMBridge

This class acts as a bridge between the Shareable Components Android service and the FastSpeech2 MBM vocoder TTS synthesizer. This class constructs the thread under which it should run, executes the TTS and waits for the result, and then returns the resulting floating-point array of the wave form with a sampling rate of 22050 Hz.

- Constructor Detail

- `FS2MBMBridge`

```
public FS2MBMBridge(mil.army.arl.shareable.tts.Configuration configuration, java.lang.String modelDir)
```

The constructor requires a reference to the folder on the Android device that contains the language-set models that FS2MBM should use to synthesize the text.

Parameters:

`configuration` - The configuration object containing the information on where to find the tf-lite model files for the TTS

`modelDir` - The folder containing the tf-lite model files for the TTS

- Method Detail

- `getErrorMessage`

```
public java.lang.String getErrorMessage()
```

Gets any error messages that were generated by FS2MBM while processing. Exceptions are caught by the bridge and then saved as messages on the

bridge that can be retrieved by the caller to return meaningful information back to the client.

Returns:

An error message that occurred during FS2MBM processing

- `synthesize`

```
public float[] synthesize(java.lang.String inputText,  
float speed)
```

Does the actual work of synthesizing the text input using the language-set models specified when constructing this class. Constructs a new thread and waits for it to complete running FS2MBM.

Parameters:

`inputText` - The text to be synthesized

`speed` - The speech synthesis speed (1 is normal, less than 1 slower, greater than 1 faster)

Returns:

The floating-point array of values of the sampled (22-kHz) speech

A.6.5 FS2MBMService

This class sets up the Android service for the FastSpeech2 with MBM Vocoder Shareable Components service. It inherits from the Shareable Components base class, which does the bulk of the work for setting up the service. It requires and communicates with the IncomingMessageHandler class, which handles all the service requests.

This class manages a list of configurations that it locates in the folder “Shareable Components/tts-fs2mbm” on the Android device. Client applications can query this service for a list of configuration languages that this service can process.

Incoming requests to this service from bound clients need to follow specific guidelines. See the Shareable Components library shareable-configuration and the two classes Actions and Parameters for full details, as well as the API documentation.

- Method Detail
- `addConfiguration`

```
protected void addConfiguration  
(java.io.InputStream stream,  
java.lang.String directory,  
java.lang.StringBuilder sbWarnings)
```

Adds a configuration to the service’s list of configurations. The stream passed into this method can be read and deserialized into a Configuration class.

Usually this stream is from an opened YAML file. If this method has problems reading the file, error messages can be appended to sbWarning. Since the service reads all folders to find configuration languages, a failure here is only a warning for this language, not a complete failure for all configurations.

This class maintains a Hashtable of configurations matching the uuid with a folder where the language is installed.

Specified by:

addConfiguration in
class mil.army.arl.shareable.ServiceBase

Parameters:

stream - The opened stream to the config.yml file

directory - The directory that the configuration was found

sbWarnings - Messages about failure to parse the config.yml file are appended to this object

- clearConfigurations

```
protected void clearConfigurations()
```

Clears the list of configurations.

Specified by:

clearConfigurations in
class mil.army.arl.shareable.ServiceBase

- getConfiguration

```
public mil.army.arl.shareable.tts.Configuration getConfiguration(java.lang.String uuid)
```

Returns the Configuration object associated with the specified uuid. The Configuration object returned is specific to TTS and contains information necessary for this service to process the request.

Specified by:

getConfiguration in
class mil.army.arl.shareable.ServiceBase

Parameters:

uuid - The unique identifier to search for

Returns:

A Configuration object that matches the uuid

- getConfigurations

```
public java.lang.String getConfigurations()
```

Returns the whole list of configurations serialized into a string. On startup, the service reads the folder "Shareable Components/tts-fs2mbm" for all

subfolders that have language configurations for the FastSpeech2 MBM service. These Configurations are saved on this object and returned as a JSON-serialized string to any client that requests it.

Specified by:

`getConfigurations` in
`class mil.army.arl.shareable.ServiceBase`

Returns:

A JSON-serialized string containing all configurations found by this service

- `getMessenger`

```
protected android.os.Messenger getMessenger()
```

Returns the Messenger that is used by the service to handle incoming requests.

Specified by:

`getMessenger` in `class mil.army.arl.shareable.ServiceBase`

Returns:

The messenger that uses the IncomingMessageHandler

- `getName`

```
public java.lang.String getName()
```

Gets the name of this service for use in error messages and logging. The word “Service” gets appended, so it is not necessary to include it here.

Specified by:

`getName` in `class mil.army.arl.shareable.ServiceBase`

Returns:

The name of this service

- `getRootFolder`

```
protected java.lang.String getRootFolder()
```

Gets the root folder where this service should look for language-set configurations.

Specified by:

`getRootFolder` in `class mil.army.arl.shareable.ServiceBase`

Returns:

A directory

A.6.6 IncomingMessageHandler

This class handles all requests to the Shareable Components FS2MBM Service sent by client applications. It inherits from a common Shareable Components base

class for message handlers, which sets up the link to the parent service and does some of the processing for configurations.

This class has code to process *execute* requests to run the FastSpeech2 synthesizer and MBM vocoder on an incoming string of text. The output produced is an array of floating point integers ranging from -1 to 1 of a waveform that has a sampling rate of 22 kHz (22050).

There is a method on the shareable-utilities Utils class that can convert this array into a byte array of integers encoded in PCM16 and to also change the sampling rate if desired. That utility library can also save the wave-form array into a WAV file.

