Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

Design Document – ELEC 477– Lab 4

1.0 - Main Problem

This project utilizes Cyclone DDS version 0.11.0 to simulate the Toronto Air Traffic Control system, focusing on two main zones: Toronto AD (Approach/Departure) and Toronto Centre (Enroute Airspace). Using real-time flight data, the simulation shows the interactions between various aircrafts and control zones, including the handoffs as aircraft transition from one zone to another. The data included parameters such as aircraft location (latitude and longitude), velocity, heading, and altitude. In addition, for the cyclone DDS messaging, IDL files were used to provide the structure and integrity of the message. Both played a pivotal role in creating the simulation setting.

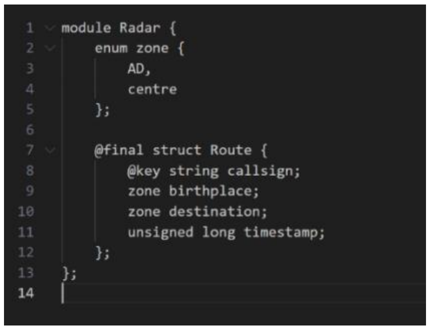
2.0 - IDL Files

The IDL file is an important component for this assignment and is used for defining data types and structures that are communicated across the DDS network. The `statekey.idl` file was provided and is used to standardize the data communication between different air traffic control zones and the aircraft. In addition to the ` statekey.idl` file, a `ZoneTransfer.idl` was created to define the specifics of how airspace transitions are managed. It is used to defining the structure for messages related to the transfer of aircraft between different control zones (i.e., TrontoAD and TorontoCentre).

The `ZoneTransfer.idl` file contains several key elements, including:

- Call Sign: Identifies the specific aircraft involved in the zone transition.
- Birthplace Zone: Indicates the airspace zone from which the aircraft is transitioning to.
- Destination Zone: Specifies the target airspace zone to which the aircraft is transitioning into.
- Timestamp: Marks the specific time at which the transfer message is generated.

The `ZoneTransfer.idl` can be seen below in Figure 1.



```
module Radar {
    enum zone {
        AD,
        centre
    };

    @final struct Route {
        @key string callsign;
        zone birthplace;
        zone destination;
        unsigned long timestamp;
    };
};
```

Figure 1: Structure of the 'ZoneTransfer.idl' file

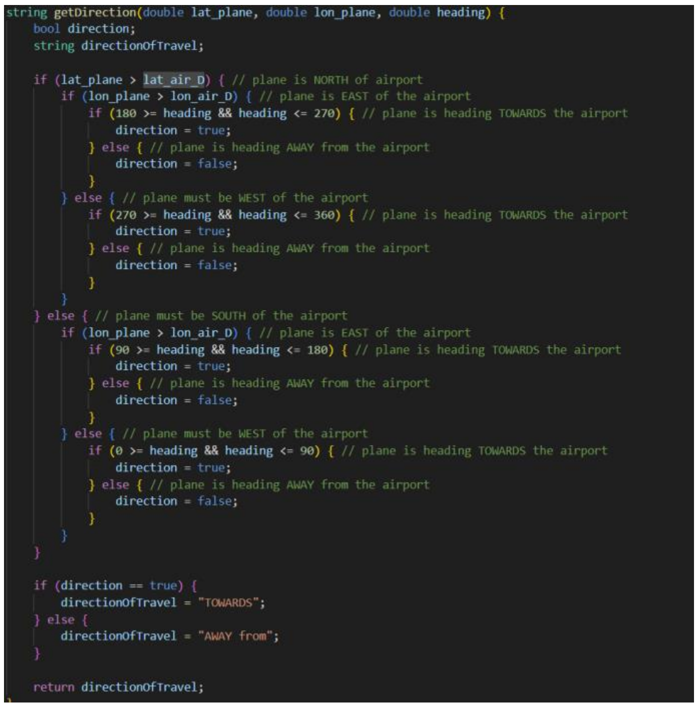
Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096

3.0 - Design Solution

Simulating the Toronto Air traffic control system involved making 3 programs. These were Toronto AD for the Toronto arrival and departure zone, Toronto Centre for the enroute airspace zone, and a Query program to allow the retrieval of the last state sample for a given flight.

3.1 - Directional Code

To establish accurate communication between Toronto AD and Toronto Centre, directional code was added to determine whether the aircraft was moving towards or away from the airport. This determination is based on aircraft position (latitude and longitude) and the Toronto Airport coordinates of 43.6771° N, 79.6334° W. For planes north and east of the airport, a directional angle between 180° and 270° indicates the plane is moving towards the airport, while any other angle indicates its moving away. On the other hand, for any planes south and west of the airport, an angle between 0° and 90° would indicate the plane is moving towards the airport.



