NAO Watchdog

Jessel Serrano

Final Draft

06 December 2013

CNT 4104 Software Project in Computer Networks

Instructor: Dr. Janusz Zalewski

Computer Science & Software Engineering Programs

Florida Gulf Coast University

Ft. Myers, FL 33965

1. Introduction

Security is becoming a major issue in modern technology. Both physical security and cyber security constantly need to be updated to keep up with the progressing pace of technology and society. This project report focuses on the NAO robot [1] as a dynamic security camera. In a world where security is becoming a major priority, the NAO robot can potentially provide a crucial security element through its integrated sensors. The robot (Figure 1) contains a plethora of sensors that are optimally designed to locate and track objects. What is even more integrative is the NAO's networking ability, as it allows the data it tracks to be uploaded to the web via its Wi-Fi module. This essentially allows the NAO robot to operate as a remote security camera that feeds live video across the Internet to a user in a different location. Not only is this security camera remote and can be accessed remotely, but it is a humanoid robot that can "watch" an object as it enters its view and keep constant track of the object. Such a technology no longer falls in the realm of an inanimate, automatically rotating camera, but rather constitutes an anthropomorphic 'watchdog.'

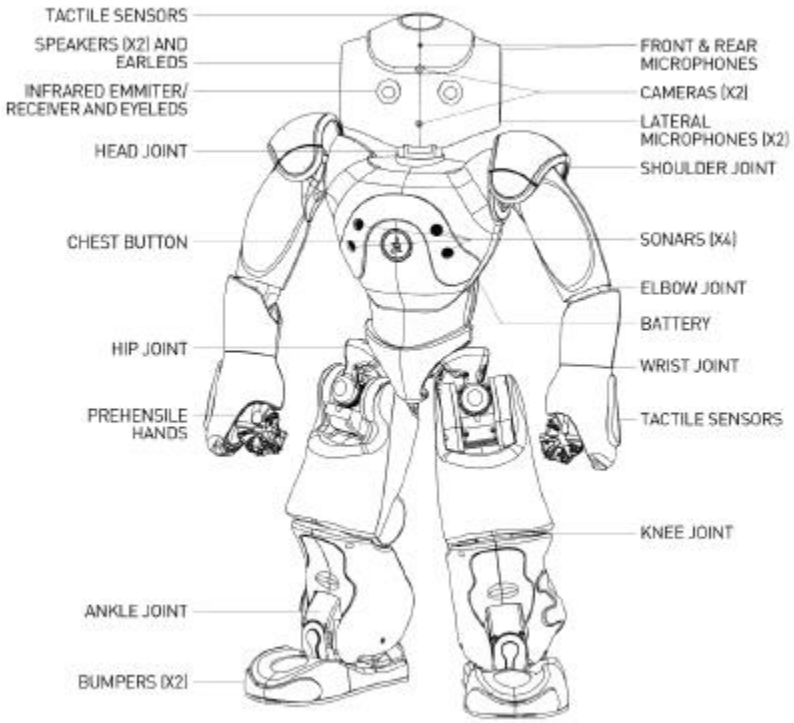

Figure 1: The NAO Humanoid Robot

2. Definition of the Problem

The problem addressed in this project is how to turn the NAO robot into a security Watchdog. This will be accomplished by first expanding on the previous project by Austin Hughes [3], in which he details how to get the NAO robot up and running and how to begin to program the robot. Then, requirements are set which the robot must meet in order to be considered a Watchdog. The initial requirements are listed below:

1. The NAO robot shall stream the video data across the Internet where a remote user can access the feed.
2. The NAO robot shall utilize its camera and other sensors to determine if an object is present in its view.
3. The NAO robot shall turn its head to keep objects and/or persons within its view.
4. Optionally, the NAO robot shall recognize objects and/or persons as they enter the robot's view.

In addition to the four main requirements set forth for this project, two networking requirements can also be set once the four main ones are met. They are as follows:

1. The NAO robot shall be controlled remotely in order to provide further security options.
2. The NAO robot shall be controlled remotely through a WebSocket user interface.

The steps to be taken to complete the four main requirements are as follows:

1. An application must be developed, using the NAOqi (Figure 2), which allows the accessing of the camera feed remotely. This is done by developing a program which makes several calls to the robot via the Internet. The robot will then return camera data.
2. Then, the NAO robot must be programmed to utilize its camera to sense moving objects, such as animals or persons. This is done using the NAOqi, which is the NAO's SDK.
3. The robot must then be programmed to move its head as necessary to keep the camera pointed towards the object it is 'watching.' This can be done using the NAOqi SDK.

