

Smart Multi-purpose Passerby Counting System

Project Report

Group Members:

Rongchuan Sun (23715251)

Brandon Ke (22731448)

Aryan Radadiya (23851299)

Yunwei Zhang (23825634)

Department of Engineering and Mathematical Sciences
The University of Western Australia

Abstract

With more people being cautious with COVID-19, a system that would allow them to monitor activity of people entering and leaving various places was deemed to be suitable to maintain the number of people at the same location. The Smart Multi-Purpose Passerby Counting System was designed so that users could deploy this system at any place to detect when people were entering and leaving. This system would need to send semi-dynamic data to an AWS IoT database in the Cloud so that it could be distributed to an Angular JS web application for users to view. They could access and analyse these statistics and read the time that they were recorded so that they could make decisions to help maintain a good number of people at a location. Ultrasonic sensors were initially tested to evaluate the performance of the system's precision and reliability, with laser distance sensors also tested to evaluate how much it improved its accuracy. This would help decide which implementation of the system would be most suitable for the project's application. Different limitations and assumptions being stated to construct an appropriate environment to deploy them in.

Contents

Abstract.....	2
1.0 Introduction	5
1.1 Motivation of the Passerby Counting System.....	5
1.1.1 Public Transportation.....	5
1.1.2 Hospitals and Medical Centres	5
1.1.3 Other Everyday Activities.....	Error! Bookmark not defined.
1.2 Project Objectives (revise near the end).....	5
2.0 Solution and Implementation	6
2.1 Design of the Passerby Counting System.....	6
2.2 Hardware System.....	6
2.2.1 Sensors.....	6
2.2.2 Microcontroller	10
2.2.3 Connectivity	10
2.2.4 Hardware Implementation & Configuration	10
2.3 Software Architecture.....	11
2.3.1 Raspberry Pi Application	12
2.3.2 Platform	14
2.3.3 Analysis	15
2.3.4 User Interface	16
2.4 Requirements and Restrictions.....	16
2.4.1 Cost	16
2.4.2 Power Requirements	16
2.4.3 Accuracy and Reliability of Sensors.....	17
2.5 Differences from the Project Proposal.....	17
2.4.1 Image Processing with RaspberryPi Camera.....	17
2.4.2 Blynk Application	17
2.5 Assumptions.....	Error! Bookmark not defined.
3.0 Results and Discussion	18
3.1 Testing.....	18
3.2 Outcome and Findings	18
4.0 Conclusion.....	20
4.1 Future Work & Recommendations	20

References	22
Appendix A: GitHub Repository for Project	23

1.0 Introduction

To resolve the concerns and issues regarding contagious and severe illnesses being passed on in public, especially in large crowds and small enclosed places, this report proposes a system that could be used by people to help monitor and maintain the number of people that are occupying a particular location. With this system, they can evaluate whether there are too many people at the same place and identify times and days at which many people are entering or leaving. It is important that they keep the maximum number of people at the same place to reduce the risk of someone that may have an illness that could easily be passed on and serious symptoms to more people.

1.1 Motivation of the Passerby Counting System

Ever since COVID-19, there have been restrictions for limited number of occupants in various places and environments. It is obvious that these restrictions have been imposed over time, with people being advised to be cautious of where they are and how busy their location is [1]. There are many different cases and locations of which people are concerned with when trying to protect themselves from contracting various illnesses within a tight space depending on how busy those places are.

1.1.1 Public Transportation

In the case of public transportation, buses via public transportation usually have a limit to how many people are allowed on board. Sometimes, at specific locations and times, there may be too many people for a bus to take e.g., CBD during peak time. Whether it is to avoid hitting the limit of weight the bus can take or to reduce the risk of people contracting contagious illnesses, especially COVID-19, the bus will not be able to take everyone if there is a large group of them at one place.

1.1.2 Hospitals and Medical Centres

Hospitals have a crucial need to continuously monitor who enters them and how many people are currently there to avoid potential issues such as hospital-overcrowding [2]. Hospitals can have many people that have contracted different kinds of illnesses, whether it be acute or serious, with COVID-19 being one of them. Having too many people located there increases the risk of spreading illnesses, which is why it is important to monitor the number of people in there or entering in.

1.2 Project Objectives

The system of the project was developed with the purpose of resolving the growing concerns of ensuring that places are not too overcrowded to avoid increasing the risk of people passing on illnesses. When using this system, people should be able to monitor the number of people that are currently at the same place and identify the trends of people leaving and entering at particular times. With this, people that access the data recorded and processed can use it to make decisions to help resolve this project's issue, such as answering questions like "what time has the most people entering my location?"

The readings sent by the system was evaluated to determine the reliability and the accuracy of different sensors so that the best option was used for the final implementation. There are already existing solutions that were experimented with [3]. This project will experiment with other sensors to determine and compare their performance in accuracy and reliability. A decision on which variation should be used for the final version of the system would be determined by their ability to achieve its main purpose, ideally with high accuracy.

2.0 Solution and Implementation

2.1 Design of the Passerby Counting System

The design of the entire system, including the hardware components and the Cloud server, is shown in Figure 1. When a passerby (person passing some gateway e.g., door, corridor) is detected from one of two sensors connected to the system's computing module (the Raspberry Pi microcontroller), these sensors would need to transmit data to that microcontroller so that it could execute some logic to update the current number of people at the deployed location. After some time, the module should transmit the data to the database stored on the cloud while being connected to a wireless network via a wireless router placed at that location. Once the data is stored in the database, it should be available for users to view in the form of a dashboard of these recordings and analytics via a User Interface i.e., a mobile application. Mobile device users should be able to view this dashboard any time when making a request.

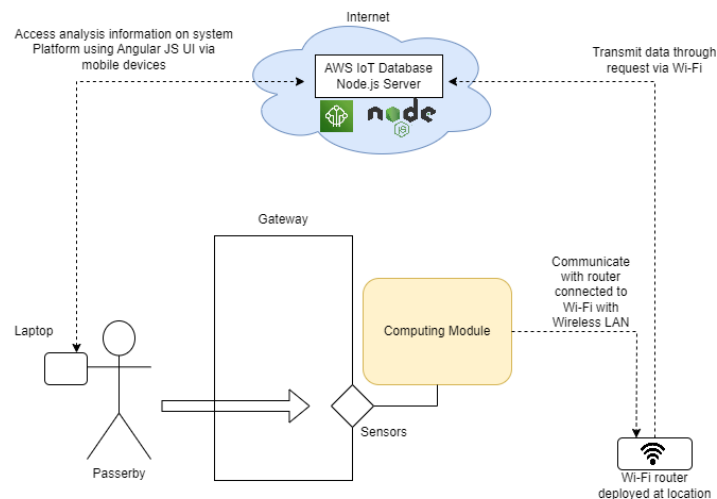


Figure 1: Diagram of the system's functionality and implementation of steps

2.2 Hardware System

2.2.1 Sensors

The sensors selected for this system were chosen knowing that very precise detections of people passing needed to be made by the system. The sensors should not cover a large area to detect people, otherwise, it was possible it could make false detections when people haven't even passed through yet e.g., PIR sensors.