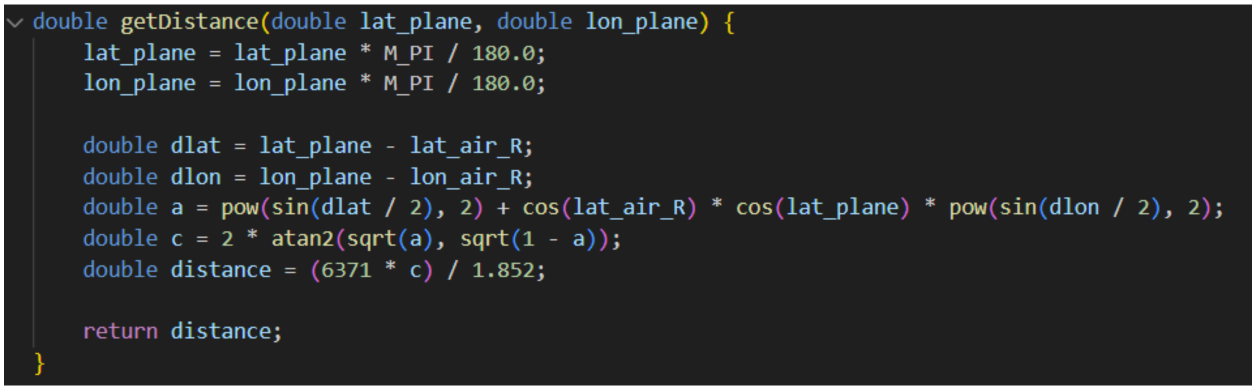
```
string getDirection(double lat_plane, double lon_plane, double heading) {
    bool direction;
    string directionOfTravel;

    if (lat_plane > lat_air_D) { // plane is NORTH of airport
        if (lon_plane > lon_air_D) { // plane is EAST of the airport
            if (180 >= heading && heading <= 270) { // plane is heading TOWARDS the airport
                direction = true;
            } else { // plane is heading AWAY from the airport
                direction = false;
            }
        } else { // plane must be WEST of the airport
            if (270 >= heading && heading <= 360) { // plane is heading TOWARDS the airport
                direction = true;
            } else { // plane is heading AWAY from the airport
                direction = false;
            }
        }
    } else { // plane must be SOUTH of the airport
        if (lon_plane > lon_air_D) { // plane is EAST of the airport
            if (90 >= heading && heading <= 180) { // plane is heading TOWARDS the airport
                direction = true;
            } else { // plane is heading AWAY from the airport
                direction = false;
            }
        } else { // plane must be WEST of the airport
            if (0 >= heading && heading <= 90) { // plane is heading TOWARDS the airport
                direction = true;
            } else { // plane is heading AWAY from the airport
                direction = false;
            }
        }
    }

    if (direction == true) {
        directionOfTravel = "TOWARDS";
    } else {
        directionOfTravel = "AWAY from";
    }

    return directionOfTravel;
}
```

Figure 2: Directional Code for TorontoAD and TorontoCentre

This logic is crucial for deciding when to send control messages. As such the directionOfTravel string returned by the getDirection function returns either “TOWARDS” or “AWAY from” and will be used by the TorontoAD and TorontoCentre later in this documentation document.

3.2 – Haversine Formula

To calculate the distance a plane was from the airport, the Haversine formula was used, shown in Figure 3 below. The code displayed below shows how distance was calculated based on radian distance between two points on the surface of a sphere, given their latitude and longitude in degrees. Since planes are flying across big distances between the zones, this formula is the most accurate to express distances given the coordinates and angle data of planes in the flight's dataset.

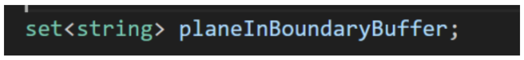


```
double getDistance(double lat_plane, double lon_plane) {
    lat_plane = lat_plane * M_PI / 180.0;
    lon_plane = lon_plane * M_PI / 180.0;

    double dlat = lat_plane - lat_air_R;
    double dlon = lon_plane - lon_air_R;
    double a = pow(sin(dlat / 2), 2) + cos(lat_air_R) * cos(lat_plane) * pow(sin(dlon / 2), 2);
    double c = 2 * atan2(sqrt(a), sqrt(1 - a));
    double distance = (6371 * c) / 1.852;

    return distance;
}
```

Figure 3: The Haversine formula used to calculate the distance between longitudinal and latitudinal points with respect to Earth

3.3 – Boundary Buffer Code

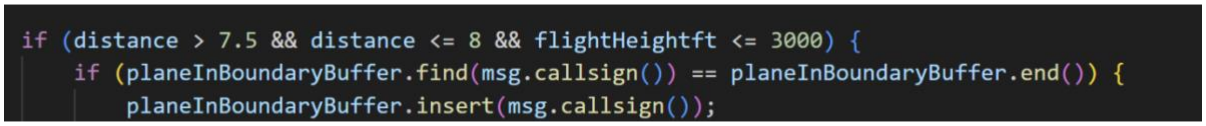
Additionally, for each flight, a buffer is provided to ensure that the flights aren't missed, and to ensure that transfer messages are sent. For example, if a buffer wasn't used, a flight could be outside of the 8nm in one reading, and then inside of 8nm in the next reading – if it was never exactly at 8nm away from the airport, a transfer message wouldn't be sent. Therefore, the buffer used for distance was 0.5nm, and the buffer used for height was 500 feet. Then, the planeInBoundaryBuffer, shown in Figure 4, was a set used to keep track of which planes have been processed regarding boundary alerts and control transfers.



```
set<string> planeInBoundaryBuffer;
```