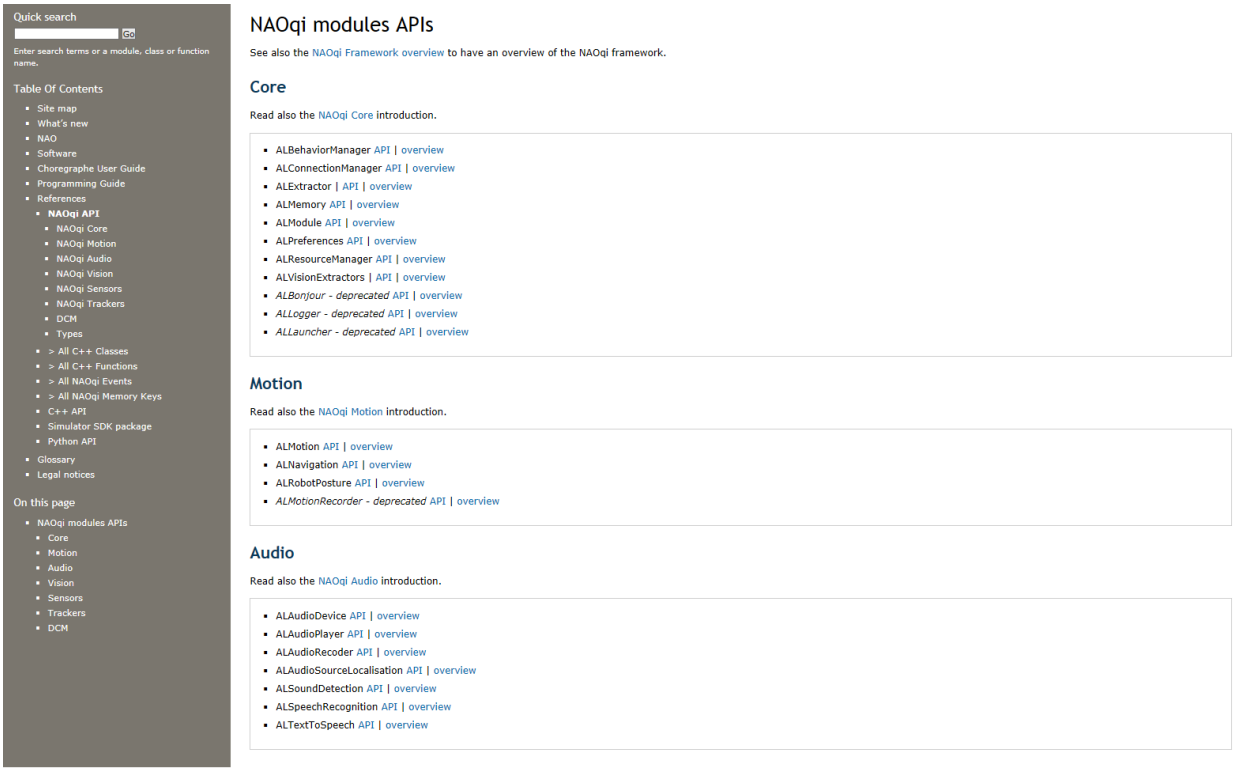

Figure 2: NAOqi SDK -

A few pre-existing methods and functions are in place from the NAOqi documentation API that are useful for this project:

- The NAOqi has an `AudioSourceLocalization` module that allows the NAO to detect sound and where it is coming from. This module is useful in tracking human movement via stepping and walking.
- The `ALVideoDevice` module controls the NAO's two camera devices on its head. By accessing these cameras, the NAO can take pictures and save them in its internal memory.
- The `ALMemory` module allows one to access the robot's internal memory using certain keys. With certain methods and keys, one can retrieve, insert, or alter data in the NAO robot.
- `ALMovementDetection` extractor enables to detect movements around the robot thanks to its camera. This method is used when no face is detected but there is movement in the NAO's field of view.

Furthermore, there is the `ALMotion` module that provides methods that facilitate movement for the NAO robot. For the purposes of this project, the head is used (where the twin cameras are located) to keep up with the objects detected. It contains four major groups of methods for controlling the:

- Joint stiffness (basically motor On-Off)
- Joint position (interpolation, reactive control)
- Walk (distance and velocity control, world position and so on)
- Robot effector in the Cartesian space (inverse kinematics, whole body constraints)

Once the four main requirements are successfully met, one could use WebSockets in order to create a web accessible and graphical NAO interface, previously created by Austin Hughes [3]. There are two components in creating this web interface:

1. Creating a server application
2. Building a WebSocket client with HTML and JavScript.

3. Design Solution

This project has many parts that require the 'Watchdog' to be complete. These parts include hardware and software alike. For hardware, there are:

- The NAO robot - Intel Atom @ 1.6 GHz, two HD 1280x960 cameras, Ethernet & Wi-Fi connections,
- A client computer.

The software side of this project utilizes some software that is native to the NAO robot and other components that are not. They include:

- NAOqi – the NAO robot's SDK,
- The actual Watchdog program – to be written in Java and compiled into a JAR file,
- Potentially WebSockets – which requires utilization of HTML and Javascript.

The completion of this project relies on the interaction of the NAO robot and the client computer, through a streaming of a live video feed from the NAO robot to the client. This is the objective of the design process. What lies underneath is the communication of the client and the NAO robot, with the client making calls to the NAO robot and the NAO robot returning data to the client through a network connection.

This communication is done via a Java program on the client side that uses the NAOqi to create proxies of the actual API that the NAO robot uses. These proxies include API from the `ALAudioSourceLocalizationProxy` and `ALMotionProxy::setPosition()`. The Source Localization API allows the robot to detect sound and where it is coming from. Combining this with the robot's motion API allows the robot to locate sound and look at the object emitting the sound. The robot then begins to send back live images of the object to the client computer. Below is an example of how to initialize the Sound Localization Proxy and subscribe the device.

```
ALAudioSourceLocalizationProxy locater = new
     ALAudioSourceLocalizationProxy(NAO_IP, NAO_Port);
locator.subscribe();
```

Once subscribed, the robot begins to collect sound data and determine where it is coming from through the use of Interaural Time Difference [1], a mechanic which uses the four microphones on the NAO robot to determine which direction sound comes from. Figure 3 demonstrates how this works. The data are then stored on the robot in the memory under the key-value

