Cognitive Robotics

Final Project – A.Y. 2021/22

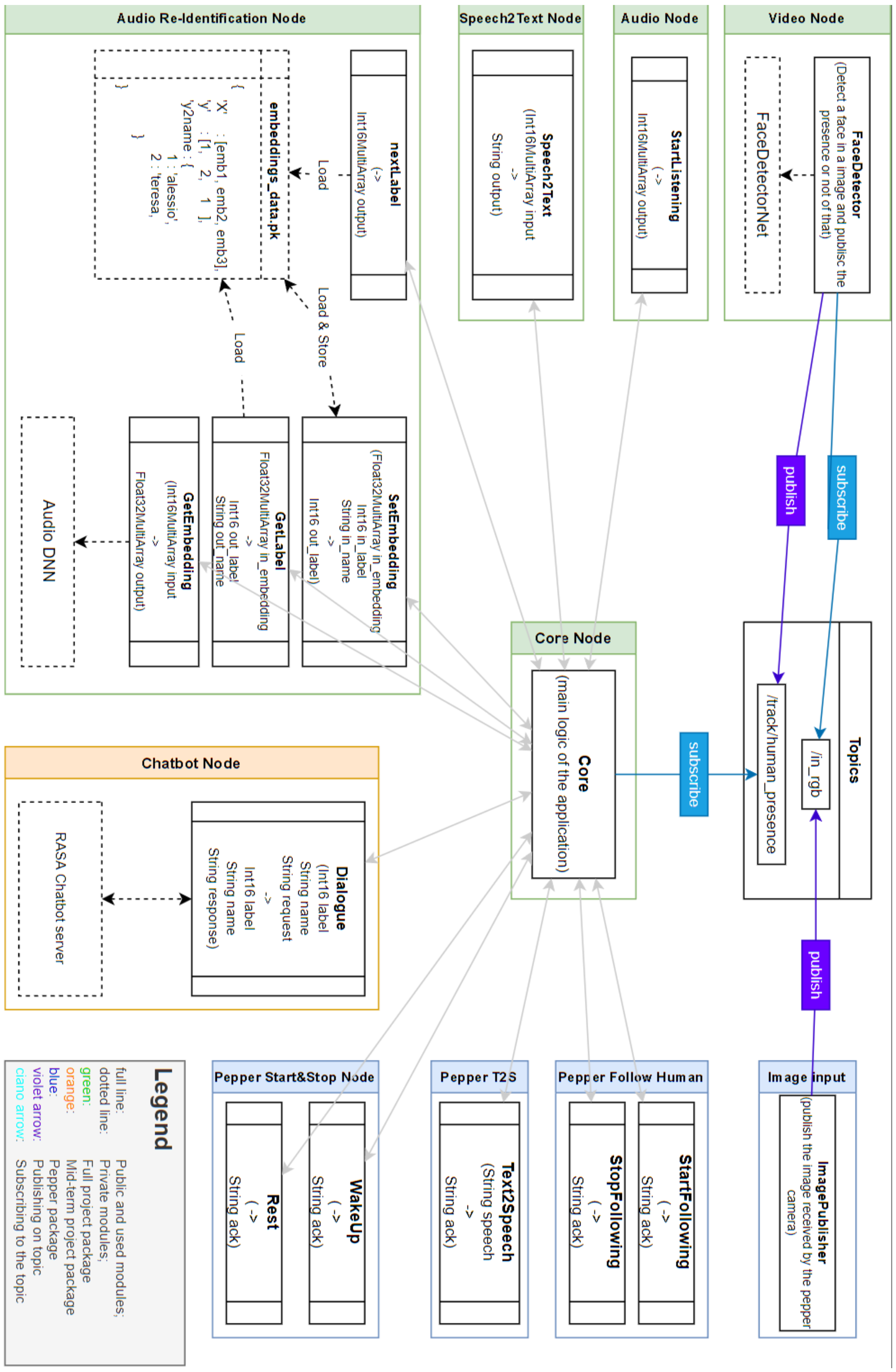
Alessio Pepe	0622701463	a.pepe108@studenti.unisa.it
Alfonso Comentale	0622701438	a.comentale9@studenti.unisa.it
Giuseppina Di Paolo	0622701510	g.dipaolo5@studenti.unisa.it
Teresa Tortorella	0622701507	t.tortorella3@studenti.unisa.it