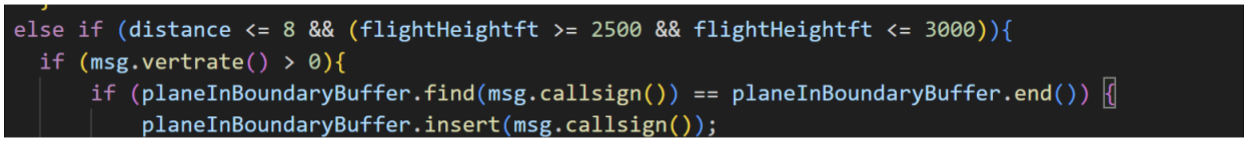
Figure 4: planeInBoundaryBuffer set

Once a plane triggers an alert it is added to this set. Two examples of a plane being added to the boundary buffer set are shown in Figure 5 and Figure 6 below. If its state update triggers the same condition again, the system checks this set first to make sure that duplicate alerts are not sent. This prevents the flooding of control transfer messages and maintains the efficiency of the communication system between air traffic control zones.



```
if (distance > 7.5 && distance <= 8 && flightHeightft <= 3000) {
    if (planeInBoundaryBuffer.find(msg.callsign()) == planeInBoundaryBuffer.end()) {
        planeInBoundaryBuffer.insert(msg.callsign());
```

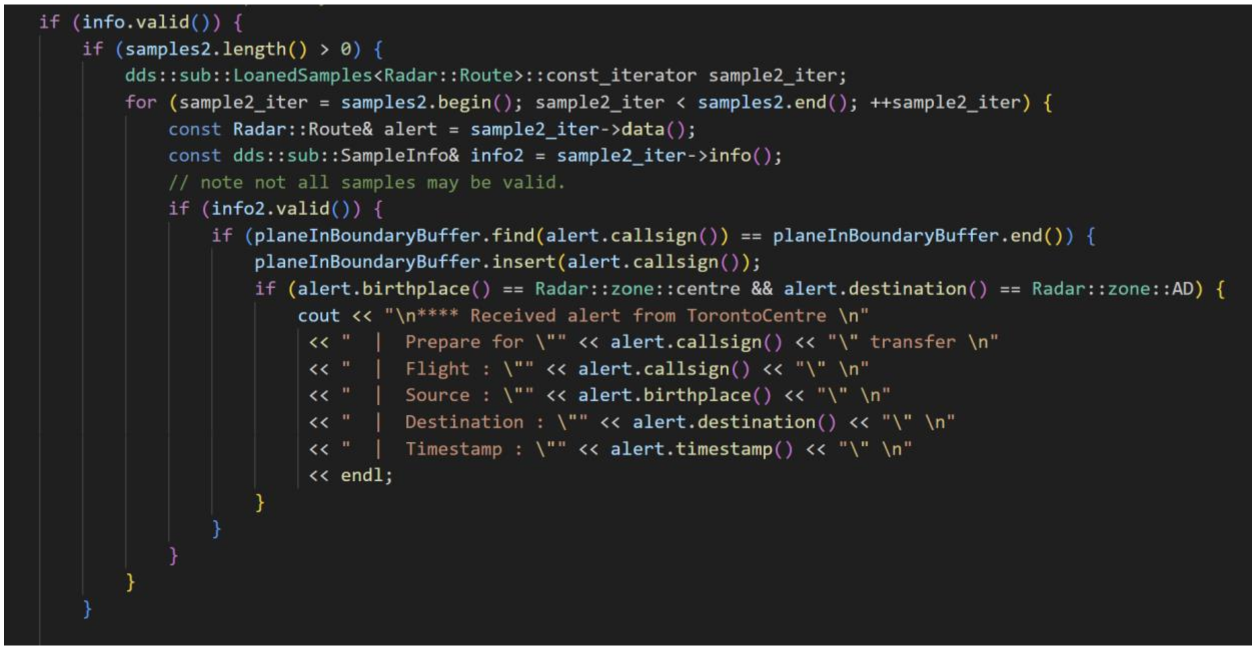
Figure 5: Adding a flight to the planeInBufferBoundary set if it is within the distance boundary (between 7.5nm and 8nm away from Toronto Pearson).



```
else if (distance <= 8 && (flightHeightft >= 2500 && flightHeightft <= 3000)){
  if (msg.vertrate() > 0){
      if (planeInBoundaryBuffer.find(msg.callsign()) == planeInBoundaryBuffer.end()) {
          planeInBoundaryBuffer.insert(msg.callsign());
```

Figure 6: Adding a flight to the planeInBufferBoundary set if it's within the altitude boundary (within 2500ft and 3000ft above Earth).

