ENSC 427: Communication Networks
Spring 2020

Final Project
NS-3 Simulation Model for LEO Satellite Networks

http://www.sfu.ca/~anysam/ensc427project.html

Anysa Manhas
301290035
anysam@sfu.ca

Amneet Mann
301275521
amneet_mann@sfu.ca

Jasmandeep Batra
301288075
jasmanb@sfu.ca

Team 11

Abstract

In this paper, a model for a low-Earth orbiting (LEO) satellite constellation network is constructed in the network simulator *ns-3*. To achieve this goal, a LEO Satellite Module is added to the existing code base. When given the number of orbital planes in the network, the number of satellites per orbital plane, and the altitude of the network, the module creates a LEO satellite network within a polar orbital configuration. This network is then used in simulations to analyze properties of LEO satellite constellations, such as average round trip time (RTT) and number of hops per network size and network altitude.

Primary results regarding LEO satellite constellation networks are obtained in this paper. Future work to be done involves building more robust LEO satellite constellation network models, and increasing the scope of simulations conducted.

Table of Contents

1 Introduction

Low-Earth orbiting (LEO) mega-constellations have been proposed by leading technology companies, such as SpaceX, Amazon, and OneWeb, with a focus on providing low-latency Internet service, particularly to regions that are not typically well-connected [1]. SpaceX is deploying StarLink, a constellation of nearly 12,000 satellites [2], to provide global coverage by 2021. Amazon's Project Kuiper will consist of 3236 satellites [3], and OneWeb will deploy a constellation of 588 satellites [4]. The Iridium satellite constellation, which provides voice and data connections, was the pioneer of LEO satellite networks. It became operational in 1998 and consists of 66 satellites at 780 km [5].

LEO satellites orbit the Earth at an altitude of 500-2000km [6]. Their lower orbit compared to geostationary satellites results in each satellite maintaining a steady orbital speed around the Earth, and also results in each satellite covering a smaller area of the Earth. Therefore, a network that aims to provide global network coverage requires many satellites to be communicating with each other and ground stations on the Earth, and also requires dynamic links to be attached and detached between the various nodes of the network as the satellites orbit. LEO satellite networks offer shorter propagation delays than terrestrial communication networks, such as fiber optic cables, while providing mobile and broadcast services [7].

Currently, there is no model present in the network simulator *ns-3* for simulating LEO satellite networks. In our project, we will design and implement a simple LEO satellite constellation and evaluate the latency of an uncongested network when communicating with a ground station. A LEO satellite network implementation requires LEO satellite nodes to be organized in orbital planes, dynamic link attachments and detachments as satellites orbit and the distance between satellites change, and routing of packets among ground stations through the network. After implementing the LEO satellite constellation, we will evaluate the network latency.

Previous work done in the area of modelling LEO satellite communication networks involve theoretical studies of design considerations of LEO satellite networks, and simulations of these networks. In his dissertation "Modelling Communications in Low-Earth-Orbit Satellite Networks," Peter Gvozdjak explored the design and use of LEO satellite networks, and issues related to building satellite networks. He pointed to key considerations when evaluating LEO satellite networks, such as constellation coverage, frequency issues, multiple access, methods, hand-offs, and inter-satellite network topology. Gvozdjak reviewed the two main orbital constellations: polar and inclined. The Iridium satellite network utilizes a polar orbit, with four links per satellite (two inter-plane and two intra-plane). He evaluated the tradeoff between lower altitude satellites, which provide lower latencies for transmission between ground stations and satellites, but also have a smaller footprint and thus require frequent hand-offs between ground stations [8].

In the paper "Delay is Not an Option: Low Latency Routing in Space," author Mark Handley implemented and evaluated a simulation of SpaceX's StarLink using details provided from SpaceX's public FCC filing. The simulated constellation included 1600 satellites with optical communication links. Routing was performed using Dijkstra's routing algorithm every 50 ms and caching the results for 200 ms into the future to determine which links would still be present when the packet reached those satellites. After evaluating multiple network configurations by modifying links between satellites, Handley concluded that the SpaceX network could provide faster Internet services than terrestrial networks for large distances [9].