Index

1 WP1 & WP2 - Architecture 2
1.1 Final Project Package 3
1.1.1 Core node 3
1.1.2 Audio node 3
1.1.3 Speech to Text node 4
1.1.4 Video Node 4
1.1.5 Re-Identification node 4
1.1.6 Chatbot package 5
1.2 Pepper package 6
1.2.1 Text to speech 6
1.2.2 Start and Stop 6
1.2.3 Image publisher 7
1.2.4 Human following 7
2 WP3 – RASA Integration 7
2.1 Chatbot new feature 7
2.2 Integration with ROS 7
2.2.1 dialogue-server Service 7
3 WP4 – Re-Identification 7
3.1 Services 8
3.1.1 getEmbedding Service 8
3.1.2 getLabel Service 8
3.1.3 setEmbedding Service 8
3.2 How to manage identities 8
3.3 Performances 8
3.4 Pepper speaking issues 8
4 WP5 & WP6 - Tests 8
4.1 Unit tests 9
4.1.1 Re-identification node 9
4.1.2 Audio node 9
4.1.3 Video node 9
4.1.4 Pepper modules 10
4.2 Integration test 10
4.2.1 Ros and Rasa 10
4.2.2 Core logic (no chatbot, no Pepper) 10
4.2.3 Core logic (no Pepper) 10
4.2.4 Final test 10
5 Conclusion 11
6 Notes 11

1 WP1 & WP2 - Architecture

This section shows the application architecture and explains all its parts.



Audio Re-Identification Node
nextLabel (-> Int16MultiArray output)
Load
embeddings_data.pk
{ 'X' : [emb1, emb2, emb3], 'y' : [1, 2, 1], 'y2name' : { 1 : 'alessio', 2 : 'teresa', } }
Load & Store
Load
SetEmbedding (Float32MultiArray in_embedding Int16 in_label String in_name -> Int16 out_label)
GetLabel Float32MultiArray in_embedding -> Int16 out_label String out_name
GetEmbedding (Int16MultiArray input -> Float32MultiArray output)
Audio DNN
Speech2Text Node
Speech2Text (Int16MultiArray input -> String output)
Audio Node
StartListening (-> Int16MultiArray output)
Video Node
FaceDetector (Detect a face in a image and publisc the presence or not of that)
FaceDetectorNet
publish
subscribe
Core Node
Core (main logic of the application)
subscribe
Topics
/in_rgb
/track/human_presence
publish
Chatbot Node
Dialogue (Int16 label String name String request -> Int16 label String name String response)
RASA Chatbot server
Legend
full line: Public and used modules;
dotted line: Private modules;
green: Full project package
orange: Mid-term project package
blue: Pepper package
violet arrow: Publishing on topic
ciano arrow: Subscribing to the topic
Pepper Start&Stop Node
WakeUp (-> String ack)
Rest (-> String ack)
Pepper T2S
Text2Speech (String speech -> String ack)
Pepper Follow Human
StartFollowing (-> String ack)
StopFollowing (-> String ack)
Image input
ImagePublisher (publish the image received by the pepper camera)

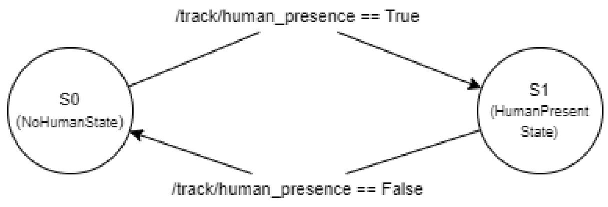
1.1 Final Project Package

In this section, we will explain the nodes entirely implemented by us to integrate the required services.