`ALAudioSourceLocalization/SoundLocated`. How the data are retrieved is shown in the code below using the Memory Proxy API:

```
ALMemoryProxy memory = new ALMemoryProxy(IP, Port);
memory.getData("ALAudioSourceLocalization/SoundLocated");
```

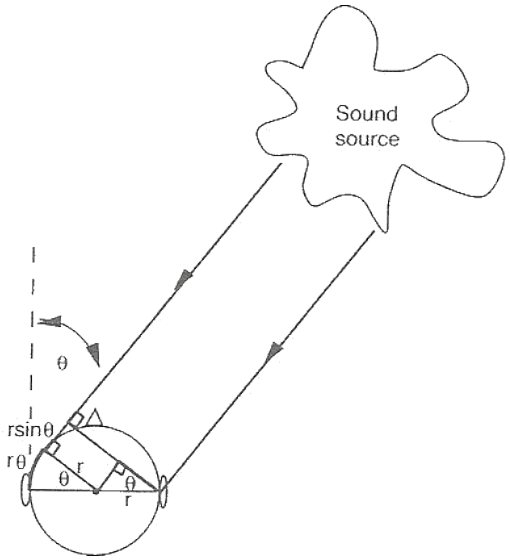

Figure 3: Interaural Time Difference used to locate sound

The data located in this memory are found to be in the format as follows [1]:

```
[ [time(sec), time(usec)],

  [azimuth(rad), elevation(rad), confidence],

  [Head Position[6D]]
]
```

The important data are the azimuth and elevation angles. Because this is a 2-dimensional array, the azimuth and elevation angle data (in radians) are stored in `array[1][0]` and `array[1][1]`, respectively. Using this array, one can find out where sound is coming from relative to the robot's head. By extracting the 1st element of this 2-D array, the two angles can be used to turn the robot's head in a certain position via the method `ALMotionProxy::setAngles()`. In this method, the first argument represents the body part to position and the second argument is the vector array of the position of the sound:

```
robot.setAngles("Head", array[1][0], 0.1f);
```

It should be noted that there are other methods to move the robot via its joints. These methods include `setPosition()` and `angleInterpolation()`. These methods, however, are blocking methods. This means the program is on pause until the robot finishes moving a certain body part. The `setAngles()` method is a non-blocking method. This means the

program will continue to run even while the robot is moving. This non-blocking method offers more fluidity and faster responses, as per the NAO documentation [1], yet each has its own pros and cons.

Once the robot's head is in position, recording can begin with the robot taking pictures and sending them back to the client. This is executed using the `ALVideoDevice` module. Using a method called `getImageRemote()`, a client is able to retrieve an image directly from the NAO camera. This can be done up to 30 pictures per second, giving a 'video' of 30 frames per second. This is the live video feed.

A diagram of how the NAO robot and the client computer interact is shown in Figure 4.