It was assumed that when a person was detected from the system, only one person would pass through it one at a time. No more than one person should walk through at the same time. It was also assumed that when someone was passing through, they must continue to walk to the other side of the system and not turn back when the first sensor detects them.

2.2.1.1 Ultrasonic Sensors

The first type of sensor that was used to test the system's functionality of detections was the RCWL-1601 Ultrasonic Distance Sensor. This sensor was used to measure the distance between itself and any

passing by people by sending out ultrasonic signals via echolocation every few tens of milliseconds; the speed of the sound pulse was 34300 cm/s. Ultrasonic sensors were the cheapest and easiest to retrieve due to the large number of them that were available by the university. But they were not the most reliable of sensors when it came to making consistent and accurate readings of distances. The process of which it needed to measure caused there to be a delay between readings; for the sensor to calculate the distance, it needs to send out an ultrasonic sound pulse and wait for that pulse to return to it so that it could use the time duration between when it was sent and returned to measure a value for the distance between itself and the reflecting surface that the sound bounced from [4]. The calculation of the distance detected by the sensors can be determined by using the pulse durations from when it was first sent out to when it was retrieved in Equation 1.

$$Distance = \frac{time \times speed}{2} \quad \text{Equation 1}$$

Once the pulse duration of the detection was recorded, it would be passed through the logic module of the software system to determine if someone has indeed passed through, and which direction they were moving at. If the distance was less than a set threshold, it would indicate that someone was close enough to the sensors for it to know someone passed through. A diagram that demonstrates how the system functions when the sensors detect a person and the logic it executes is shown in Figure 2. The physical hardware wiring of this implementation is shown in Figure 3 using the Fritzing software tool.

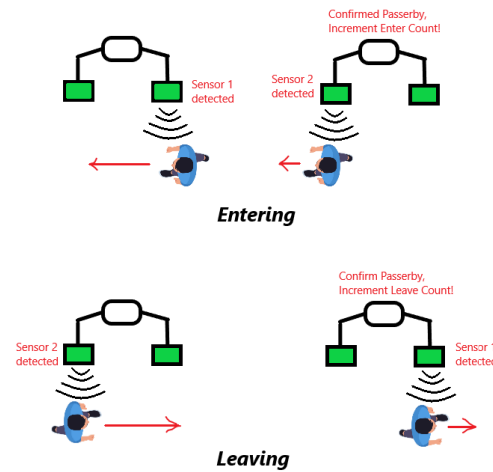


Figure 2: Visual representation of sensors detecting people when they are entering and leaving.

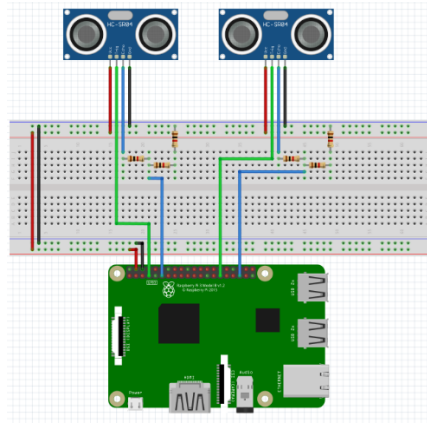


Figure 3: Wiring Diagram of Hardware system for Ultrasonic Sensors Implementation

The required configuration for this setup is as follows:

- 6 x 1k ohm resistors
- VCC connects to Pin 4 (5V)
- Trig connects to Pin 7 and Pin 29 (GPIO4 and GPIO5)
- Echo connects to R1 and R2 (1k ohm)
- R3 and R4 (1k ohm) connects from R1 to Ground
- R5 and R6 (1k ohm) connects from R2 to Ground
- A wire from R1 and R3+R4 connects to Pin 11 (GPIO17)
- A wire from R2 and R5+R6 connects to Pin 35 (GPIO19)
- GND connects to Pin 6

The idea behind using two sensors was so that the system would be able to determine the direction of the person passing by to make updates to the number of people that have entered and left during a time period. If a sensor on one side detected someone passing first, the system would set up some Boolean flag so that when the next sensor detects something within a few readings, it would know that someone had already passed through the first sensor and make an appropriate inference as to if they were entering or leaving.

2.2.1.2 Laser Distance Sensors

An alternative sensor type that was considered for the system was the VL53L1X PiicoDev Laser Distance Sensor. This sensor was capable of using Time-of-Flight (ToF) of the light transmitted from the sensor to measure the distance between itself and people passing by. This works similarly to the ultrasonic sensors in which it uses ToF to calculate distances periodically, but by relying on light instead of sound, the pulses transmitted by this sensor was deemed to being more reliable in returning quick and robust measurements from detections (the speed of light is way faster than the speed of sound). The accuracy and precision of laser distance sensors could also prove to being better than ultrasonic sensors since they cannot be interfered with temperature, humidity, or even other noise signals, except for particular radio frequencies. While it can prove itself to being more accurate and faster than ultrasonic sensors, it is also makes it the more expensive of the sensors due to the technology involved in its production, and also not as widely as accessible as ultrasonic sensors due to making it popular with its precise and reliable traits. Figure 4 shows a simple diagram of the hardware setup of the laser distance sensors

implementation; since PiicoDev components were not available in the Fritzing app, other tools were used. As shown in the figure, the setup for laser distance sensors is easier than what was required for ultrasonic sensors. A PiicoDev adapter for RaspberryPi allowed simple connection between the laser distance sensors and the microcontroller by using PiicoDev cables. The functionality of the system with the sensors is no different than when ultrasonic sensors were implemented; the laser distance sensors were used to confirm if someone has passed through the system and determine the direction they were moving at. The implementation and execution of sensors is similar to Figure 3, but instead, laser distance sensors replace the ultrasonic sensors.

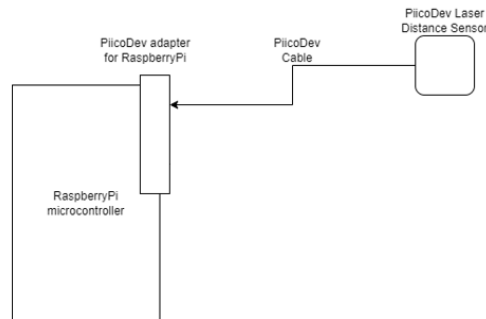


Figure 4: Wiring diagram of Hardware System for Laser Distance Sensors Implementation

The required configuration for this version is simpler than ultrasonic sensors:

- Connect PiicoDev adapter for Raspberry Pi on microcontroller pins (a hole for each one).
- PiicoDev cable to sensor.
- PiicoDev cable to PiicoDev adapter.

A risk about using these sensors for this project was their availability. These sensors could not be purchased from any local stores in Perth or even Western Australia and had to be bought from the online source PiicoDev. Therefore, while the comparison of performance between the different sensors is not the highest priority, it is still ideal to understand which would be more suitable for the project's application. In the end, the sensors were delivered during the last few weeks of the semester, enough time to be tested. However, upon setting up, there was a limitation of the laser distance sensors that were brought up.