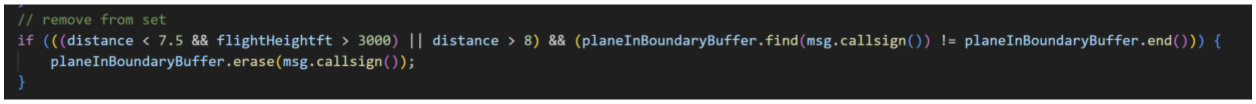
Additionally, a plane can be added to the planeInBoundaryBuffer set based on transfer messages received from other airspaces. For example, in Figure 7 below, if a transfer message is received from a different airspace, the callsign of the plane is added to the set. This means that once the plane actually arrives within in the boundary, the message is not printed twice.



```
  if (info.valid()) {
      if (samples2.length() > 0) {
          dds::sub::LoanedSamples<Radar::Route>::const_iterator sample2_iter;
          for (sample2_iter = samples2.begin(); sample2_iter < samples2.end(); ++sample2_iter) {
              const Radar::Route& alert = sample2_iter->data();
              const dds::sub::SampleInfo& info2 = sample2_iter->info();
              // note not all samples may be valid.
              if (info2.valid()) {
                  if (planeInBoundaryBuffer.find(alert.callsign()) == planeInBoundaryBuffer.end()) {
                      planeInBoundaryBuffer.insert(alert.callsign());
                      if (alert.birthplace() == Radar::zone::centre && alert.destination() == Radar::zone::AD) {
                          cout << "\n**** Received alert from TorontoCentre \n"
                           << "  |  Prepare for \"" << alert.callsign() << "\" transfer \n"
                           << "  |  Flight : \"" << alert.callsign() << "\" \n"
                           << "  |  Source : \"" << alert.birthplace() << "\" \n"
                           << "  |  Destination : \"" << alert.destination() << "\" \n"
                           << "  |  Timestamp : \"" << alert.timestamp() << "\" \n"
                           << endl;
                      }
                  }
              }
          }
      }
```

Figure 7: Adding plane to planeInBoundaryBuffer based on transfer messages.

Then, the aircraft is removed from the planeInBoundaryBuffer when there is no longer a risk of it triggering the same alert it was previously added for. This can be seen in Figure 8 below.



```
// remove from set
if (((distance < 7.5 && flightHeightft > 3000) || distance > 8) && (planeInBoundaryBuffer.find(msg.callsign()) != planeInBoundaryBuffer.end())) {
    planeInBoundaryBuffer.erase(msg.callsign());
}
```

Figure 8: Criteria for removing a plane from the planeInBoundaryBuffer set.

The criteria for removal include:

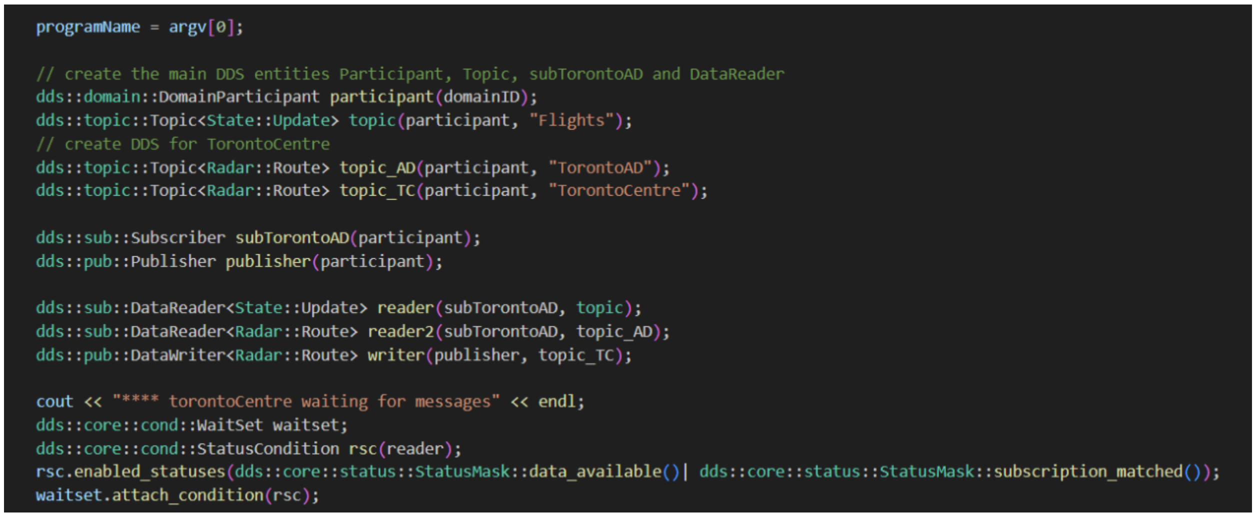
- Crossing through the 8nm boundary from airport
- Altitude changes

Therefore, the system must monitor the relevant parameters (location, altitude) to determine when an aircraft's state no longer matches the condition that triggered its addition to the buffer.

4.0 – Toronto AD

The Toronto AD program manages the airspace around the airport up to a radius of 8 nautical miles and up to a ceiling of 3000 feet. It tracks aircraft within this zone and automatically sends control transfer messages when an aircraft crosses the boundary of this airspace, either by moving away from the airport and crossing the 8nm boundary or by climbing through the 3000ft ceiling.

