Implementation of Consensus Control Using Lambda-Edge Architecture

Sean Ackels, Student Member, IEEE

Abstract—Swarm robotics are an important field to researchers because of their ability to perform complex tasks that no individual in the swarm could accomplish otherwise. They have the advantage of being able to cover a larger area and collect more data, but at the cost of a reliance on networked communications that increase delay in the system. Because of this reliance on networked communication, swarm robotics can be easily adapted to work with cloud based controls, which offer far more computational power to the system. Cloud controls have been further improved by the use of unique architectures such as the Lambda-Edge architecture, which splits tasks into batch and speed components that are only partly run on the cloud in order to increase responsiveness. This paper proposes a cloud control system that uses the Lambda-Edge architecture in order to control a swarm that is capable of learning an unknown sensor function and distributing individuals based on it. The control system is then proven using a Gazebo simulation.

Index Terms—cloud computing, edge computing, lambda architecture, consensus control, swarm robotics

1 INTRODUCTION

In the modern era, the field of robotics is a major player in research and industry, with new technologies coming out every year that open up new opportunities to implement robotics into a wider array of applications. One such technology that has seen continuous improvements in recent years is swarm robotics. A robotic swarm is a group of robots that work together to complete a task. Robotic swarms can collect more data and cover a larger area than an individual, and because of this are able to complete complex tasks quicker. Some examples of tasks that have been accomplished using swarms are rescuing passengers of a sinking ship, scanning an area for people, and tracking a moving target [1]. A swarm is able to complete these tasks by having its individuals form a consensus on the solution to a problem. The faster a consensus can be found and the more accurate it is, the better the swarm will perform.

Consensus controllers are responsible for forming the consensus in a swarm. A consensus controller functions by sharing information between members of the swarm then making a decision based on that information in order to solve the problem [2], [3]. Consensus controls benefit from being very adaptable to varied applications, for instance they can be run non-linearly for complex and accurate applications or linearly for simple and quick ones [4], [5]. In addition to linearization, a consensus controller can also be centralized or decentralized. A centralized consensus controller forms a solution on one central system then spreads that single solution to all individuals [6]. Because there is only one solution generated, all individuals will always have the same solution and therefore reach a consensus. This comes at the cost of having to ensure that each individual can connect to the central system, which can be difficult in many scenarios. Decentralized controllers use only information from neighboring individuals and is computed on board each member of the swarm [7]. This allows them to function even if each member of the swarm cannot connect to all other members, but at the cost of multiple solutions being developed for the problem.

Consensus controls suffer from other drawbacks besides those mentioned previously, specifically networked related issues such as latency and packet loss. These issues are unavoidable because of the need for each individual in the system to communicate, whether between each other or with a central controller. Latency is a delay caused by the time it takes to transmit a message between two points connected by a network. The increased delay caused by latency can lead to a controller becoming unstable and also increases the response time to disturbances in the system. Packet loss is caused by a packet sent over the network not reaching its destination, and can be interpreted as an infinite delay for that packet. The user must decide how to handle a dropped packet by either resending the packet or skipping it [8]. These issues must be taken into account when designing a consensus controller otherwise it will run the risk of performing poorly or becoming unstable.

Cloud computing is the act of sending computational problems through the internet to a separate computer to be processed, then receiving the solution back through the internet for use. Some companies have turned this into a business by charging for users to connect and use their servers like a utility, but the computer can also be owned by an individual or group also [9]. Because the information is sent over a network, cloud computing is very simple to implement on a centralized consensus controller [6]. By doing this, it is possible to calculate more complex control laws then any individual in the system or edge device could while allowing one to decrease the computational time of the controller through tools such as parallelization. Besides the inherent costs of using a centralized consensus controller mentioned above, the only downside to using cloud computing is the potential cost if the user does not
have access to their own server, in which case they will have
to pay to use a companies.