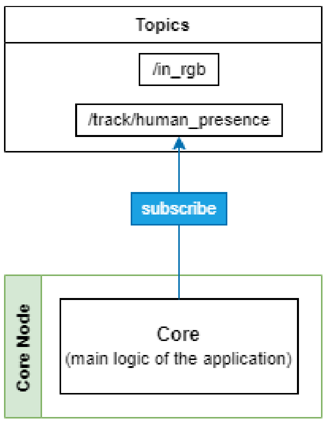
1.1.1 Core node

The Core application implements all the integration logic described in the following chapters. That has the following requirements:

1. This module must change its state if a person is present. The states are 2: one state with no person in which Pepper does not listen and speak; one state in which a person is present in front of Pepper, who listens and speaks.



   1.1. This module must enable audio detection if a person is present.
   1.2. This module must disable audio detection if a person is not present.
2. If a person is present and speaks (second state), the audio must be used to obtain the text answer (to use the chatbot), to identify (if known) the person using the re-identification module and provide the answer by the Pepper TTS.



We implemented this node in a thread-safe way due this node has some shared resources. In particular, the presence or not of a person is read by both functions, so we synchronized that with a mutex.

This node makes all the service call of our program, and some of them are repeated in an abbreviated time. For this reason, we used a persistent service proxy for the most called services (we provided some functions to make the code more straightforward).

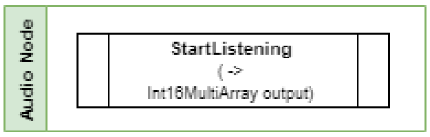
Changing some parameters in the config.py file, we can use the core logic without using Pepper and the chatbot to test that part individually. We simulated the chatbot response from the input line, the video stream from our webcam and the TTS with gTTS.

1.1.2 Audio node

The Audio Node was implemented in audio_detected.py. That module provides a service that starts the audio detection and return the listened audio when acquired.

That has the following requirements:

3. Detect the audio in a controlled environment (with silence).
4. Provide a starting service who return the detected audio.

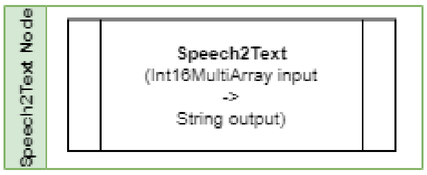


Both requirements were implemented in the same file as a choice because we need to keep the state of the microphone to start that without generating conflicts.

We implemented this node in a thread-safe way due this node uses some shared resources.

1.1.3 Speech to Text node

The Speech to Text was implemented in speech2text.py. That provides a service that inputs a raw audio wave and outputs a text contained in that audio. That is the only requirement of this module.



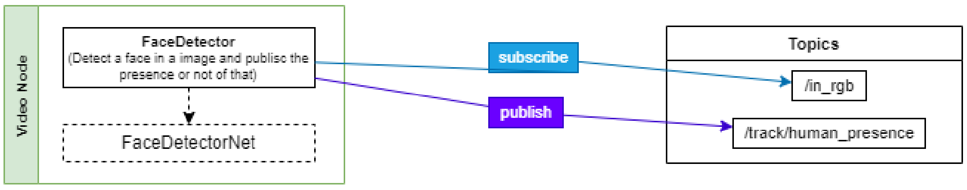
That module was implemented as an interface of the google speech services to simplify the call.

1.1.4 Video Node

The Video Node was implemented in video_detector.py. That has a module that subscribes to a topic in which the video frame will be published and uses a DNN to detect if a person is present. Considering the frame as a stream (because it is), this module publishes a Boolean that tells if a person is present in front of the camera.

That has the following requirements:

5. The module must understand if a person is present in the video stream.
    5.1. The module must manage the ghost situation.



That module was implemented separately from the camera source node to have the possibility to test that using an auxiliary node with another camera (and not just with the pepper camera). We do not need information about the detected people, just their presence. We maintained the state for a certain number of frames with the opposite state before changing that to avoid a ghost situation.

We implemented this node in a thread-safe way due this node has shared resources.

The detection was made with a network that has good performances with masked people, but the tracking service on Pepper (which allows us to track the face of the person in front of Pepper) is not so good with masked faces. So, we can say that this part works just without masks.