Since the program uses DDS for real-time data handling and subscribes to state updates from aircraft to determine their positions relative to the zone boundaries, the DDS implementation involved defining various components.



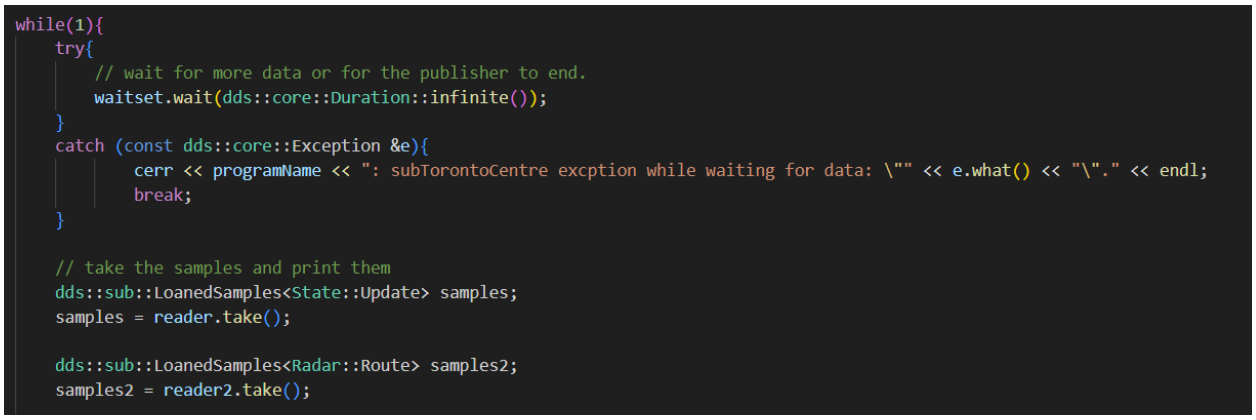
```
programName = argv[0];

// create the main DDS entities Participant, Topic, subTorontoAD and DataReader
dds::domain::DomainParticipant participant(domainID);
dds::topic::Topic<State::Update> topic(participant, "Flights");
// create DDS for TorontoCentre
dds::topic::Topic<Radar::Route> topic_AD(participant, "TorontoAD");
dds::topic::Topic<Radar::Route> topic_TC(participant, "TorontoCentre");

dds::sub::Subscriber subTorontoAD(participant);
dds::pub::Publisher publisher(participant);

dds::sub::DataReader<State::Update> reader(subTorontoAD, topic);
dds::sub::DataReader<Radar::Route> reader2(subTorontoAD, topic_AD);
dds::pub::DataWriter<Radar::Route> writer(publisher, topic_TC);

cout << "**** torontoCentre waiting for messages" << endl;
dds::core::cond::WaitSet waitset;
dds::core::cond::StatusCondition rsc(reader);
rsc.enabled_statuses(dds::core::status::StatusMask::data_available()| dds::core::status::StatusMask::subscription_matched());
waitset.attach_condition(rsc);
```

Figure 9: Cyclone DDS Settings for Toronto AD

The domainID is set to our team number 'Team25' and is how the DDS domain is set for our specific simulation within CycloneDDS. Then the topic entities are defined of which for the simulation including "Flights" for all the flight data along with the zones "TorontoAD" and "TorontoCentre". In addition, subscribers and publishers are also set, the subscriber specific to the program the DDS simulation is set in, so for the example above – set to subTorontoAD to represent the TorontoAD subscriber.

Then the DataReader objects 'reader' and 'reader2' are set up for receiving data from the "Flights" topic and its own "topic_AD" topic so it can see the respective terminals for each of these topics and pick up the data where needed. And finally, the DataWriter is set up to publish data in the "TorontoCentre" topic so it can send alerts when needed.

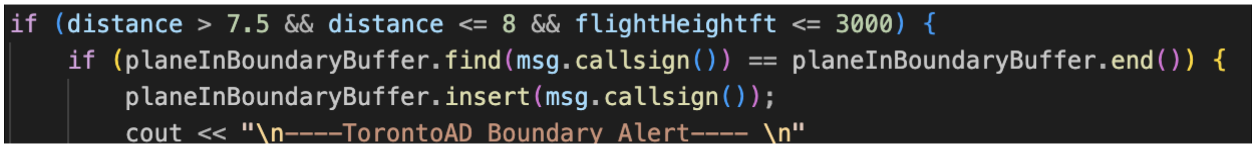
The rest of the code is set-up to wait for the specific conditions of whether the plane is moving about in a zone or between zones. It does so through the use of WaitSets to set up an asynchronous wait and through the While (1) loop, which routinely checks whether the State::Update and Radar::Route match those conditions.



```
while(1){
    try{
        // wait for more data or for the publisher to end.
        waitset.wait(dds::core::Duration::infinite());
    }
    catch (const dds::core::Exception &e){
            cerr << programName << ": subTorontoCentre excption while waiting for data: \"" << e.what() << "\"." << endl;
            break;
    }

    // take the samples and print them
    dds::sub::LoanedSamples<State::Update> samples;
    samples = reader.take();

    dds::sub::LoanedSamples<Radar::Route> samples2;
    samples2 = reader2.take();
```