There has been recent developments in cloud computing
that help alleviate issues such as latency and response time
in the form of the Lambda architecture \[10\]. The Lambda
architecture works by splitting the algorithm for solving
a problem into three layers, the batch, speed, and serving
layers. The batch layer is responsible for finding an accurate
solution to the problem, the speed layer finds a quick
solution to the problem, and the serving layer decides which
solution should be used and sends it to the swarm. By
splitting the algorithm in this manner, a system can respond
quickly to disturbances while still being able to accurately
solve the given problem. This architecture can be further
improved by moving the speed layer to an edge device,
called the Lambda-Edge architecture, so that it can have
an even faster response \[11\]. The drawbacks of these ar-
chitectures is the complexity they add to a system. It can be
difficult to determine the best way to split an algorithm, and
furthermore requires the implementation of two separate
algorithms for each layer.

This paper proposes a consensus controller to control a
robotic swarm and implements an improved Lambda-Edge
architecture to increase performance. The purpose of the
algorithm is to use data collected by each individual in the
swarm to learn and model an unknown sensor function and
place each individual near maximums in that model. An
example of a use case for such an algorithm is when a fire
department wants to monitor a fire. The fire department
will want to be able to view all areas affected by the fire, but
not have the manpower to spare to control the individuals
of the swarm. The consensus controller is centralized and
uses cloud computing to run the batch layer of the Lambda-
Edge architecture. Other research in similar areas focus on
improving the consensus controller itself, the performance
of distributed controllers, or other uses of a Lambda-Edge
based controller \[11\], \[12\], \[13\], \[14\].

The rest of the paper is structured as follows. Section 2
discusses basic topics needed for understanding the con-
troller followed by Section 3, which discusses the actual
controller architecture. Section 4 discusses the experimental
setup used to test the controller and the results of those tests.
Finally, section 5 discusses our conclusions and future work
to be done followed by a bibliography.

2 Background Information

2.1 Consensus Controllers

Consensus controllers have been researched for the past two
decades. In this time, ample documentation has been written
describing the benefits, faults, applications, and forms
of consensus controllers. As such this section will provide a
basic description and relevant design criteria for designing
a consensus controller.

A consensus controller is a controller that uses informa-
tion from multiple interconnected systems to find a solution,
also known as a consensus, to a given problem. The con-
troller can govern the entire swarm in a centralized model or
be calculated on each individual using information from its
neighbors in a decentralized model. For this paper, a central-
ized model was chosen because it allows for more accurate
results and the ability to be implemented easier with the
Lambda-Edge architecture. The controller in this paper is
based on that proposed in \[15\], which uses a decentralized
controller and a different modeling method. The design of
the controller took several criteria into account:

- Robustness
- Computational Power
- Scalability
- Parallelizability
- Bandwidth

The controller should be robust enough to continue func-
tioning in non-ideal conditions such as unforeseen network
delay, frequent packet drops, and loss of signal. Further-
more, it must be able to work with a range of devices with
varying computational ability, thus forcing the design to be
realistic and achievable by most users. The design of the
controller should be as scalable as possible to allow it to
function on large scale systems. The part of the controller
running on the cloud should be made parallelizable so that
it may take full advantage of cloud computing abilities.
Lastly, transmitted data should be kept as minimal as pos-
sible in order to prevent over-saturation of the network,
which would decrease the performance of the system. These
criteria are important to consider when designing a consen-
sus controller to ensure that it can function properly and
reliably.

2.2 Cloud Computing

Cloud computing is a recent development in control theory
that is still being developed by researchers and businesses.
This paper is based on a definition that goes as follows;
cloud computing is the use of a separate computer over a
network to perform calculations for a system. Unlike some
definitions, such as the one provided in \[9\], the owner of
the computer is irrelevant and can be maintained by large
businesses charging a fee or by the user themselves. The
purpose of cloud computing is to allow a user to perform
complex computations that would otherwise be too slow or
impossible on board a system. Cloud computers are able to
process large amount of data much quicker than traditional
systems because they can easily parallelize computations.
This also means that certain applications do not benefit as
much as others based on their ability to be parallelized. Fur-
thermore, cloud systems can be designed to automatically
scale their size to fit a problem, which means that applica-
tions that can be scaled benefit more from cloud systems.
These benefits allow cloud computing to be very useful in
the field of swarm controls. The ability to parallelize the
computations needed for each individual and the ability for
the system to scale means that the time it takes to calculate
values for one individual in the system is roughly similar
to calculating the whole swarm. Furthermore, the inherent
delay in sending data to a cloud server can be accounted
for easily in controllers used for swarms because they must
already account for network delay between systems.