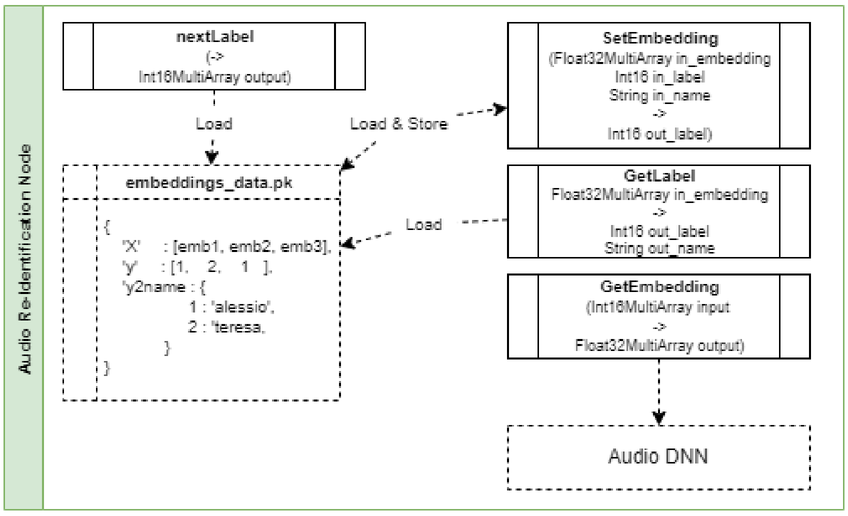
1.1.5 Re-Identification node

The Re-Identification was implemented in reidentification.py. That provides different services: a service to get the embedding of audio (using a DNN), a service to set an embedding for one identity, a service to get the identity of a person (if known) from a new embedding, and a service to get the following unused label.

That has the following requirements:

6. The module must identify known people from an audio waveform.

7. The module must understand if a person is unknown.
8. The module must construct its people database dynamically.
9. The data of the module need to be persistent.



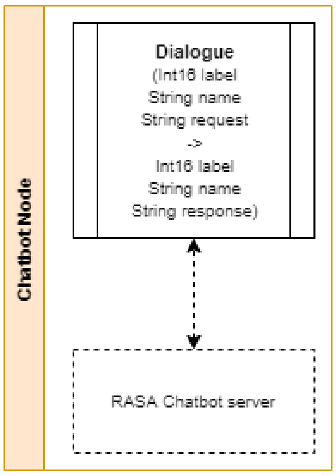
We implemented it all in the same node. That is because we need to keep the state of all embeddings in all services. Also, there are coherent in a logical way. We saved all the data we had in a pickle file to maintain the persistence of the data. The services will read and write on that data. The embedding will be generated from an audio waveform using a DNN. We keep a label for each person (more than the name) to avoid exchanging two people with the same name.

We implemented this node in a thread-safe way due this node has multiple callbacks, which uses some shared resources (for example, the file).

1.1.6 Chatbot package

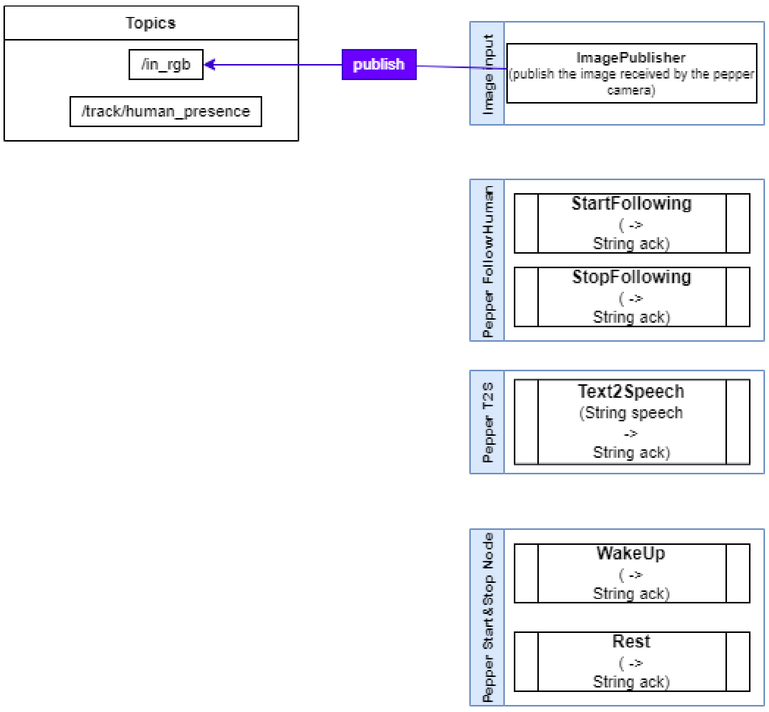
The Chatbot Node was implemented in dialogue_server.py. That provides a service to pass the question to the bot (and the identity) and receive the answer. A better explanation of that module will be provided in the next WP. That has the following requirements.