- Constructor Detail

- IncomingMessageHandler

```
public IncomingMessageHandler(FS2MBMService service)
```

Creates the handler to deal with incoming request messages. Takes a reference to the service so it can call methods on it to get configurations.

Parameters:

service - The FS2MBMService object that holds language configurations

- Method Detail

- handleExecuteRequest

```
protected void handleExecuteRequest  
(android.os.Message msg)
```

Does the main work of handling incoming TTS requests. The message received by the service will contain a bundle of key-value pairs of data. The keys this handler expects are outlined in the shareable-configuration library found in the Actions and Parameters objects as static properties.

The FastSpeech2 engine operates on a text-string input passed as an input parameter. If the client application specified an input file already on the filesystem, the handler will open and read that file line by line, appending newline characters as needed.

If the input text is too large (500 kB), the client should send them in a series of smaller chunks. This method is equipped to deal with reassembling the chunks before sending it to the TTS.

Specified by:

`handleExecuteRequest` in
`class mil.army.arl.shareable.MessageHandlerBase`

Parameters:

msg - The incoming request to execute the TTS

See Also:
Actions, Parameters

A.6.7 MBMelGan

This class is responsible for running the MBM vocoder module using TF-Lite.

- Constructor Detail

- MBMelGan

```
public MBMelGan(java.lang.String modulePath)
```

Constructs this class with the specified TF-Lite model file.

Parameters:

`modulePath` - The fully qualified path and name of the TF-Lite model file

- Method Detail

- getAudio

```
public float[] getAudio  
(org.tensorflow.lite.support.tensorbuffer.TensorBuffer  
input)
```

Creates the waveform data from the tensor specified, which will be the output of the FastSpeech2 module.

Parameters:

`input` - The output of the FastSpeech2 TF-Lite module

Returns:

A floating-point array representing the sampled speech output at 22 kHz

A.6.8 NumberNorm

Number Normalization Class for Arabic TTS. This class normalizes numbers found in text to improve the quality and uniformness of the synthesized speech using ArToDigits class.

- Constructor Detail

- NumberNorm

```
public NumberNorm(java.lang.String text)
```

Creates the class with the string to be normalized.

- Method Detail

- getOutputTest

```
public java.lang.String getOutputTest()
```

Returns the string with the numbers normalized.

A.6.9 Processor

This class processes the text before running the TTS to normalize it in a number of ways. It normalizes numbers, money, among other things that can be represented in text through NumberNorm and ArDigitsToAlph classes in different ways. They need to be consistent with the data that was used to train the TensorFlow model files.

- Constructor Detail

- Processor

```
public Processor(java.lang.String mapper)
```

Constructs this class and takes the fully qualified path to a mapper JSON file that is included with this LS's model files.

Parameters:

mapper - A JSON-encoded file of symbol text tokens mapped to integers

- Method Detail

- loadMapper

```
public void loadMapper(java.lang.String fileName)
```

This method loads the mapper.json file specified to read in all of the symbol tokens mapped to integer ids.

Parameters:

fileName - The fully qualified path and name of the mapper.json file

- textToIds

```
public int[] textToIds(java.lang.String text)
```

Converts a string of text into an array of numbers.

Parameters:

text - The original text

Returns:

The same text represented as numbers

A.6.10 TTSTDecoder

This class runs the workflow for processing TTS. It runs the processor, the fastspeech2 module, and then the vocoder. The run method is intended to run in its own thread.

- Constructor Detail

- TTSDecoder

```
public TTSDecoder(java.lang.String fastspeech,  
java.lang.String vocoder,      java.lang.String mapper,  
java.lang.String inputText, float speed)
```

The constructor takes a number of parameters to indicate the locations of the tf-lite models for FastSpeech2, the MBM vocoder, the mapping configuration file, and the input text to work with.

Parameters:

fastspeech - The fully qualified path and name of the FastSpeech2 tf-lite model file

vocoder - The fully qualified path and name of the MBM vocoder tf-lite model file

mapper - The fully qualified path and name of the mapper.json file

inputText - The text to synthesize

speed - The speech synthesis speed (1 is normal, less than 1 slower, greater than 1 faster)

- Method Detail

- getErrorMessage

```
public java.lang.String getErrorMessage()
```

Gets any error messages that were generated by FS2MBM while processing. Exceptions are caught by the bridge and then saved as messages on the bridge that can be retrieved by the caller to return meaningful information back to the client.

Returns:

An error message that occurred during FS2MBM processing

- getTTSOutput

```
public float[] getTTSOutput()
```

Gets the resulting floating-point array of values for the waveform produced as a result of the TTS operation.

- run

```
public void run()
```

Does the actual work of synthesizing from the text input using the values specified when constructing this class.

Specified by:

run in interface `java.lang.Runnable`

List of Symbols, Abbreviations, and Acronyms

API	application programming interface
ASR	automatic speech recognition
ID	identification
JSON	JavaScript Object Notation
LS	Language Set
MT	machine translation
OpenNMT	Open Neural Machine Translation
TF-Lite	TensorFlow Lite
TTS	text-to-speech
ULTRA	Universal Language Translator
UUID	universally unique identifier

1 DEFENSE TECHNICAL
(PDF) INFORMATION CTR
DTIC OCA

1 DEVCOM ARL
(PDF) FCDD RLD DCI
TECH LIB

3 DEVCOM ARL
(PDF) FCDD RLC IB
C SCHLESIGER
S LAROCCA
L HERNANDEZ