Due to the design of the PiicoDev VL53L1X laser distance sensors, each one would have the same I2C address. That would mean that both sensors would return the same value regardless of which one actually made the detection [5]. Because of this, the bi-directional functionality of the system could not be implemented with laser sensors. However, it could still be used to work with a unidirectional functionality for passerbys even though the flexibility of location deployment is not large e.g., detect if someone at the entry gate of a supermarket passes by. Though, the number of cases that could test its precision and reliability would be limited compared to the ultrasonic sensors. If two of these sensors are planned to be used in a similar system, an I2C multiplexer or an additional Raspberry Pi (assuming that both would communicate with the Database API to send their own data) to have it as bi-directional. But due to the limited time left for the project and the limited resources we had on hand, we had to stick to testing it as unidirectional.

2.2.2 Microcontroller

Due to our skillsets and knowledge in programming, the Raspberry Pi was selected so that everyone could easily program and interact with the microcontroller in Python. Python was a simple programming language that all of us could use and were familiar with, so it was decided that the Raspberry Pi would be the main computing module of the system over the Arduino microcontrollers, where most of must would need to be familiar and have a good grasp with the C/C++ language. Python also has a large selection of libraries and packages that can be used for various functionalities that are from open sources, which can include allowing the Raspberry Pi to interact with Piicodev Laser Distance Sensor VL53L1X [6].

2.2.3 Connectivity / Communication

For this project, it was assumed that the system would be deployed at a location that hosted a wireless network through a Wi-Fi router. This would allow the Raspberry Pi to connect to a wireless network so that it could transmit data over the Internet using Wireless LAN; the Wi-Fi router acts as a gateway between the microcontroller and the Internet. No other measurements or implementations to send data from the Raspberry Pi to the Internet was necessary since the microcontroller already had the capability to directly connect with wireless networks [7].

2.2.4 Hardware Implementation & Configuration

Once the components of the hardware system were all connected and set up, it needed to be deployed so that the two sensors were parallel and at a distance from each other. This was so that when a person was passing through it, each sensor should make a detection at two consecutive time iterations which should match the average walking speed of a person (but this can depend on whether they are indeed walking or running).

The sensors also needed to be placed so that they were horizontal. Figure 5 shows a screenshot of the ultrasonic sensors implementation of the system while it was being tested. The laser distance sensors implementation was similar to the ultrasonic design, except only one sensor was placed on the side for detections.

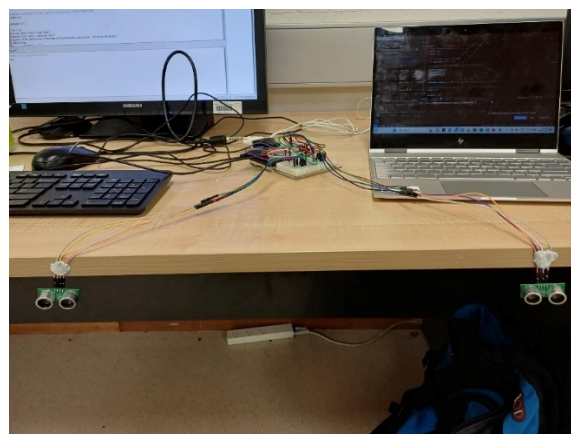


Figure 5: Hardware system of the project, with the Ultrasonic Sensors stuck to the side of the tables with blue tack so that they were facing the correct direction. The distance between them is 60cm.

Depending on the scenario considered, there may need to be a certain amount of distance between them to ensure that they can successfully detect people passing by. For example, ultrasonic sensors

require time to send and receive a noise signal, as discussed in Section 2.2.1.1. If the sensors were too close together, it is likely that the second signal would miss the person passing by after they have passed the first one due to its delay. Therefore, hallways and corridors are recommended locations to deploy the system if ultrasonic sensors are used. Experiments were also conducted when the distance between sensors was at 120cm to evaluate the accuracy of the system for cases when people would be jogging or running past the sensors instead of walking (to match the speed of their movements). It must be kept in mind that the larger the distance is between the sensors, the less flexible the deployment location of the system becomes.

2.3 Software Architecture

Figure 6 shows a summary of the software system's architecture. It would first need to read any incoming detections from the sensors, then feed those detections to the logics module to determine if the current count of people should be updated, as well as increment either the number of people that entered or left during a time period. Once a certain amount of time has passed, the system would need to transmit the data in the form of a JSON object to the Node.js server using an API connected to an AWS IoT database.

The software architecture for the unidirectional implementation of the system with laser distance sensors is shown in Figure 7. Since the computation for detections was simple, there was no need for a logics module to determine the direction of people passing.

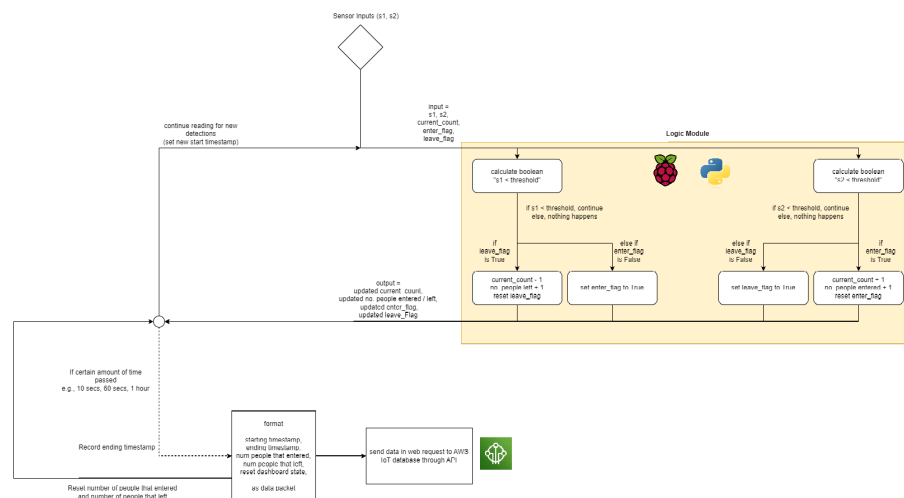


Figure 6: Diagram of System's Bi-directional Software Architecture with detections and computation

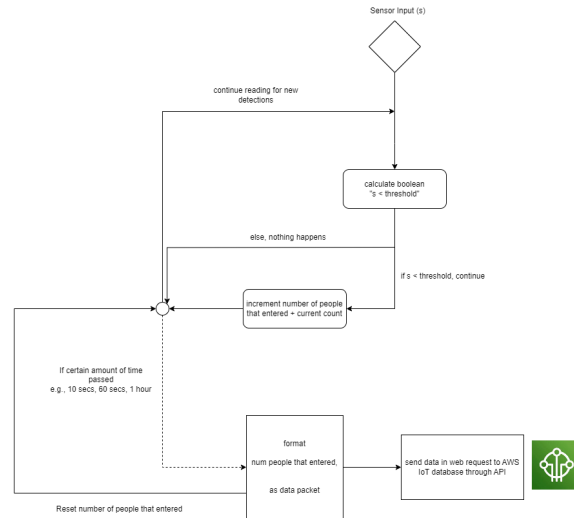


Figure 7: Diagram of System's Unidirectional Software Architecture with detections and computations

2.3.1 Raspberry Pi Application

A group of Python programs were set up to maintain different parts of the software architecture of the system. There would need to be a component that reads any incoming detections from the sensors connected to the Raspberry Pi so that it could calculate the distance measurements recorded and feed it to the next component that would process it and apply logic to these inputs. The decision to make the system bi-directional with its detections was so that the system would be capable of incrementing and decrementing the current count of people without having to use two different gateways for the system, since not all locations will have two different points for entry and exit. The logic's component is shown in the Listing A.1 of Appendix A, which shows how Boolean values are passed in this program to confirm people entering and leaving. Listings A.2 and A.3 in Appendix A shows code as to how the detections from the ultrasonic and laser distance sensors, respectively, are processed for calculating count statistics and to send through the API.