2.3 Lambda and Lambda-Edge Architectures

The Lambda architecture is designed to improve the re-
response time of a cloud system by splitting an algorithm into
two layers, the batch and speed layer, with a third serving layer to provide the solutions to the user. These three layers are all run in the cloud in parallel. The batch layer is responsible for calculating accurate answers using large amounts of data. Because the batch layer is very time consuming, it is not possible to use it for real time applications such as controls. To make up for this shortcoming, the speed layer is used to provide quick, less accurate solutions to the problem using only a subset of the data. Lastly, the serving layer takes in the output of both layers and makes them available for the user to query. By using solutions provided by the speed and batch layers, a cloud computer is able to be both fast and accurate. The Lambda architecture’s main drawback is the added complexity it requires to be implemented. The user must create two potentially completely different algorithms for their system along with the ability to decide which solution to query for.

The Lambda-Edge architecture is a recently developed improvement to the Lambda architecture, which moves the speed layer from the cloud to an edge device and splits the serving layer between the two. The purpose of this move is to allow the user to query the speed layer quicker by reducing the latency of the read. This allows the system to react even faster to stimulus than the original Lambda architecture at the cost of being more complex by forcing the user to read information from two locations. This architecture is strongly dependent on the capabilities of the edge device and make the assumption that it is capable of calculating the speed layer. If this assumption is false, the system can be slower than the traditional Lambda architecture because the time lost from querying an edge device over the cloud is less than the time gained from the drop in computational ability of the edge device. Figure 4 models the Lambda-Edge architecture. The model shows how the speed and batch layers are on different devices but running concurrently, with the final serving layer being on all levels of the system. The same model can be used for the Lambda architecture by simply changing the color of the speed layer to red.

This paper makes a further improvement to the Lambda-Edge architecture by further splitting the speed layer between the edge and end devices. Traditionally, end devices have more computational power than the edge, therefore it makes sense to utilize some of that ability to improve the speed layer. Once could also move the entire speed layer onto each device depending on the application. The actual implementation of this improvement will be discussed further below.

3 Proposed Controller Architecture

The proposed controller was developed to solve a sensor placement optimization problem that uses drones as the sensors. The goal is to distribute each drone around maximums in a sensor function. The sensor function is unknown by the controller when it begins, so it must use sensor values from each individual to model the sensor function. The model is then used to determine the placement of each drone, who are responsible for receiving the coordinates and planning a path to them. Figure 5 demonstrates an example scenario for this controller. In the figure it can be seen that the centroid is different from the location of each individual, and is biased in the direction of the maximum of the sensor function at the right end. After some time, each robot will move in the direction of each centroid, which will change the Voronoi diagram and therefore the centroids. Once each robot reaches the centroid of its region and the Voronoi diagram remains constant, then the system has reached equilibrium and each individual will be placed around the maximum of the sensor function. In the given example the final placement would be each individual placed near the right side.

The controller has two main components; updating each robots position and updating the model used to determine each position. Both are run in parallel and updated as the system continues to gather sensor data from the environment. The following subsections will go into detail about how each component is run.

3.1 Updating the Robots Position

Each drone in the swarm will be responsible for calculating its next position. The desired position is the centroid of the Voronoi region of the drone based on the modeled sensor function, which is from the method developed in [15]. A Voronoi region of a point is the region where all positions are closer to the point then any other significant points. In this case, other significant points would be the positions of other drones in the swarm. The centroid in this scenario is synonymous with the center of mass of an object, where the mass of the Voronoi region is represented by the modeled sensor function and from it the total mass and the moment are found. These values are then used to find the center of mass of the Voronoi region, which is referred to as the centroid in this paper. When the drone reaches the centroid of the Voronoi region and the system as a whole reaches an equilibrium then the placement of each sensor has been found. As the drone moves, it will publish its position and a reading of the actual sensor function to be used to improve the model of the sensor function. Algorithm 1 shows how these actions are performed on each drone.

```
Algorithm 1: Algorithm to Update Position

Input: Positions, Model or Desired Positions
Output: Position, Data
1 Determine if batch layer was received;
2 if Batch layer was not received then
3 Update Voronoi plot based on positions;
4 Pull Voronoi regions vertices and use to calculate centroid based on modeled sensor function;
5 Use centroid to calculate next positions;
6 Begin moving to calculated position;
7 else
8 Begin moving to received position;
9 Publish current position with real sensor function;
```