10. A name and a quantity identify each element in the list.
11. Once recognized the identity of the user, the robot must allow him/her to:
    11.1. List the items within his/her shopping list
    11.2. Insert a new item in his/her shopping list
    11.3. Remove an item from his/her shopping list
    11.4. Empty his/her shopping list
12. It must provide an interface to communicate with the chatbot from ROS.



1.2 Pepper package

The pepper package is represented in blue in the architecture image. We modified the node provided to us to simplify the integration. We also wrote a new module (Following Human) to improve the pepper interaction with the person.



These methods are implemented using NaoQi as a proxy to the Pepper function. Most of these nodes can leave Pepper in an incoherent state, so they have a shutdown callback to avoid this issue (for example, the Start and stop node has a shutdown callback which puts Pepper in a rest position).

1.2.1 Text to speech

Text to Speech uses the Pepper audio to make Pepper speak. This is an adapter of NaoQi function ALTextToSpeach.say. We provide that module as a service.

1.2.2 Start and Stop

Start & Stop is used to simplify the action of wake-up and rest Pepper. We modified this module to remove the wake-up call at the beginning of the module.

1.2.3 Image publisher
Image Publisher is used to receive Pepper's video stream. This module is implemented as a publisher which acquires the images from ALVideo and publishes that on a topic.

1.2.4 Human following
Human Following provides two services: one to start following people with the pepper head and the other to stop that movement. We provided those services as an interface of the Pepper module ALBasicAwarness. This service is beneficial in our application because if Pepper looks at us, we move in front of it without having problems. Unfortunately, this module does not work very well with masked people.

2 WP3 – RASA Integration

2.1 Chatbot new feature
From the chatbot architecture of the midterm test (link to chatbot report[1]), only the possibility to empty the list, previously not present, has been added.

In addition, the ability to manage multiple identities has been added. This improvement was realized by inserting a new parameter to get by the chatbot form: the name of the people (also the person id from re-identification will be stored in a slot but does not influence the conversation; it is needed to run the bot action).

2.2 Integration with ROS
To integrate the RASA server with the re-identification module, we needed to pass a parameter (identity) from the outside of the chatbot and, after the conversational round, read the identity parameters inside the chatbot from the external.

To do that, we implemented an interface as a ROS node. That interface provides a service that inputs the question string, an identity (label and name). It receives the answer string and the identity (label and name) inside the chatbot after the answer.

We made that making a POST request to the rasa server specifying the slot to fill to synchronize the status of the re-identification with the RASA status. After the message will be sent to the rasa server with a POST request. Then the answer was provided, the service made a GET request to obtain the actual identity inside the RASA server. We made that because the chatbot will acquire the person's name if it is unknown.

The label will be passed from ROS, and that ID will be used to access the correct shopping list depending on what person was in front of Pepper. This because it allows us to manage people with the same name. We always send the following free id because there is a case in which an unknown person asks for something and give the name in the last round of conversation (in this case, in our application flow, the chatbot will not have the information required if we do not pass next label).

2.2.1 dialogue-server Service
The integration as we talked about before was implemented by the ROS service dialogue_server. This service inputs the three parameters (text, id, and name) and outputs the updated three parameters.

3 WP4 – Re-Identification
We made the re-identification using the audio wave. We decided that because we would still need the audio for talking with the robot. Also, the microphone has particularly decent quality, instead of the pepper camera with no comparable quality.