Later after the first test was conducted, more updates were made so that the accuracy of the sensors' detections were more accurate. The different results from testing before and after making the update concerning ultrasonic sensors can be shown in Section 3.2. Initially, the sensors would read and detect pulse signals one after the other. This would mean that while the first sensor was waiting for their signal to return, the other signal was in idle, neither sending out or waiting for their signal. The fix ensured that both sensors would send out and wait for a pulse signal at the same time, implementing synchronous signals.

The program was designed to handle cases where a system may make a false detection after making a successful one, which would leave it in an enter or leave state before someone else passes through. This problem is demonstrated in Figure 8.

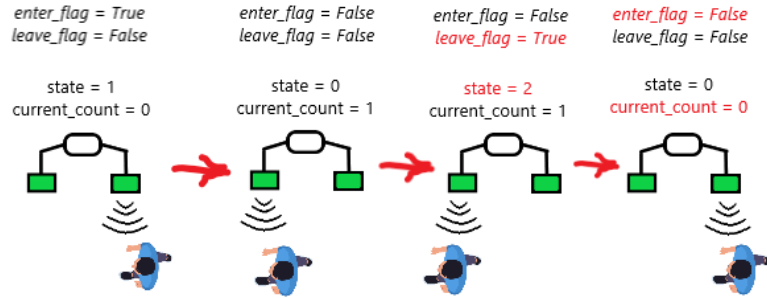


Figure 8: Visual Representation of system making a misdetection when two people are passing by (first person does not pass through second sensor on time).

2.3.1.1 Counting program

The code in Listing A.2 shows the `distance_sensor` program, and Listing A.1 shows the `count` program (Appendix A). These include full steps of reading data from sensor, count, and make it a list preparing to sending to server.

First the `distance_sensor` program would read data from the two sensors ‘`sensor1`’ and ‘`sensor2`’, and return data: sensor name, distance detected, and time, which would be used for counting and analysis. We used Trigger 7, Echo 11 and Trigger 29, Echo 35 for the detect program. The `RPi.GPIO` library was imported to help measure the distances of people passing. Timestamps were also recorded to determine the duration of which these detections were made to be displayed on the UI.

After receiving data from `distance_sensor` from the two sensors with at least 60cm (or 120cm) distance between each other, the `count` program would compute the required counts. If no one passed by, the distance of each sensors detected would be over a set threshold for distance, e.g., 80cm. Every time someone comes or leaves passing by the sensors, they would detect the nearest item that had a distance less than 80cm. That sensor would need to remember that it detected something before by setting a status. Then, according to the status and sequence of sensors, we would collect the data and change the values of `inCount` and `outCount` with the possibilities shown in Table 1.

Scenario	Step	Description
Coming	1	Sensor 1 detected distance < 80cm
	2	Set status = 1 (enter_flag on)
	3	Sensor 2 detected distance < 80cm at next time step
	4	Increment enterCount by 1 and set status = 0
Leaving	1	Sensor 2 detected distance < 80cm
	2	Set status = 2 (leave_flag on)
	3	Sensor 1 detected distance < 80cm at next time step
	4	Increment leaveCount by 1 and set status = 0
Not Actually Coming	1	Sensor 1 detected distance < 80cm
	2	Set status = 1 (enter_flag on)
	3	Sensor 2 and Sensor 1 continues to detect distances >= 80cm for the next few time steps
	4	After a few time steps, set status = 0

Not Actually Leaving	1	Sensor 2 detected distance < 80cm
	2	Set status = 2 (leave_flag on)
	3	Sensor 1 and Sensor 2 continues to detect distances >= 80cm for the next few time steps
	4	After a few time steps, set status = 0

Table 1: Summary of steps taken by software system when someone is detected passing by.

After we collected the data when someone would either come in or out, we would set a 30 second time period (or some other set interval) to create a dictionary and based on the data to circulate the number of people entering and leaving. The structure of the JSON packet is displayed in Listing 1.

```
data = {
    "id": "2",
    "startTime": <starting timestamp of the data>,
    "endTime": <ending timestamp of the data>,
    "inCount": <number of people that entered during time>,
    "outCount": <number of people that left during time>,
    "reset": <flag to determine if data on UI should be reset>,
}
```

Listing 1: Structure of data to send through API in the form of a JSON object.

After a certain amount of time had passed, the program would create a JSON file called 'data.json' and store the dictionary in it, which would be sent through the connected API to be displayed on the UI. Note that the reset field determines if the statistics on the User Interface (UI) should be reset – which would be when the software of the system starts/restarts on the Thonny IDE of the Raspberry Pi, where all counts are reset to 0.

2.3.1.2 API Communication

The last software component was the part that handled the transmission of data to the system's Platform. After a duration of time has passed, the system would pack up the data shown in the previous section to be sent as a PATCH request through Python's requests package. The API URL to the system's database on AWS was provided to the system so that it would know where it had to send the JSON data to. If the response returned to the system was successful, it would reset its enter and leave count to prepare data for the next packet. If it was not successful, it would keep the current enter and leave count for the next attempt to send the packet to the API.

2.3.2 Platform

2.3.2.1 Node.js

Node.js is used to install and build the server file with express.js, which is used to fetch data from the data collection python file, which may include data of period, population, coming and leaving numbers, and then we validate & store the data on Postgres database to save data for future use of get Apis as well as different analysis purposes. After creating the JSON file in counting step, an API would be called to send a request to it using a specific URL. A server that was responsible for handling the API and connection between the system itself and the UI was developed using Node.js.

Using node.js, we created counts of people (entering and leaving) and the timestamp data to show in the UI's dashboard, which was accessible by (ideally) the location admin who would want to continuously monitor data and gain insights from the trends shown on the dashboard.