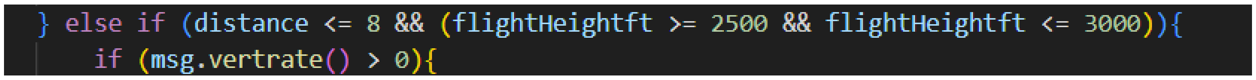
Figure 10: Asynchronous Wait with readers on Toronto AD

If there is no more data to process, the code breaks. However, while data is processing, there will first be a check to see whether a message from Toronto Centre has been received alerting a plane is approaching the airport. If that is the case, it prepares for the transfer through the planeInBoundaryBuffer mechanism and tracks the plane. If it is leaving the airport the plane distance is calculated to track when it will cross the boundary. Once the boundary is cross the TorontoAD program displays the Boundary Alert sent to Toronto Centre to state the plane is leaving the airport zone. The first boundary condition is if the plane is approaching the 8nm and below 3000ft (Figure 11) or if the plane is within the 8nm boundary and the aircraft is above the 2500 feet with a positive vertical rate (Figure 12).



```
if (distance > 7.5 && distance <= 8 && flightHeightft <= 3000) {
    if (planeInBoundaryBuffer.find(msg.callsign()) == planeInBoundaryBuffer.end()) {
        planeInBoundaryBuffer.insert(msg.callsign());
        cout << "\n----TorontoAD Boundary Alert---- \n"
```

Figure 11: Toronto AD Boundary Conditions 1



```
  } else if (distance <= 8 && (flightHeightft >= 2500 && flightHeightft <= 3000)){
      if (msg.vertrate() > 0){
```

Figure 12: Toronto AD Boundary Conditions 2

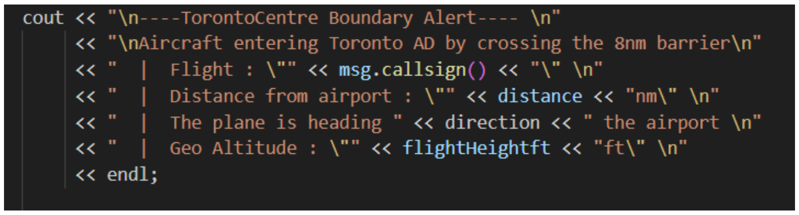
Otherwise, if there are no samples to read from or the readers' publisher is no longer available, the program breaks.

5.0 – Toronto Centre

The Toronto Centre program oversees the enroute airspace, which covers areas beyond the immediate approach and departure zones around the airport. Like Toronto AD, it monitors aircraft entering its airspace from the Toronto AD zone and manages the handoff process, ensuring that aircraft transitions between zones are smooth and efficient. The program handles control transfer messages indicating when an aircraft is moving into its airspace, either by descending through the 3000ft ceiling or approaching the 8nm boundary from outside.

The DDS program for Toronto Centre is the pretty similar to the TorontoAD code. With the key difference being that the program focuses on when the plane is entering the airport rather than leaving, so the boundary conditions are slightly different.

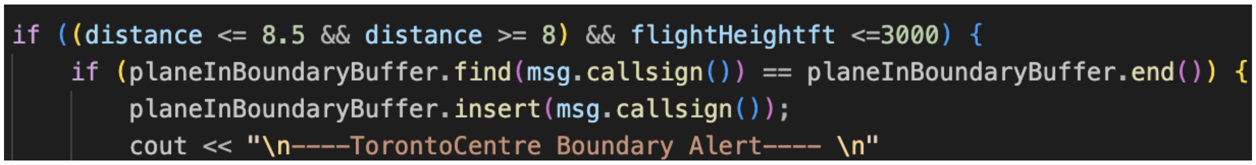
While the data is processing, the program first checks the TorontoAD writer to check if there are any planes leaving the airport and currently approaching the Toronto Centre zone. Again, using the planeInBoundaryBuffer mechanism to track the plane. If the plane is instead, entering the airport, then the plane distance is calculated, and the Toronto Centre Boundary Alert is prepared.



```
cout << "\n----TorontoCentre Boundary Alert---- \n"
    << "\nAircraft entering Toronto AD by crossing the 8nm barrier\n"
    << "  |  Flight : \"" << msg.callsign() << "\" \n"
    << "  |  Distance from airport : \"" << distance << "nm\" \n"
    << "  |  The plane is heading " << direction << " the airport \n"
    << "  |  Geo Altitude : \"" << flightHeightft << "ft\" \n"
    << endl;
```

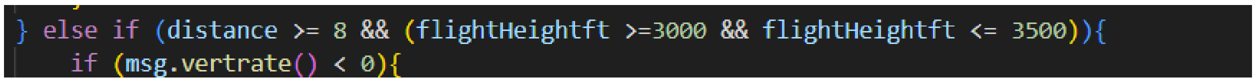
Figure 13: Toronto Centre Boundary Alert