This algorithm makes a decision on what to do based on if the batch layer was received. If it is not received, it will use the linear model to calculate its next position and then begin moving to it. If it is received then it will instead skip the speed layer and immediately begin moving to the position. This decision represents the improvement to the Lambda-Edge architecture this paper is testing; the batch layer not
only updates its model but calculates the position for the individual while the speed layer updates its model on an edge device then calculates the position of the individual itself. In order to convert this system to a Lambda-Edge or Lambda architecture, one must only move where each component is calculated to a different device.

The implementation of the Voronoi region is based on Scipy’s Voronoi solver that is bounded to a region determined by the user. The mass and moment of each drone’s Voronoi region is found using Monte-Carlo integration implemented through Sci-Kit Monaco. Monte-Carlo integration was chosen because it is simple to test if a point is in the region or not, as opposed to traditional integration which would need separate integrals for each vertice of a Voronoi region. The integration is further sped up by dynamically changing the bounds of the generated points to a box around the current Voronoi region instead of the whole space.

3.2 Updating the Modeled Sensor Function

The model of the sensor function is crucial to ensuring that a sufficient distribution of the drones is reached. There is a linear and non-linear model that is used for this component due to the fact that it is split for the Lambda-Edge architecture. Data for the models is collected from each individual in the swarm, with recent data being used by the linear model and all recorded data being used by the non-linear model. The current implementation of the controller makes the assumption that the sensor function will not significantly change or move during the course of the trial, which is why all previous information can be used. Both approximations will be described in depth below.

3.2.1 Linear Approximation

Algorithm 2: Algorithm to Update Linear Approximation Model

<table>
<thead>
<tr>
<th>Input: Positions, Data, Desired Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: Model</td>
</tr>
</tbody>
</table>

1. Publish compiled list of positions and sensor data;
2. Calculate Linear Approximation;
3. Publish linear model;

Algorithm 2 demonstrates how the linear approximation is performed. This algorithm will be run on an edge device and represents half of the speed layer in the Lambda-Edge architecture. It is responsible for gathering the current positions of all individuals in the swarm for use by itself and the non-linear approximation. The linear approximation uses only the most current data from each individual and is intended only to provide a general trend in the data with the goal of moving individuals close enough to their final position. The serving layer makes its decision based off of the value of the desired positions. By using the value of the batch layer to determine whether it should be sent, the algorithm avoids having to send another variable that states whether it has reached equilibrium or not.

The linear approximation uses an ordinary least squares formulation to find a plane that best fits the current data received from individuals in the swarm. Ordinary least squares works by minimizing the sum of squared residuals, where the residuals are the difference between the calculated plane

\[ ax + by + c = z \]
and each actual value $z_i$ as shown in equation (1).

$$E(a, b, c) = \sum_{i=1}^{m} [(ax_i + by_i + c) - z_i]^2$$ (1)

The final goal is to solve for the constants $a$, $b$, and $c$, which will provide the plane that best fits the given data. The equations pertaining to solving these variables are listed below:

$$A \begin{bmatrix} a \\ b \\ c \end{bmatrix} = B;$$ (2)

$$A = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots \\ x_m & y_m & 1 \end{bmatrix}, \quad B = \begin{bmatrix} z_0 \\ z_1 \\ \vdots \\ z_m \end{bmatrix}$$

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = (A^T A)^{-1} A^T B$$ (3)