2.3.2.2 Postgres DB on AWS

```
1 select * from locations;
2
3 select * from location_timesheets
4 order by startTime;
```

Data Output Messages Notifications										
	Id [PK] integer	startTime bigint	endTime bigint	InCount integer	outCount integer	locationId integer	totalCount integer			
1	5	1697040229543	1697040291717	25	5	2	20			
2	6	1697040292717	1697040352717	15	0	2	35			
3	7	1697040353717	1697040413717	20	0	2	55			
4	8	1697040414717	1697040474717	2	20	2	37			
5	9	1697040475717	1697040535717	1	13	2	25			
6	10	1697124909108	1697124969108	25	2	2	23			
7	11	1697124970108	1697125030108	15	0	2	38			

Figure 9: Sample of the Postgres on AWS with an SQL query to fetch some rows.

For the database, we had used Postgres as this was a popular Relational Database Management System, especially for IOT (Internet of Things) devices as it provided fast data retrievals.

To deploy the database, we have used AWS cloud, which provided us RDS (Relational Database Service) to simplify the process of setting up, operating, and scaling a relational database in the cloud. A sample of the database is shown in Figure 9.

The Postgres database was accessed by the node.js server to store the data of the location and its timesheet-related data so that the GET APIs could use this data whenever they send a request to the UI. This data could also be useful for further improvements if we wanted to do some time-interval-based analysis for that location.

2.3.3 Analysis

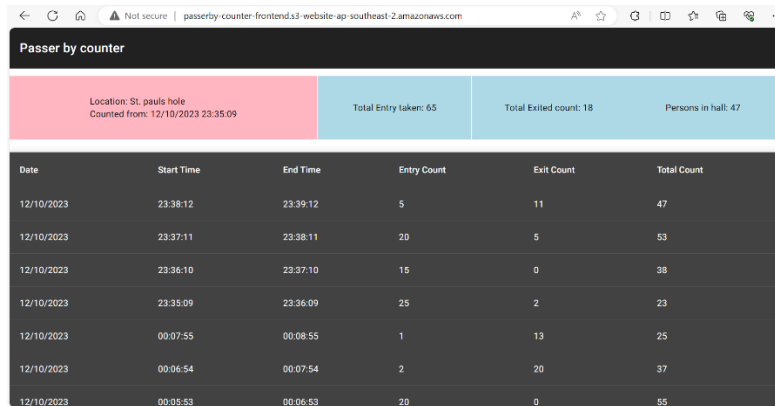
In the server we have managed to store the data for single execution cycle of sensor. For example, a sensor would have started detecting at 10am, meaning it would need to send the data on detections periodically e.g., every 30 seconds or 1 minute. We would store that data in the database for a set time interval continuously. We would show the data in an Angular JS frontend UI for counts of current intervals as well as show a list of all activity in timesheet for every recorded interval that was set by us (somewhat real-time data, depending on how small the interval was).

The dashboard also shows the starting time of the current time interval for recording so that admin users can check how many users entered from some point in time and how many people have exited.

From the dashboard, admin users can check which time interval during the day that had the most entries and exits recorded. As this was the base project, we could make changes in data processing from the server end to change the data representation in the UI for various application as per the requirements.

2.3.4 User Interface (UI)

2.3.4.1 Angular JS



Passerby counter					
Location: St. pauls hole Counted from: 12/10/2023 23:35:09		Total Entry taken: 65	Total Exited count: 18	Persons in hall: 47	
Date	Start Time	End Time	Entry Count	Exit Count	Total Count
12/10/2023	23:38:12	23:39:12	5	11	47
12/10/2023	23:37:11	23:38:11	20	5	53
12/10/2023	23:36:10	23:37:10	15	0	98
12/10/2023	23:35:09	23:36:09	25	2	23
12/10/2023	00:07:55	00:08:55	1	13	25
12/10/2023	00:06:54	00:07:54	2	20	37
12/10/2023	00:05:53	00:06:53	20	8	55

Figure 10: Screenshot of the User Interface displaying real-time dynamic data of the system, with some summary statistics at the top (<http://passerby-counter-frontend.s3-website-ap-southeast-2.amazonaws.com/>).

For the frontend of the UI, we used Angular JS technology for its benefits to develop the data-related applications like two-way databinding and component-based architecture etc. The design of the UI is displayed in Figure 10 and can be accessed through a URL while the server is active. To host the frontend, we used AWS Simple Storage Service with its static website hosting feature.

For the data representation in the UI part, we used the GET location details API and GET dashboard API to get original data calculated using sensors. And then we show the current counts at the top of the UI, and below that we represented a paginated list of activities done on every e.g., minute (again, this interval can depend on what was set by us).

2.4 Requirements and Restrictions

2.4.1 Cost

A budget of \$50 was initially set for the project. Since the university already had most of the components required for the system on hand, the costs of these items were not included in the project's budget. This included the Raspberry Pi, the breadboard, the power supply charger for the Raspberry Pi, jumpwires, resistors, a PiicoDev Raspberry Pi adapter, PiicoDev cables, and ultrasonic sensors.

However, the alternative sensor that was used for experimentation: the "VL53L1X PiicoDev Laser Distance Sensor," was required to be purchased from a source across the east of Australia. Each laser distance sensor incurred a cost of \$20.55, with the standard shipping cost for both being \$7.36. The total cost of these sensors, including shipping, was \$49.26, barely reaching the budget of the project.

2.4.2 Power Requirements

The system should be deployed at a location where people may frequently enter and leave. These locations should usually be populated and have easy access to electricity (no rural areas), such as

restaurants, hospitals, schools etc. Because of this, power requirements for the system were not a huge concern to focus on for this project.

2.4.3 Accuracy and Reliability of Sensors

It is important that the sensors that will be used in the final implementation of the system are considered accurate and reliable. Accurate in the sense that the readings that the detections and distances the sensors return values that are correct, and reliable in the sense that the detections made are tolerant to being interrupted and remaining consistent with the rate of which these detections are made e.g., the sensors should make a detection every 0.1 seconds without any delays. To evaluate the accuracy and reliability of the different sensors, both will be tested under different case scenarios that will be introduced in Section 3.1, with the results of the system's evaluation in Section 3.2 (this should test them when different numbers of people enter and leave during a time period).

2.5 Differences from the Project Proposal

During the development of the system, there had to be a few decisions made by us to exclude some functionalities or change some choices based on the design of the system that differ from the proposal. This section covers these decisions and explanations as to why these decisions were made.

2.5.1 Image Processing with RaspberryPi Camera

The project proposal originally mentioned that one of the functionalities of the system was to use the RaspberryPi Camera to take photos of people passing by and send them through image processing so that the system would be able to extract more details and information on who has passed by. It was decided that this functionality was not going to be implemented due to the time constraint of this project and the complexity of this function (a whole other component for an image processing algorithm, in addition to gathering photos for training and testing, and libraries e.g., OpenCV or Mediapipe, would be required to ensure that this functionality is successful). This project aims to focus on the core functionality of tracking the number of people at a location and recording the number of people entering and leaving during a period of time.