[1] Extended link: https://drive.google.com/file/d/1mBsvdBMM4Qh7HSYwsYAoSkcyyv_YnrGZ/view?usp=sharing

We use the re-identification module that we have explained in the first chapter.

3.1 Services

3.1.1 getEmbedding Service

All the process of re-identification works with the audio embeddings. These will be obtained with this service, that takes in input an audio waveform and outputs a feature embedding representing that audio.

This is made transforming the audio in a Mel Spectrogram. After the Spectrogram will be passed at a DNN able to generate very discriminative audio feature embeddings.

3.1.2 getLabel Service

When the get label service was called, a new embedding was provided. The new embedding will be compared with all the saved embeddings with the cosine similarity metrics. If the probability of belonging to the identity with less distance is greater than a fixed threshold, the identity will be returned. Otherwise, the unknown label will be returned.

3.1.3 setEmbedding Service

This service associates a label and a name to one embedding. Those parameters will be stored in a file with all the other saved embeddings to make the data persistent.

3.2 How to manage identities

The state of the person identity will be kept by the core, who, received an audio, use the embedding service. After using that module, use the get label service to get a label if the person is already known. If the person is not known, the core keeps the embedding in a buffer waiting for the conversation to go on, and the chatbot returns a name. When a name is provided (or immediately, if the person is already known), the embedding will be set associated with a new label in the database. With this flow, we can build the database dynamically.

3.3 Performances

The highest cost in terms of time and compute efforts is in get embedding service, and getLabel service. Unfortunately, we need to compute the embeddings of all audios that contain voice. Fortunately, we used this module with a video detector to keep the identity after getting that, until the person leaves the scene. This improvement made the application faster because avoid computing the label each time.

3.4 Pepper speaking issues

The environment is supposed to be quiet. However, Pepper has to Speech, so the microphone listens to the pepper speech audio and messes up the re-identification in the best case. The core enters a loop where Pepper's Speech has listened in the worst case.

To avoid that problem, we stop the audio acquisition before Pepper starts speaking and restart when Pepper ends speaking. We made that using a synchronous listen service.

Furthermore, there is a particular case of interest for this description. When there is no person in front of the robot, it stops listening and only resumes when a person is present. This improvement is an optimization in terms of resources used. Furthermore, knowing the presence status of a person or not allows us to reset the chatbot scene so as not to have strange behaviours of this. In this regard, the video detector and the following node have been implemented. Also, if no one is in front of Pepper, keeping the environment controlled (silent) is no need.

4 WP5 & WP6 - Tests

We have used a bottom-up strategy to develop tests: we started from unit tests and then developed integration tests.

4.1 Unit tests

4.1.1 Re-identification node

We have developed 7 test cases for this unit. We have a set of samples used for training and another set for tests for each test.

The samples for each set are structured in the following way. In the following table, we indicate "training set samples/test set samples" for each identity for each test case. Note that there is audio from unknown people who are not in training set in all test cases. This audio was recorded using the Pepper microphone to ensure that the module works on Pepper, so it is simpler to demonstrate that the re-identification works.

	IDENTITY 1	IDENTITY 2	IDENTITY 3	IDENTITY 4	UNKNOWN
CASE 1	3/2	0/0	0/0	0/0	0/3
CASE 2	3/2	3/2	0/0	0/0	0/3
CASE 3	5/2	5/3	5/2	0/0	0/3
CASE 4	5/2	5/2	3/2	0/0	0/3
CASE 5	5/2	5/2	5/2	0/0	0/3
CASE 6	5/2	5/2	5/2	3/2	0/4
CASE 7	4/2	4/2	4/2	3/2	0/5

The purpose of this test was to demonstrate that requirements have been respected. The number of samples is not low because we save 6 embeddings for each person to not overfit on an identity with a higher number of samples in our implementation.

This test can be launched with the test_reidentification.launch file inside the fp_audio package and works for all cases.

4.1.2 Audio node

Since this module is a sort of "adapter" for wrapping the google speech recognition model, we have chosen not to develop tests for this unit. The functionality can be tested using the core, also without Pepper and the chatbot.