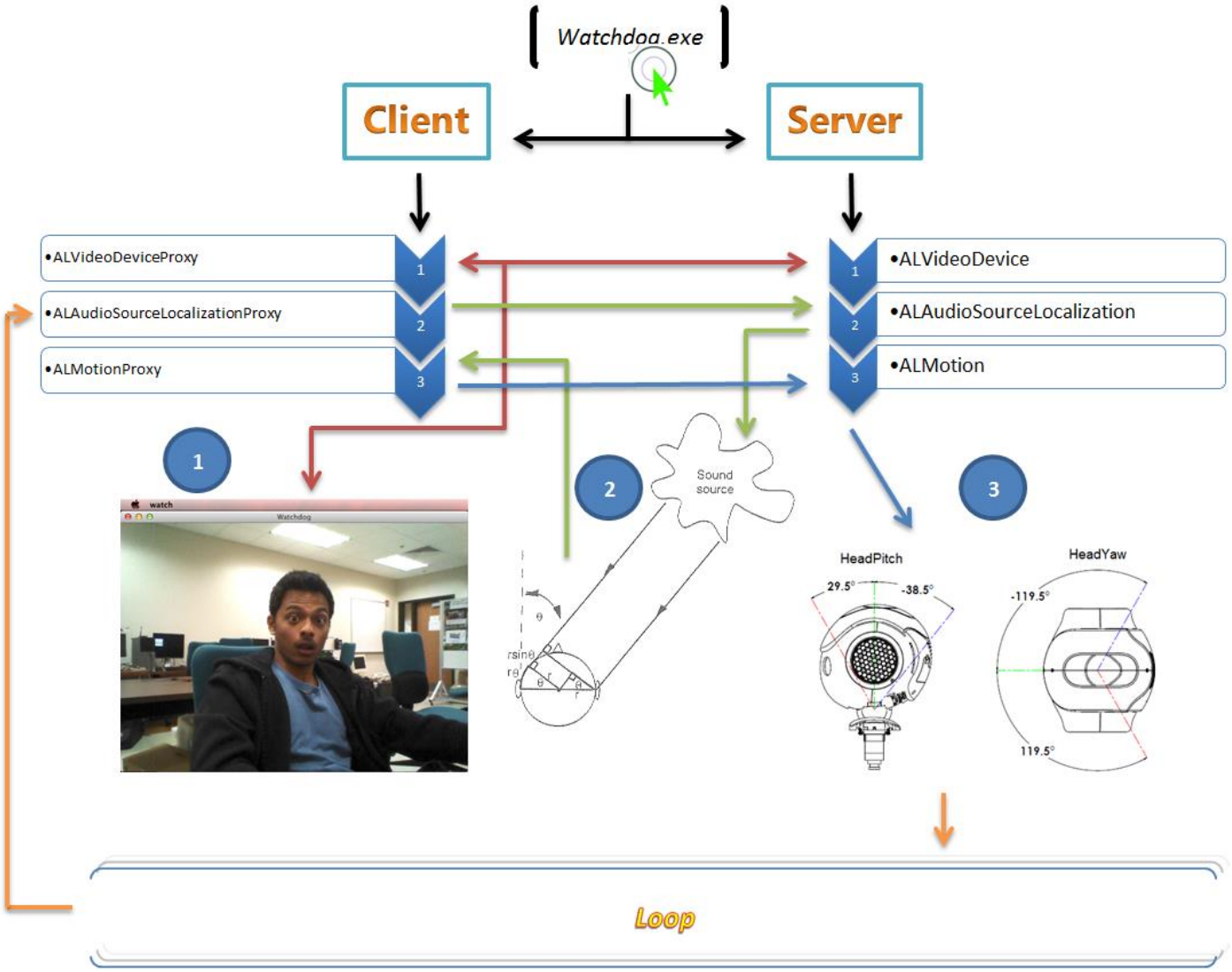

Figure 4: Watchdog Illustration – Design Diagram

4. Implementation

There are three basic parts of the Watchdog project that allow it to function:

1. Audio Source Localization,
2. Motion and Head Positioning,
3. Video Transmission.

4.1 Audio Source Localization

For this project, a method has been made to selectively determine the location of sound and retrieve the data of the direction of the sound from the robot's memory. The method which does that is shown below.

```
public static float[] soundLocalizer(ALAudioSourceLocalizationProxy
loc) {

      //Create the proxy for Memory
      ALMemoryProxy mem =
                  new ALMemoryProxy(NAO_IP, NAO_PORT);

      //Sets the audio sensibility to detect quiet or loud sounds
      loc.setParameter("Sensibility", new Variant(0.9f));

         //Store data in Variant type
      Variant soundData = new Variant(mem.getData(
                  ("ALAudioSourceLocalization/SoundLocated")));

      //Converts Variant data to float array,
      //just takes the azimuth and elevation angles
      float[] pos = soundData.getElement(1).toFloatArray();
      System.out.println("azimuth: " + pos[0] +
                  " | Elevation: " + pos[1]);
      return pos;
    }
```

This method returns a float array. Once the Sound Localizer Proxy is subscribed (not shown in this method), the robot begins to collect data on sound. It stores the data in the *ALAudioSourceLocalization/SoundLocated* key, in the form of an array (described in Section 3). Once the float array is obtained containing the azimuth and elevation angles, the position of the head can then be set so as to turn the head so that is faces the source of the sound, or the person making the sound. A respective call looks as shown below:

```
float [] headPosition = soundLocalizer();
```

4.2 Motion and Head Positioning

Once the float array containing the position of the object emitting sound has been stored into the float array called `headPosition`, the robot can then use this information to position its head accordingly. Doing this does not take much code or conversion of types; a proxy for the Motion API is created and then the method to set the position is called. This is shown below:

```
ALMotionProxy robot = new ALMotionProxy(NAO_IP, NAO_PORT);
robot.wakeUp();
robot.setPosition("Head", 2, headPosition, 1f, 63);
robot.rest();
```

The method `setAngles()` has three arguments which corresponds to different things. In the NAO documentation under motion classified as Cartesian [1], they are shown in the specification of the method:

```
void ALMotionProxy::setAngles(const AL::ALValue& names, const
AL::ALValue& angles, const float& fractionMaxSpeed)¶
Sets angles. This is a non-blocking call.

  Parameters:  • names – The name or names of joints, chains, "Body",
                    "JointActuators", "Joints" or "Actuators".
               • angles – One or more angles in radians
               • fractionMaxSpeed – The fraction of maximum speed to use
```

The chain name for this project includes the Head, which includes both HeadYaw and HeadPitch. HeadYaw and HeadPitch control the robot's Head position as up/down and left/right.

The `setAngles()` method can also accommodate lists or arrays of names, angles, and speeds as arguments. This allows the robot to move multiple joints at the same time in one method call. This is illustrated below in the following code:

```
robot.setAngles( new Variant( new String[]{"HeadYaw", "HeadPitch"}),
                      new Variant( new float[]{headPosition[0],
                           headPosition[1]}), 0.1f);
```

This one call tells the robot to yaw it Head however many azimuth radians and pitch its Head however many elevation radians. The speed has been set to the same for each movement.

Once the position of the head has been set to look at the source of the sound, the robot can then begin taking pictures and sending it back to the client.

4.3 Video Transmission

Instead of using the proxy for the Video Recorder on the NAO robot, the Video Device proxy is utilized instead. This is because the Video Recorder API records video and then stores it in the robot's memory. One would then have to go into the memory and retrieve the video. However, the video would not be real-time, due to the retrieving of the video and then the playback. The Video Device API, however, allows the instant retrieval of pictures taken from the NAO robot using the `getImageRemote()` method. This is shown below:

```
ALVideoDeviceProxy videoDevice = new ALVideoDeviceProxy(NAOQI_IP,
NAOQI_PORT);

videoDevice.subscribe("java", 1, 11, 250);
Variant ret = videoDevice.getImageRemote("java");
videoDevice.unsubscribe("java");

// Video device documentation explain that image is element 6
Variant imageV = ret.getElement(6);

// display image from byte array
byte[] binaryImage = imageV.toBinary();

new ShowImage(binaryImage);
```

Once the image is retrieved, it is stored in the Variant type and accessed through another Variant type variable. It is then converted to a binary array, which is passed to the method `ShowImage()`. `ShowImage()` takes the binary array and creates the image that the robot took. This image is shown in a window created by Java. Once this process is looped infinitely, the robot will constantly be taking pictures and displaying it on the client's window. This is equivalent to the live video feed of the object.