2.5.2 Blynk Application

The proposal for the project initially stated that the Blynk application was considered to hosting the database to store the data being sent and act as the UI for mobile users to access that data. The decision for database, Platform, and UI was changed to relying on Angular JS to construct the API and server and AWS IoT to host the server and the database. This was due to Blynk not provided any resources that allowed us to conduct any analysis or processing on the data being sent. Blynk only allowed us to send the data directly from our system to the application set up, and then display the raw data that was transmitted. Since we wished to select an option that was more flexible with processing the transmitted data over the cloud, the alternative of using AWS IoT to host the database and Angular JS was chosen for the Platform and the UI.

3.0 Results and Discussion

3.1 Testing

To determine the reliability and accuracy of the different implementations of the system, different case scenarios were considered for the sake of testing the software and hardware components of the project. Depending on the results from the measurements, a decision to which implementation is preferred would be decided after evaluation. The following case scenarios were used for testing:

1. More than one person enters,
2. More than one person enters, more than one person leaves,
3. Everyone enters, everyone leaves,
4. More than one person enters (while jogging or running),
5. More than one person enters, more than one person leaves (while jogging or running),
6. Everyone enters, everyone leaves (while jogging or running).

While evaluating the reliability and accuracy of the system, multiple values were recorded and compared to with the ground truths of the cases. These values were the current count, count of people that entered, and the count of people that left, which are shown in Table 2 and 3.

After the first test of the software and hardware system, there were some fixes and updates to the software with the purpose of improving the accuracy of detections. Once the second test was conducted, it was then that scenarios 4 to 6 were introduced, so Table 2 in Section 3.2 will not cover the results for scenarios 4 to 6 and any results for laser distance sensors. With the laser distance sensor implementation, only cases 1 and 4 could be tested due to its limitation of only working unidirectional with the sensors' shared I2C address. These results can still be shown in Table 3 along with the full results of the ultrasonic sensors version.

3.2 Outcome and Findings

To determine the accuracy of the system with different sensor implementations, the F1 score of each count was calculated by comparing the current number of people at the location, the number of people that have entered during recording, and the number of people that have left during recording. The values recorded during tested were compared to the ground truths of those tests to measure how reliable and accurate the system was with detecting people passing by and maintaining the correct number of people over time. After the F1 score was calculated for each of those count values, the average F1 scores of those results was calculated to get the overall accuracy of the system under different case scenarios (as stated in Section 3.1). Equation 2 shows its calculation.

$$F1\ score = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \quad \text{Equation 2}$$

TP (true-positive) represents the number of detections that were correct, *FP* (false-positive) represents the number of false detections that the system has made (for example, if the recorded count was 14 and the ground truth value was 10, then *FP* would be $14 - 10 = 4$), and *FN* (false-negative) represents the number of detections that the system did not successfully make (for example, if the recorded count was 7 and the ground truth value was 10, then *FN* would be $10 - 7 = 3$). The average F1 scores for all case scenarios for each sensor tested is shown in Table 2 and 3. In the case that the ground truth is 0 and *TP* is 0, then the F1 score would be 100%, and 0% if the ground truth is 0 and *TP* is not 0.

Sensor	Case Nr.	Current Count			Entering Count			Leaving Count			Average F1 (%)
		Recorded	Actual	F1 (%)	Recorded	Actual	F1 (%)	Recorded	Actual	F1 (%)	
Ultrasonic	1	4	10	57	6	10	75	2	0	0	44
	2	0	7	0	2	10	33	2	3	80	37
	3	0	0	100	3	7	60	3	7	60	73

Table 2: Results from testing ultrasonic sensors version of system before making optimisation updates (green rows shows the case scenarios that resulted in the best accuracy, while the orange rows show the worst ones).

Sensor	Case Nr.	Current Count			Entering Count			Leaving Count			Average F1 (%)
		Recorded	Actual	F1 (%)	Recorded	Actual	F1 (%)	Recorded	Actual	F1 (%)	
Ultrasonic (60cm)	1	9	10	94	9	10	94	0	0	100	96
	2	12	5	58	14	10	83	2	5	57	66
	3	11	0	0	15	10	80	4	10	57	45
	4	7	10	82	7	10	82	0	0	100	88
	5	8	5	76	9	10	94	1	5	33	67
	6	12	0	0	13	10	86	1	10	18	34
Ultrasonic (120cm)	1	16	20	88	16	20	88	0	0	100	92
	2	6	10	75	16	20	88	10	10	100	87
	3	0	0	100	16	20	88	16	20	88	92
	4	11	20	70	11	20	70	0	0	100	80
	5	8	10	88	11	20	70	3	10	46	68
	6	0	0	100	6	20	46	6	20	46	64
Laser Distance	1	11	10	95	-	-	-	11	10	95	95
	4	10	10	100	-	-	-	10	10	100	100

Table 3: Results from testing ultrasonic sensors version of system after making optimisation updates (green rows shows the case scenarios that resulted in the best accuracy, while the orange rows show the worst ones).

Looking at the results of Table 3, ultrasonic sensors proved to providing more reliable and accurate detections for the system when the distance between them was larger. With a distance of 120cm, all F1 scores were calculated to being greater than 50%, while there were some scores that achieved below that percentage when the distance was at 60cm. For the sake of accuracy, it would be ideal to have a large distance between the sensors to get more precise readings. But this would cost the flexibility of the location that the system can be deployed at; gateways and doors won't all be as long as corridors and hallways.

After testing the unidirectional laser distance sensors variation of the system, it was found to have very high precision and reliability compared to ultrasonic sensors. Sometimes, the detections made by the sensors was too sensitive that a timeout after a detection was made was necessary to ensure it was not making any misdetections with the same person passing by (a problem that was demonstrated in Figure 8 in Section 2.3.1). But even though laser distance sensors proved to having higher accuracy, the main purpose of this project's bi-directional functionality could only be achieved when using ultrasonic sensors, which meant it had more flexibility in the location that it could be deployed at (can be any entry point rather than a one-way point). In the end, the ultrasonic variation as the most suitable implementation, at the cost of that accuracy.

4.0 Conclusion

Multiple variations of the project's system were developed and tested to determine its performance and ability to send useful information that could be used by potential users to monitor and maintain the number of people that enter, leave, and stay at a location (especially one that can be very busy). An ultrasonic and laser distance sensor implementation was programmed, set up, and evaluated to determine how accurate they could detect people passing by. Ultrasonic sensors did not return the best outcome for precision and reliability, but it did meet the criteria of the system to provide information on passerbys in a bi-directional manner. Laser distance sensors had the highest precision and was reliable with its readings, but it failed to meet the goal of achieving bi-directional behaviour due to the limitations and constraints of the hardware used. Users may decide which version of the project would suit their interests more depending on what they prioritise. In the case of having the system do the main functionality of the project, the ultrasonic sensors were best.

Both the server and UI would need to be managed such that the data collected from the sensors is represented in ways that users could monitor as close to real-time and gain insights to help make decisions about maintaining the number of people at a location. The design of the database schema and APIs were designed in a way so that the flow between "sensor and node server" and "server and UI" could be achieved.