The boundary cross for the Toronto Centre is then opposite to the boundary conditions of TorontoAD. TorontoCentre first checks if the plane is approaching the 8nm boundary and below 3000ft (Figure 14) and if not then it checks if the plane is within 8nm, approaches the 3000 ft boundary, and if the plane's vertical rate is negative (Figure 15). Shown below:



```
if ((distance <= 8.5 && distance >= 8) && flightHeightft <=3000) {
    if (planeInBoundaryBuffer.find(msg.callsign()) == planeInBoundaryBuffer.end()) {
        planeInBoundaryBuffer.insert(msg.callsign());
        cout << "\n----TorontoCentre Boundary Alert---- \n"
```

Figure 14: Toronto Centre Boundary Conditions 1



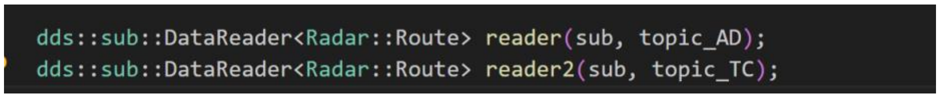
```
} else if (distance >= 8 && (flightHeightft >=3000 && flightHeightft <= 3500)){
    if (msg.vertrate() < 0){
```

Figure 15: Toronto Centre Boundary Conditions 2

Then, the message is sent to AD through the Toronto Centre writer, which alerts AD on its terminal instead.

6.0 – Query

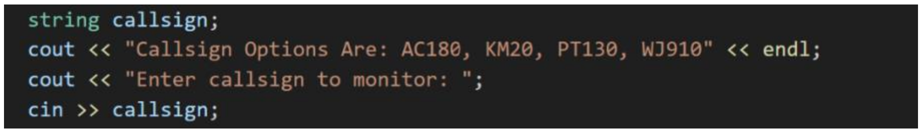
The 'query.cpp' program is used to query the current state of a specific flight based on its call sign. It listens and processes zone transfer messages related to a specific flight. To do so, it subscribes to messages published to the two topics (topic_AD and topic_TC), which correspond to transfer messages sent by torontoAD and torontoCentre, shown in Figure 16 below.



```
dds::sub::DataReader<Radar::Route> reader(sub, topic_AD);
dds::sub::DataReader<Radar::Route> reader2(sub, topic_TC);
```

Figure 16: Query.cpp subscribing to messages published to topic_AD and topic_TC.

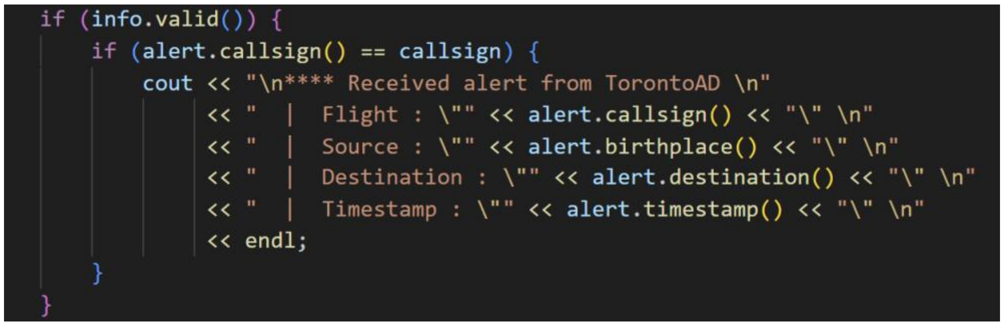
When running the program, the user will be prompted to enter the callsign of the flight it wants to monitor, shown in Figure 17. The options for callsigns in the dataSorted.csv file are also included for convenience.



```
string callsign;
cout << "Callsign Options Are: AC180, KM20, PT130, WJ910" << endl;
cout << "Enter callsign to monitor: ";
cin >> callsign;
```

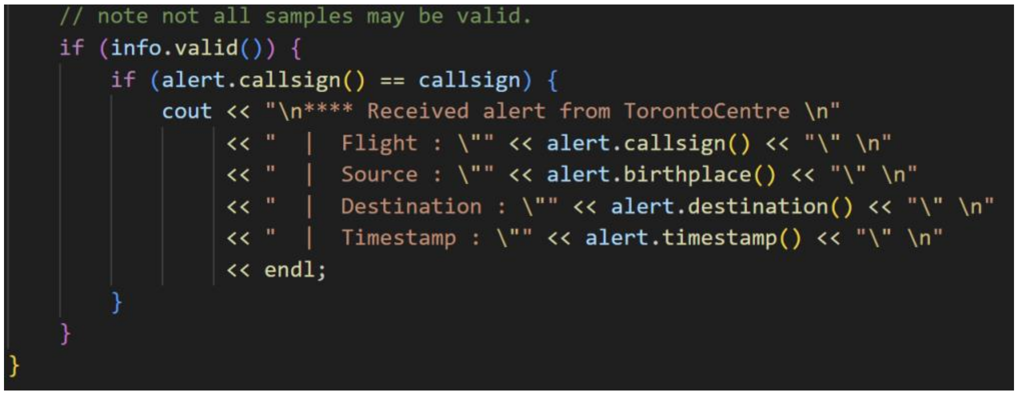
Figure 17: Prompting user to input callsign to monitor.