In order for the pictures to display onto the client screen, a graphics component is implemented using Java's graphics library, such as Swing. For this video feed, JFrame is used to create a window and then upload the picture into the window. In order to loop the program successfully, the image needs to be constantly refreshed in the JFrame window without constantly recreating the window. This is done using Java's `repaint()` method, which clears the current window and allows a new image to be displayed.

5. Experiments

Testing of the NAO Watchdog includes testing of each individual part (Audio Source Localizing, Head Positioning, and Video Transmission). Audio Source Localizing and Head Positioning are tested together as a unit in order to determine if the NAO can turn its head towards the source of sound successfully. Then, Video Transmission is tested to determine if the NAO can smoothly feed live images to the client computer. Once these parts are tested individually, it is all tested together to determine if the Watchdog successfully runs.

To run the whole program in two clicks, the client computer has to be connected to the network that the NAO robot is on. If necessary, the specific ports have to be open (9559) and the IP address has to be statically assigned to the NAO robot. After, double-clicking the Watchdog.jar file (shown in Figure 5), the NAO will start streaming video. The NAO will track sound by double-clicking the Watchdog_Track.jar file. When both files are run together, the NAO will become a dynamic security camera.

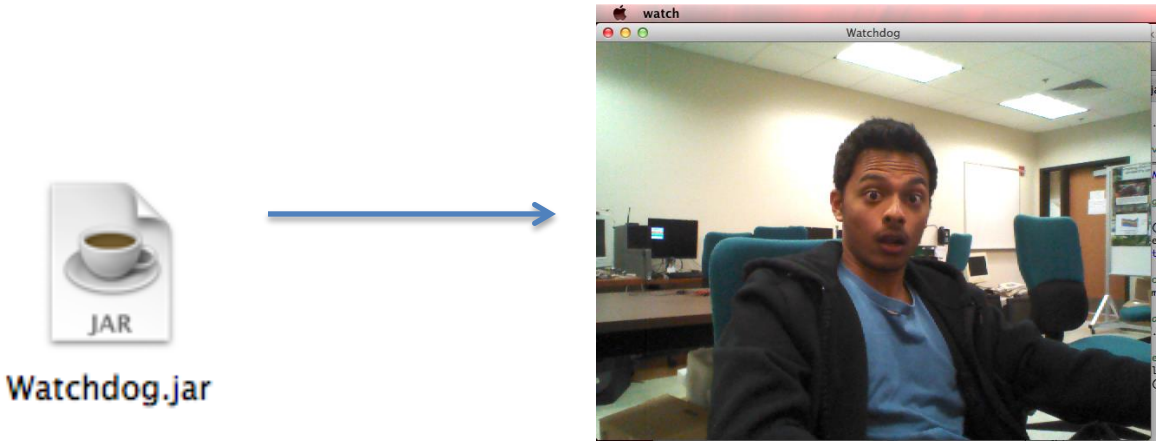

Figure 5: Running the Watchdog program

5.1 Audio Source Localizing & Head Positioning

In testing whether the NAO robot can turn its head towards the source of a sound, the Watchdog_Track.jar file is started. To start, double click the Watchdog_Track.jar file. When doing this in testing, however, the program ran without any way to stop it. To run the tracking program and stop it requires running the JAR file from the command prompt (or terminal for Macs).

To start the program in the command prompt or terminal, navigate to the directory where the JAR file is located. In the command prompt, type “java –jar Watchdog_Track.jar” without the quotes. In testing, doing this would sometimes give an error of Unsatisfied Link: Unable to

locate library path. To fix this, the jNAOqi library JAR file is added to the directory containing the Watchdog_Track.jar file. The libjnaoqi.dll file (libnaoqi.jnilib on Macs) should also be in the directory as well. These library files can be found on NAO's community website [1]. After, type into the command prompt:

```
java –jar Watchdog_Track.jar –D java.library.path=”jnaoqi”
```

This causes java to link the program to the NAO library, thereby allowing it to run. To close the program, on the keyboard press CTRL+C.

Once the program runs, it loops infinitely as the NAO constantly locates sound and mathematically determines the angle and elevation of the source of the sound. The program prints the azimuth angle and the angle of elevation to the screen on the client computer. The robot will then turn its head using the azimuth angle and the angle of elevation to face the source of the sound.

In this test, the source of the sound is the snap of fingers. When the sound is produced, the robot's head does not track the source very accurately. The head turns sparsely, and when it does it often overshoots the direction of the source. However, the robot's head does face the source directly sometimes.

In order to diagnose this issue, the sensitivity of the Source Localization Device can be changed from anywhere between 0 and 1, where 1 is the most sensitive sounds. Changing the sensibility is done by the following call.

```
locater.setParameter(“Sensibility”, new Variant(0.9f);
```

The sensitivity in this test is set at 0.9, or very sensitive. This is so the NAO can detect footsteps or finger snaps easily.

When the sensitivity is set higher, the robot responds more often to noises. It turns its head toward the source more often. However, it does not "follow" a sound smoothly, often jerking its head "to see where the sound went."

5.2 Video Transmission

In testing whether the NAO robot can feed live images continuously first requires starting Watchdog.jar file. Starting the Watchdog.jar file is done in the same way as in Section 5.2 with the Watchdog_Track.jar file. However, by starting the program through double-clicking the JAR file, one can exit the program by closing the window the program produces.