4.1 Future Work & Recommendations

A recommendation, according to the results in Section 3.2, would be that the system needs to be implemented at a spot that would allow the sensors to have a good distance between each other so that it can match the average walking speed of people, mostly long corridors and hallways, rather than narrow gateways and doors. This is assuming that ultrasonic sensors are used to detect the motion of people. If another sensor was connected to the system, it is likely that the conditions of where it can be deployed would be less restricted. But this would not be a concern if laser distance sensors were used instead due to its very fast and precise detection time. However, if laser distance sensors are considered for this system in the future, another microcontroller e.g., Raspberry Pi, should be added so that a second sensor can be used to detect people passing in the other direction since more than one laser distance sensor cannot be connected to the same microcontroller as discussed in Section 2.2.1.2. These two computing modules must somehow communicate with the other one so synchronously send their data together so that the system knows that the data they sent are supposed to be read and displayed together.

Due to the scope and constraints set for this project, there were still many aspects of the system that could have been improved and may be considered in future works. For one, more types of sensors could have been tested that would be simple setting up and running as ultrasonic sensors. One would be the laser distance sensors discussed throughout this report. Another candidate for the system would be gravity microwave motion sensors, but since they cannot be used to calculate the distances of people from itself, it can only be used for unidirectional gateways to simply detect if people are passing it, not which direction they are moving.

The environment that the system is deployed in may also be introduced as a variable to the system's accuracy. Examples of these factors would be the temperature and humidity, the location being inside or outside, being deployed at a place that is metropolitan (where it is likely that there is going to be

interference with other communicating devices) or rural (where Wi-Fi signals and mobile data are likely to be limited). These would be used to test the reliability of communication between the project and the database connected to it over the Cloud to evaluate its reliability.

The need to add user authentication into the system could be desirable to ensure that the data is secured when being processed and stored to the server database. We could simply create super-admin (us) APIs and UIs (User Interface) to register new users and provide credentials to the admin user (manager / owner) of the location. After that, we could add layers of verification of user authentication before serving data on the public Internet. This helps reinforce the security and privacy of data handling of the system.

References

- [1] B. McGrath, "COVID-19 cases explode again in Japan, South Korea," *World Socialist Web Site*, Jul. 30, 2022. Available: <https://www.wsws.org/en/articles/2022/07/30/xvna-j30.html>
- [2] Peter Allely, "Hospital overcrowding could kill more people than COVID-19 in WA," *ABC News*, Feb. 01, 2022. Available: <https://www.abc.net.au/news/2022-02-02/hospital-overcrowding-could-kill-more-people-than-covid-19-in-wa/100792258>
- [3] Nayana R and Bharathi Malakreddy A, "IoT Based Passenger Count System in Public Transport," *International Journal of Innovations in Engineering and Technology (IJJET)*, vol. 10, pp. 93-96. Available: <https://ijiet.com/wp-content/uploads/2018/06/14.pdf>
- [4] MaxBotix, "How Ultrasonic Sensors Work," *MaxBotix*, Mar. 01, 2023. <https://maxbotix.com/blogs/blog/how-ultrasonic-sensors-work>
- [5] "How do I change the I2C address of a PiicoDev sensor?" Core Electronics Forum, Jun. 15, 2022. <https://forum.core-electronics.com.au/t/how-do-i-change-the-i2c-address-of-a-piicodev-sensor/14359/2> (accessed Oct. 11, 2023).
- [6] Michael, "PIICoDeV Distance Sensor VL53L1X - Raspberry Pi Guide," *Core Electronics*. Available: <https://core-electronics.com.au/guides/piicodev-distance-sensor-vl53l1x-raspberry-pi-guide/>
- [7] Raspberrypi.org, 2023. Available: <https://projects.raspberrypi.org/en/projects/raspberry-pi-using/3>

Appendix A: GitHub Repositories for Project

Multi-Purpose Passerby Counting System: <https://github.com/Yunwei-Zhang/CITS5506-GP40/tree/main>

- Logics Module: <https://github.com/Yunwei-Zhang/CITS5506-GP40/blob/main/logic.py>
- Ultrasonic Sensors: https://github.com/Yunwei-Zhang/CITS5506-GP40/blob/main/distance_sensor.py
- Laser Distance Sensors: https://github.com/Yunwei-Zhang/CITS5506-GP40/blob/main/laser_distance_sensor.py

```
import distance_sensor as ds
import time
import requests

from datetime import datetime

# Initialize the current count to 0
current_count = 0
enter_count = 0
leave_count = 0
detected = False

# Define the threshold distance (less than 1 meter)
threshold_distance = 120 # Assume 100 centimeters represent 1 meter

# Define the threshold for the time duration for each packet
THRESHOLD_TIME = 30
threshold_time = THRESHOLD_TIME

# Define a variable to keep track of the previous sensor identifier
state = 0
counter = 0
COUNTER_THRESH = 5

dataset = []

res = {}
packet_start_time = time.time()
packet_start_date = datetime.now()
packet_start_datestring = datetime.strftime(packet_start_date, "%d-%m-%Y %H:%M:%S")

# Variable to determine if the statistics shown on the UI should be reset, should be when the
system first starts.
reset = True

while True:
```

```

res["target"] = "Bus"
# Sensor 1: Trig 4, Echo 17
#s1, d1, t1 = ds.detect(7, 11, "sensor 1")#right
# Sensor 2: Trig 5, Echo 35
#s2, d2, t2 = ds.detect(29, 35, "sensor 2")#left
d1, d2, t = ds.detect()

# If state is 0, check Sensor 1 for detecting a person passing by
if state == 0:
    if d1 <= threshold_distance:
        state = 1
# If state is 0, check Sensor 2 for detecting a person passing by
if state == 0:
    if d2 <= threshold_distance:
        state = 2
# If state is 1, check Sensor 2 for detecting a person passing by
elif state == 1:
    if d2 <= threshold_distance:
        current_count += 1
        state = 0
        enter_count += 1
        detected = True
        print("count++")
    else:
        counter += 1
        #with open('data.json','w') as json_file:
        #    json.dump(res,json_file, indent = 4)

# If state is 2, check Sensor 1 for detecting a person passing by
elif state == 2:
    if d1 <= threshold_distance:
        if current_count >= 1:
            current_count -= 1
            leave_count += 1
            detected = True
            print("count--")
        state = 0
    else:
        counter += 1
        #with open('data.json','w') as json_file:
        #    json.dump(res,json_file, indent = 4)

packet_end_time = time.time()

```



```

packet_end_date = datetime.now() #8 9 1
packet_end_datestring = datetime.strftime(packet_end_date, "%d-%m-%Y %H:%M:%S")
packet_duration = packet_end_time - packet_start_time

if detected:
    # Print the current count of people
    print("status: " + str(state))
    print(f"Current count: {current_count}, Enter: {enter_count}, Leave: {leave_count}")
    print(f"Sensor 1: {d1}, Sensor 2: {d2}")
    print("come time: " + str(packet_start_datestring) + ", leave time: " +
str(packet_end_datestring))
    #print(res)
    detected = False

if counter > COUNTER_THRESH:
    state = 0
    counter = 0

if packet_duration > threshold_time and (enter_count > 0 or leave_count > 0):
    #data = {
    #    'id': '2',
    #    'time_come': packet_start_datestring,
    #    'time_leave': packet_end_datestring,
    #    'people_count': current_count,
    #    'inCount': enter_count,
    #    'outCount': leave_count
    #}
    data = {
        "id": "2",
        "startTime": str(int(packet_start_date.timestamp() * 1000)),
        "endTime": str(int(packet_end_date.timestamp() * 1000)),
        "inCount": str(enter_count),
        "outCount": str(leave_count),
        "reset": str(reset),
    }
    dataset.append(data)

    # If dataset exceeds a size, keep most recent data on system
    if len(dataset) > 100:
        dataset.pop(0)

    # Send to AWS Database
    try:
        api_url = 'http://3.27.155.65:3000/api/location/updateCount'
        print("\nSending data...")