When a transfer message is sent with the callsign indicated by the user, it will be printed to the terminal. The code to do this is shown in Figure 18 and Figure 19 below.



```
if (info.valid()) {
    if (alert.callsign() == callsign) {
        cout << "\n**** Received alert from TorontoAD \n"
             << "  |  Flight : \"" << alert.callsign() << "\" \n"
             << "  |  Source : \"" << alert.birthplace() << "\" \n"
             << "  |  Destination : \"" << alert.destination() << "\" \n"
             << "  |  Timestamp : \"" << alert.timestamp() << "\" \n"
             << endl;
    }
}
```

Figure 18: Output of a zone transfer alert received for a specified flight in torontoAD, showing the flight's callsign, source zone, destination zone, and the timestamp of the alert.

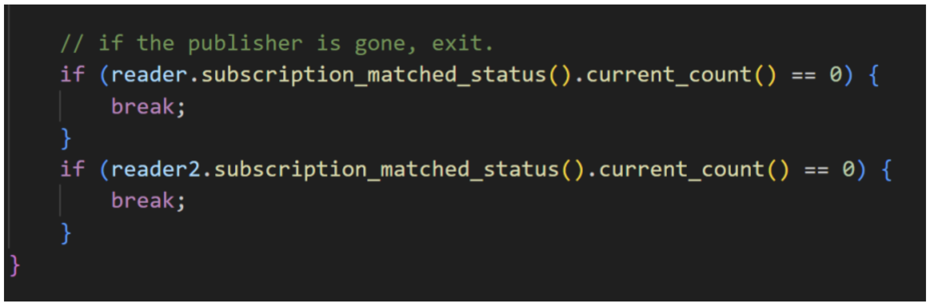


```
    // note not all samples may be valid.
    if (info.valid()) {
        if (alert.callsign() == callsign) {
            cout << "\n**** Received alert from TorontoCentre \n"
                 << "  |  Flight : \"" << alert.callsign() << "\" \n"
                 << "  |  Source : \"" << alert.birthplace() << "\" \n"
                 << "  |  Destination : \"" << alert.destination() << "\" \n"
                 << "  |  Timestamp : \"" << alert.timestamp() << "\" \n"
                 << endl;
        }
    }
}
```

Figure 19: Output of a zone transfer alert received for a specified flight in torontoCentre, showing the flight's callsign, source zone, destination zone, and the timestamp of the alert.

The monitoring loop within 'query.cpp' runs indefinitely and is always listening for new messages that match the specified callsign. However, it will terminate when there is an absence of publishers to the subscribed topics, which indicates the simulation has ended. This is shown in Figure 20 below.

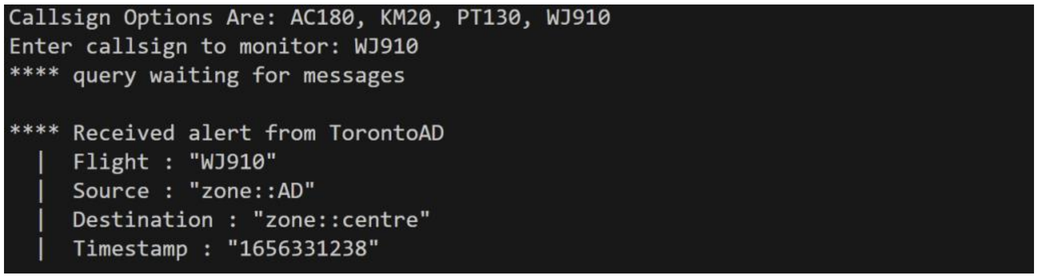
Aidan Kealey - 20151256
Kailey Fejer - 20182434
Paulina Flores - 20152096



```
    // if the publisher is gone, exit.
    if (reader.subscription_matched_status().current_count() == 0) {
        break;
    }
    if (reader2.subscription_matched_status().current_count() == 0) {
        break;
    }
}
```

Figure 20: Condition for terminating the query.cpp program.

The terminal output for flight WJ910 is shown in Figure 21 below.



```
Callsign Options Are: AC180, KM20, PT130, WJ910
Enter callsign to monitor: WJ910
**** query waiting for messages

**** Received alert from TorontoAD
  |  Flight : "WJ910"
  |  Source : "zone::AD"
  |  Destination : "zone::centre"
  |  Timestamp : "1656331238"
```

Figure 21: Terminal output when running query.cpp for callsign WJ910.

2.0 – Testing

This code was tested using the dataSorted.csv file which was called by the flights.cpp file. This file contains a dataset showing various flight parameters, where each row represents a snapshot of an aircraft's state at a specific timestamp. It provides information like its position, movement, and identification. Based on this, 4 test cases were explored, and the results are shown in the accompanying testing document. The testing document also provides examples of output form the query.cpp program.