Once the program runs, it loops infinitely as it constantly receives images taken by the NAO. It converts the images from a byte array into a Buffered Image and then continuously updates a Frame with the image. What is produced is a window of a video feed of the NAO's point of view.

In this test, the program was able to successfully create a JFrame that added an image to it. This image was the converted image from the NAO. This can be seen in the figure below.

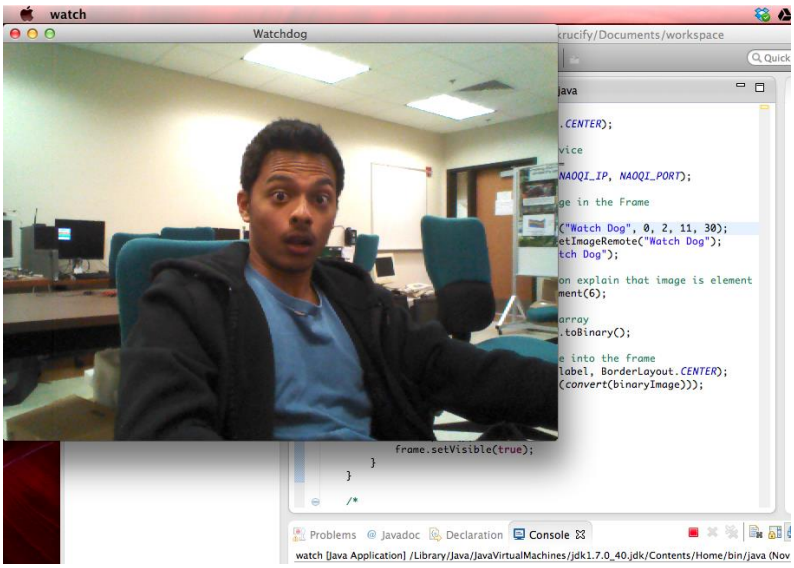

Figure 6: JFrame containing image from NAO

The image transmission requests are done in a while loop. Updating the JFrame's image through the use of the repaint method, the window becomes a live video feed of the NAO's point of view. This is illustrated in Figure 7.

Testing was also done with different resolutions. What was found was that the most smooth feed resulted in the lowest resolution, 320x240. When tests are performed with the next resolution up, 640x480, the feed is considerably choppier. This is most likely due to the transfer of image data via the current network. An image with more pixels than 320x240 would take longer to be transmitted via the network and then converted into a Buffered Image.