```

```

        response = requests.patch(api_url, json=data, timeout=5)
        status = response.status_code
        if status != 200:
            print('-----')
            print(f"Something went wrong: {response.json()}")
            print('-----')

            # Wait for another set time duration before sending again
            threshold_time += THRESHOLD_TIME
            continue
        else:
            print('-----')
            print('Successfully sent.')
            print('-----')
    except Exception as e:
        print('-----')
        print(f"{e.__class__.__name__}: {e}")
        print("Retry next time")
        print('-----')

        # Wait for another set time duration before sending again
        threshold_time += THRESHOLD_TIME
    else:
        # Uncomment when actually sending data
        enter_count = 0
        leave_count = 0
        threshold_time = THRESHOLD_TIME
        packet_start_time = time.time()
        if reset:
            reset = False

    res["data"] = dataset

    # Wait for some time before taking the next measurement
    time.sleep(0.2)

```

Listing A.1: *count.py file (handles sensor detection, logic, and transmissions)*

```

import RPi.GPIO as GPIO
import time
from datetime import datetime

TRIGGER1 = 7
TRIGGER2 = 29
ECHO1 = 11

```

```

ECHO2 = 35

def detect():
    #GPIO.cleanup()
    try:
        GPIO.setmode(GPIO.BOARD)
        GPIO.setup(TRIGGER1, GPIO.OUT)
        GPIO.setup(ECHO1, GPIO.IN)
        GPIO.setup(TRIGGER2, GPIO.OUT)
        GPIO.setup(ECHO2, GPIO.IN)

        GPIO.output(TRIGGER1, GPIO.LOW)
        time.sleep(0.5)

        GPIO.output(TRIGGER1, GPIO.HIGH)
        GPIO.output(TRIGGER2, GPIO.HIGH)
        time.sleep(0.00001)
        GPIO.output(TRIGGER1, GPIO.LOW)
        GPIO.output(TRIGGER2, GPIO.LOW)

        pulse_start_time1 = 0
        pulse_start_time2 = 0
        pulse_end_time1 = 0
        pulse_end_time2 = 0

        while pulse_start_time1 <= 0 or pulse_start_time2 <= 0:
            if GPIO.input(ECHO1) == 1 and pulse_start_time1 <= 0:
                pulse_start_time1 = time.time()
            if GPIO.input(ECHO2) == 1 and pulse_start_time2 <= 0:
                pulse_start_time2 = time.time()
        while pulse_end_time1 <= 0 or pulse_end_time2 <= 0:
            if GPIO.input(ECHO1) == 0 and pulse_end_time1 <= 0:
                pulse_end_time1 = time.time()
            if GPIO.input(ECHO2) == 0 and pulse_end_time2 <= 0:
                pulse_end_time2 = time.time()

        pulse_duration1 = pulse_end_time1 - pulse_start_time1
        distance1 = round(pulse_duration1 * 17150, 2)

        pulse_duration2 = pulse_end_time2 - pulse_start_time2
        distance2 = round(pulse_duration2 * 17150, 2)

        time_current = datetime.utcnow().strftime('%Y-%m-%d %H:%M:%S.%f')[:-2]
        return distance1, distance2, time_current

```

```
finally:
    GPIO.cleanup()
```

Listing A.2: distance_sensor python code

```
from PiicoDev_VL53L1X import PiicoDev_VL53L1X
from datetime import datetime
import time
import requests

dataset = []

# Define the threshold distance (less than 1 meter)
threshold_distance = 120 # Assume 100 centimeters represent 1 meter

# Define the threshold for the time duration for each packet
THRESHOLD_TIME = 30
threshold_time = THRESHOLD_TIME

# Value to keep track of number of people that passed by during a period, and current number
of people at the place.
count = 0
current_count = 0

distSensor = PiicoDev_VL53L1X()

# Prepare starting time of packet
packet_start_time = time.time()
packet_start_date = datetime.now()
packet_start_datestring = datetime.strftime(packet_start_date, "%d-%m-%Y %H:%M:%S")

# Variable to determine if the statistics shown on the UI should be reset, should be when the
system first starts.
reset = True

while True:
    dist = distSensor.read()
    dist = dist / 10

    if dist < threshold_distance:
        count += 1
        current_count += 1
        print("Someone passed!")
        print(f"Current count: {current_count}, Enter: {count}")
```

```

        print(f"Sensor: {dist}")
        print("come time: " + str(packet_start_datestring) + ", leave time: " +
str(packet_end_datestring))

        # Wait for the person to pass by
        time.sleep(0.7)

    packet_end_time = time.time()
    packet_end_date = datetime.now() #8 9 1
    packet_end_datestring = datetime.strftime(packet_end_date, "%d-%m-%Y %H:%M:%S")
    packet_duration = packet_end_time - packet_start_time

    if packet_duration > threshold_time and count > 0:
        # Only inCount or outCount can be updated due to unidirectional behaviour of this
system.
        data = {
            "id": "2",
            "startTime": str(int(packet_start_date.timestamp() * 1000)),
            "endTime": str(int(packet_end_date.timestamp() * 1000)),
            "inCount": str(count),
            "outCount": "0",
            "reset": str(reset),
        }
        print(data)

        dataset.append(data)
        # If dataset exceeds a size, keep most recent data on system
        if len(dataset) > 100:
            dataset.pop(0)

        # Send to AWS Database
        try:
            api_url = 'http://3.27.155.65:3000/api/location/updateCount'
            print("\nSending data...")
            response = requests.patch(api_url, json=data, timeout=5)
            status = response.status_code
            if status != 200:
                print("-----")
                print(f"Something went wrong: {response.json()}")
                print("-----")

                # Wait for another set time duration before sending again
                threshold_time += THRESHOLD_TIME
                continue
            else:

```

```

        print("-----")
        print('Successfully sent.')
        print("-----")
    except Exception as e:
        print("-----")
        print(f"{e.__class__.__name__}: {e}")
        print("Retry next time")
        print("-----")

        # Wait for another set time duration before sending again
        threshold_time += THRESHOLD_TIME
    else:
        # Uncomment when actually sending data
        threshold_time = THRESHOLD_TIME
        count = 0
        packet_start_time = time.time()
        if reset:
            reset = False

time.sleep(0.1)

```

Listing A.3: *laser_distance_sensor python code*