4.1.3 Video node

We initially tried the detector starting the image publisher node to test the video detection, which acquired video from our webcam and the video detection node. We subscribed manually to the topic to see if it worked properly. After that, we saved the video stream from Pepper and used that to provide some test cases that prove the functionality of the requirements. This second part allows us to say that this module works also combined with the ALTracking and so on Pepper.

The test cases are projected as:

1. The video stream contains a masked person who enters the scene and then leaves the scene for the first case. In this test case, Pepper's head is moving, producing different ghost frames. We expect an output like True, False.
2. The second test case is like the first, but the person unmasks himself during the stream. In addition, there are more ghost's situations. We expect an output like True, False.
3. In this case, the person enters and leaves the scene two times, so we expect an output like True, False, True, False.
4. This case has a person already in the scene and then left. Also, in this test, there are a lot of ghost's situations. We expect an output like True, False.
5. In this case, there is not a person, and some ghost's situations occur. We do not expect any output because the state does not change.

6. There is only one-person masked face in this last case, so we expect just an output of True

These tests can be launched with “test_detector.launch” in the fp_audio package. All these tests have passed.

4.1.4 Pepper modules

We have launched the “pepper.launch” file in the pepper_nodes package to try these modules. After that, we called the provided services with the command *rosservices call [service_name] [parameters]*.

4.2 Integration test

4.2.1 Ros and Rasa

To make an integration test and try the Ros interface for Rasa, we used an interface from the command line, which takes in input ID, name and the text question and answers with the new ID, the new name, and the answer. This interface uses the service we used in the main application logic and can be tried using the launch file dialogue.xml inside the ros_chatbot package.

We cannot provide any result because it is just an adapter for the rasa chatbot. The results of the chatbot were provided in the midterm report.

4.2.2 Core logic (no chatbot, no Pepper)

This integration test integrates video detection with the core logic, audio services, and re-identification modules. That can be launched setting in the config.py file CHATBOT_RUNNING=False and ON_PEPPER=False and launching the core_s.launch file inside the fp_audio package.

The flow of this test is the same flow of the complete application, but the answer from the chatbot was asked by the input line in the console and all Pepper services are not called.

We cannot provide any result, but we can ensure in our test that the work is correct (but the re-identification does not work well with our microphones because they have been tuned on the Pepper microphone).

4.2.3 Core logic (no Pepper)

This integration test is remarkably like the last one. To launch that, you must set CHATBOT_RUNNING=True and ON_PEPPER=False. You can try all the main logic with these tests, including the chatbot, which is not simulated anymore. The Pepper services are not called.

4.2.4 Final test

To make the final test, we need to set CHATBOT_RUNNING=True and ON_PEPPER=True and launch all.launch or both chatbot.launch, pepper.launch and core.launch. Now, the application works completely. We provide some videos at the following link[2]. representing the following test cases:

1.1 A person enters the scene and has a talk. Then he/she exits.
1.2 The same person enters again in the scene and checks the status.
2.1 A new person enters the scene and has a talk, checking for the status.
2.2 Same of the second test.
2.3 Same of the second test.
3 All people enter again inside the scene and check for the status.

This test aims to try the re-identification, the tracking, the audio, the video, and the chatbot live to test their interaction and not their working because these results are already provided.

[2] Extended link: https://youtu.be/UyyMqou_CBE

5 Conclusion

We designed the RASA chatbot in the midterm test. We have designed the general architecture of the application and explained all its parts. We have described the integration choices between the two parts and the design choices of the re-identification module. Finally, we have documented the tests that demonstrate the actual operation.

The results obtained in unit testing are good for the situations we have foreseen, and which are foreseen by the requirements. The chatbot is quite robust to the speech to text module (not supplemented by us). In the presence of silence, these errors are reduced, and consistent results are almost always obtained.

6 Notes

The source code of the described project is available on GitHub at the link[3]. The instruction to run the project and the tests are present in the README.md file.

[3] Extended link: https://github.com/pepealessio/cr_pepper_shoppingbot