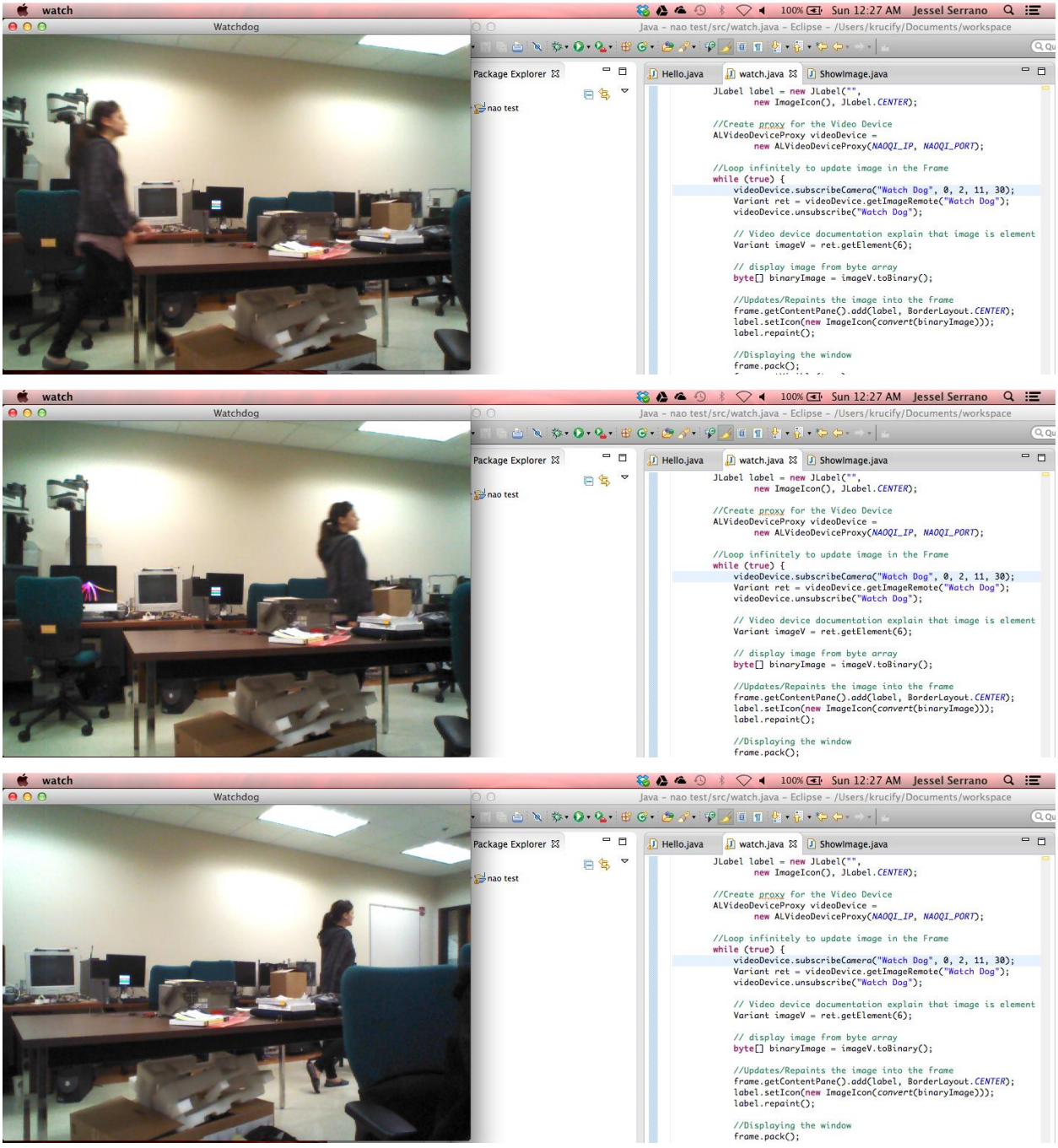

Figure 7: An Illustration of the NAO's live video feed

5.3 Audio & Head + Video: Watchdog Test

In testing both components together, the Watchdog concept would be theoretically complete. The NAO would be able to locate sound, face the sound, and transmit what it is seeing via the Internet. Both these components could be compiled into one large file instead of the two separate files, Tracker and watch. However, after running both files separately and simultaneously, it was found that the NAO successfully became a watchdog, despite two different programs running at the same time.

Once both components ran together smoothly and independent of each other, the decision was made to compile the whole package as is and, if need be, create two separate JAR files: one to ‘see’ what the robot is seeing (Watchdog.jar), and the other to have the robot track sound (Watchdog_Track.jar). Extracting the program into JAR files proved difficult however. This is because the program requires an external java library, which is the NAOqi library (called jnaoqi). When the JAR files are extracted and ran, it loses this external library. In order to solve this, a plugin for Eclipse is used called FatJar, which exports the program as a JAR file with the external library packaged into it [4]. However, it may still be necessary to have the library DLL file (or jnilib file on Macs). Having the JAR file and the extra library files is the bare minimum required to export the Watchdog program and run it on any computer connected to the NAO on the same network.

6. Conclusion

The NAO Watchdog involves using the NAO robot to act as a security camera. It takes live images from the NAO's camera modules and returns them to a client computer. It can then use those images to create a live video feed. The robot also finds the source of sound and turns its head to keep a live feed on the source. This allows the robot to act dynamically and track objects and report footage of the object.

In this project, the NAO Watchdog was fully implemented. However, the operation of the Watchdog is far from smooth. In order for full implementation to be achieved, the program requires the NAO to track sound, move its head to face the sound, and transmit live images to the client. This project successfully implemented the transmission of images into a live video feed on the client computer. The feed can be a smooth 30 fps as long as a low resolution, such as 320x240, is used. Tracking and facing sound, however, was trickier, as the motion of the robot's head depend on precise angles. Since sound can come from multiple angles, especially in a small room, the exact location of the source of the sound may be difficult to determine for the NAO. Thus, when localizing the sound, the angles of azimuth and elevation, two angles needed for positioning the head, were often accurate but imprecise. This causes the head to turn and often miss the target as the target may have already walked past the robot.

On another note on the sound, sometimes the NAO will detect sounds that are not very audible to the human ear. This causes the NAO to turn its head towards rather random events/objects. In order to avoid this, a third element of the Sound Data array is used: this is the confidence (explained in Section 3). The confidence is on a scale of zero to one and determines the confidence to which the NAO heard a definite sound. After thorough testing, it was determined that anything above 0.3 confidence is a good enough indicator of human/loud sounds. Using this parameter, the Watchdog was able to be slightly adjusted to become more precise in its tracking.

In the final development of the Watchdog project, it was discovered that the two components of the Watchdog (tracking and video transmission) could be run separately or simultaneously on the NAO robot. This allowed the creation of two distinct JAR files that can be run together or independently. Or, one could run the Watchdog.jar file to have the whole program run at once, automatically. For this project, two JAR files were created: Watchdog.jar and Watchdog_Track.jar. The latter is the program that tracks sound and positions the NAO's

head towards the sound. When running this JAR file, it was found that there was no way to exit the client once the program had started. This has obvious implications. To fix this and provide a way to exit the client, the program would need to include event handlers and special events on closing. A window with a 'CLOSE' button could suffice. For the Watchdog.jar file, the program closes properly when exiting the window it creates. Special events can be programmed on closing for this as well.

Further extension of this project can include making the robot more responsive to sound, thus, making the robot more apt at tracking. The Watchdog project can also be implemented to use motion tracking. This is done by first determining where an object is via sound, then turning to face the object. Once it has detected the object, the robot will then track it via the camera module and the object's motion. Finally, the Watchdog can truly become a Watchdog by implementing full-body motion into the robot, so the robot may follow and track the target more efficiently. Having a robot that can track by following on foot is one step away from military drones.

7. References

[1]"NAO Doc." *Aldebaran Robotics*. Aldebaran Robotics, 2012. URL: <https://community.aldebaran-robotics.com/doc/1-14/>.

[2] Beiter, M. , B. Coltin, S. Liemhetcharat. *An Introduction to Robotics with NAO*: Aldebaran Robotics, 2012.

[3] Hughes, A. "Working with the NAO Humanoid Robot." Florida Gulf Coast University, 31 August 2013. URL: <http://itech.fgcu.edu/faculty/zalewski/projects/files/HughesWorkingWithNaoZV7.pdf>.