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} \sum_{i=1}^{m} x_i^2 & \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} x_i \\ \sum_{i=1}^{m} x_i y_i & \sum_{i=1}^{m} y_i^2 & \sum_{i=1}^{m} y_i \\ \sum_{i=1}^{m} x_i & \sum_{i=1}^{m} y_i & 1 \end{bmatrix}^{-1} \begin{bmatrix} \sum_{i=1}^{m} x_i z_i \\ \sum_{i=1}^{m} y_i z_i \\ \sum_{i=1}^{m} z_i \end{bmatrix}$$ (4)

Equation (2) shows the matrix form of the plane, where $A$ and $B$ represent the matrices themselves. The number of rows in each matrix is equal to $m$, the amount of data points provided. The next step is to perform a pseudo left inverse on matrix $A$, as shown in equation (3). This inverse is used because matrix $A$ typically isn’t a square matrix.

Equation (4) finally substitutes the true values of $A$ and $B$ back into equation (5).

### 3.2.2 Non-Linear Approximation

**Algorithm 3: Algorithm to Update Non-Linear Approximation Model**

**Input:** Positions, Data  
**Output:** Desired Positions  

1. Calculate updated non-linear model using new data;  
2. Determine if model has reached equilibrium;  
3. if Equilibrium = TRUE then  
4. Calculate optimal positions for all individuals;  
5. Publish calculated positions;  
6. else  
7. Publish None

Algorithm (3) demonstrates how the non-linear approximation is performed. The data received from the edge, which consists of the current position and sensor reading for each position of each individual, is added to a list of all positions and sensor readings collected by the individuals. It was decided that all data would be used instead of a recent subset because of the assumption that the sensor function is not changing over time and as such all data would be relevant. Furthermore, there is an assumption that storage is sufficiently large to store all data collected from the system. The test for equilibrium in the model is based on how much the model changes between iterations. The change is calculated by subtracting the new model from the previous, taking the absolute value of the change, then testing if any part is above the threshold. If not, then the system has reached equilibrium and can be used by each individual. Other possible criteria that can be used to determine whether the batch data is used are the number of input samples or the difference between the modeled sensor function and the most recent input data.

The positions for each individual are found using a simulator. The most recent points are used as a starting position, and the new positions are calculated using the same method as described in (1). Some changes are made to the algorithm to help it converge faster than a true simulation by allowing each point to travel instantly to its desired position. This allows the simulation to finish much faster then one that aims to model a robots movements accurately. Once the simulation phase runs, the final positions of the swarm is returned to be published by the batch layer.

### 3.3 Lambda-Edge Implementation

Each layer for the Lambda-Edge architecture is required to model the unknown sensor function then decide on a position for each member of the swarm. The batch layer uses a non-linear approximation and calculates both on the cloud, while the speed layer calculates the linear model on the edge and the position on the end device. The decision to split the speed layer further instead of using the original architecture was to allow each individual to continuously calculate a new position as it needed it instead of waiting for each layer to finish processing their models to move. With this setup, the speed layer can run in parallel with the end device and update its model as it is completed. The individual then can use its most recent saved model to continuously calculate its position. This should be expanded in the future.

### 3.4 ROS Implementation

ROS is a middleware that is capable of handling messages between different peers in a peer-to-peer network. It works on a subscription basis, where a process called a node can subscribe to a topic that is published by another process. Nodes handle unprocessed data from a topic in an event-driven manner, meaning that data from topics is handled asynchronously. ROS also contains tools for synchronous communication, package management, data storage, and more. For this application, ROS is a good starting point for implementing and testing because it is easy to implement. Furthermore, ROS provides end-to-end control of the messages being passed, which is critical for ensuring that it accurately represents the improved architecture. For this scenario the cloud, edge, and end layers are all separate nodes in the system. Each one subscribes directly to the topic it needs information from as opposed to receiving information from the edge device. This is because the master node for the ROS system will be the edge device, so all topics must go through it regardless. All nodes will run in parallel and make decisions based on the most recent data received. Figure (3) below shows the setup of scenario with four individuals in the swarm.
4 EXPERIMENTAL RESULTS

4.1 Setup

4.2 Results

5 CONCLUSION

ACKNOWLEDGMENTS

REFERENCES