[4] Hechler, F. “Fat Jar Eclipse Plug-In.” SourceForge, 02 December 2013. URL: <http://fjep.sourceforge.net>.

8. Appendix

The appendix is divided into two categories, each containing its own file/program.

8.1 Tracker

```
import com.aldebaran.proxy.*;

public class Tracker {
    public static String NAO_IP = "69.88.163.51";
    public static int NAO_PORT = 9559;

    public static void main(String[] args) {
        ALTextToSpeechProxy tts = new ALTextToSpeechProxy(NAO_IP,
NAO_PORT);
        ALMotionProxy robot = new ALMotionProxy(NAO_IP, NAO_PORT);
      ALAudioSourceLocalizationProxy locator =
                new ALAudioSourceLocalizationProxy(NAO_IP, NAO_PORT);


      //Loops infinitely to track movement
        while (true) {
           locator.subscribe("SoundLocal");

              //This statement calls a method to get audio source data
              float [] headPosition = soundLocalizer(locator);

              //Prints 6D Vector array for position of sound
              System.out.print("Set: ");
              for (int i=0; i<3; i++) {
                 System.out.print(headPosition[i] + " | ");
              }
              System.out.println();

              //MOVING
              //tts.say("I see you.");
              robot.wakeUp();
              try { Thread.sleep(100); }
              catch (Exception e) {};

              //Sets the current head position of the robot
              //if confidence interval is above 0.3 confidence
              if (headPosition[2] > 0.30f || headPosition[2] < 0.1f) {
                    robot.setAngles(
                       new Variant(
                                 new String[]{"HeadYaw", "HeadPitch"}),
                       new Variant(
                                 new float[]{headPosition[0],
headPosition[1]}),
                       0.1f);
              }
```

```
                try { Thread.sleep(100); }
                catch (Exception e) {};

                //Prints the current head position of the robot
                /*System.out.print("Get: ");
                for (int i=0; i<6; i++) {
                   System.out.print(
                               robot.getPosition("Head", 2, true)[i] + " |
");
                }*/

                System.out.println();
                robot.rest();
                locator.unsubscribe("SoundLocal");
            }
        }

    public static float[]
soundLocalizer(ALAudioSourceLocalizationProxy loc) {

      //Create the proxy for Memory
      ALMemoryProxy mem =
                new ALMemoryProxy(NAO_IP, NAO_PORT);

      //Sets the audio sensibility to detect quiet or loud sounds
      loc.setParameter("Sensibility", new Variant(0.9f));

         //Store data in Variant type
      Variant soundData = new Variant(mem.getData(
                ("ALAudioSourceLocalization/SoundLocated")));

      //Converts Variant data to float array,
      //just takes the azimuth and elevation angles
      float[] pos = soundData.getElement(1).toFloatArray();
      System.out.println("azimuth: " + pos[0] +
                " | Elevation: " + pos[1]);
      return pos;
    }
}
```

8.2 Watcher

```
import java.awt.image.BufferedImage;
import java.awt.*;
import javax.swing.*;

import com.aldebaran.proxy.*;

public class watch {

	private static BufferedImage img;
	private static String NAOQI_IP = "69.88.163.51";
	private static int NAOQI_PORT = 9559;

	/*
	 * This method first creates a Frame for the video feed.
	 * It then registers a proxy on the Video Device for the
	 * NAO. It retrieves an image and converts the image into
	 * a Buffered Image using the convert() method.
	 * It then loops and continuously grabs images from
	 * the NAO, converts, and displays them in the Frame.
	 */
	public static void insertImage() {
		//Creating the window
		JFrame frame = new JFrame("Watchdog");
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.setPreferredSize(new Dimension(320, 240));
		JLabel label = new JLabel("",
				new ImageIcon(), JLabel.CENTER);

		//Create proxy for the Video Device
		ALVideoDeviceProxy videoDevice =
				new ALVideoDeviceProxy(NAOQI_IP, NAOQI_PORT);

		//Loop infinitely to update image in the Frame
		while (true) {
			videoDevice.subscribeCamera("WatchDog", 0, 1, 11, 30);
			Variant ret = videoDevice.getImageRemote("WatchDog");
			videoDevice.unsubscribe("WatchDog");

			// Video device documentation explain that image is
element 6
			Variant imageV = ret.getElement(6);

			// display image from byte array
			byte[] binaryImage = imageV.toBinary();

			//Updates/Repaints the image into the frame
			frame.getContentPane().add(label,
BorderLayout.CENTER);
			label.setIcon(new ImageIcon(convert(binaryImage)));
			label.repaint();
```

```
                //Displaying the window
                frame.pack();
                frame.setVisible(true);
            }
        }

        /*
         * This method coverts a byte[] of image data into a
         * Buffered Image  for display
         */
        public static BufferedImage convert(byte[] buff) {
            //Converting the NAO image buffer into a usable image
            int[] intArray;
            intArray = new int[320*240];
            for(int i = 0; i < 320*240; i++)
            {
                intArray[i] = ((255 & 0xFF) << 24) | //alpha
                      ((buff[i*3+0] & 0xFF) << 16) | //red
                      ((buff[i*3+1] & 0xFF) << 8)  | //green
                      ((buff[i*3+2] & 0xFF) << 0); //blue
            }

            img = new BufferedImage(320, 240,
BufferedImage.TYPE_INT_RGB);
            img.setRGB(0, 0, 320, 240, intArray, 0, 320);

            //Returns the converted image as a BufferedImage
            return img;
        }

        public static void main(String args[]) {

            //this method inserts an image continuously into a JFrame
            insertImage();

        }
}
```