2 Building a Network

To construct a LEO satellite constellation network, multiple structures which were not currently present in *ns-3* needed to be designed and implemented. Two mobility models were required: one for the LEO satellites in the constellation network, and one for the ground stations interacting with the LEO satellites. The mobility model for the LEO satellite in the constellation network must configure the initial position of the LEO satellites for a network, maintain the positions of the LEO satellites as time passes and they orbit around the Earth, and calculate the changing distances between satellites at specific points in time. The mobility model for the ground station must configure the position of the ground stations and must calculate the distance between themselves and the orbiting satellites at specific points in time.

A top level controller, the LEO Satellite Config model was also required for the LEO satellite constellation network. This model is responsible for maintaining and updating links between satellites, and between satellites and ground stations. Additionally, this top level controller is responsible for routing of packets between nodes in the network.

Fig. 1 below illustrates how the LEO Satellite Module fits within the existing base *ns-3* structure. *ns-3* is composed of interdependent modules that provide different functionalities to the simulator tool. We have extended the base *ns-3* with the addition of the leo-satellite module. The dependencies of the leo-satellite module are shown in red in the below figure. The figure also illustrates the various components residing in the leo-satellite module. Model components include both mobility models and the LEO Satellite Config model, and example components include the example script used to collect simulation data.

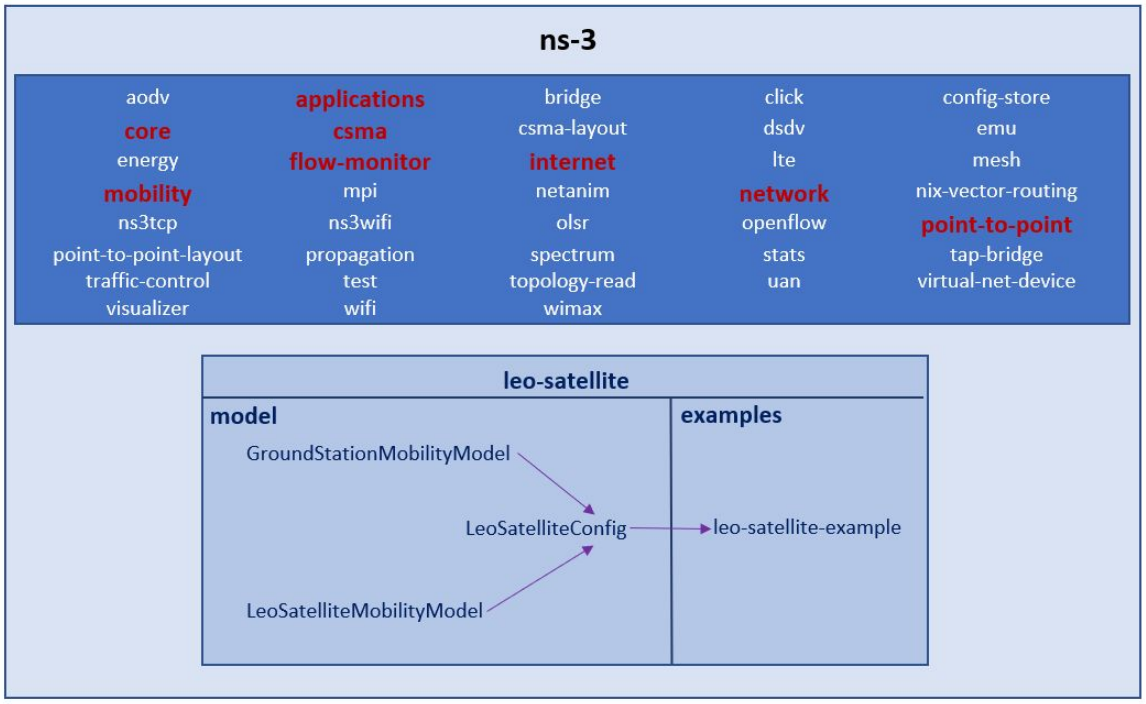

Figure 1: High-level overview of the leo-satellite module within the existing *ns-3*.

2.1 Mobility Models

2.1.1 LEO Satellite Mobility Model

The LEO Satellite Mobility Model is derived from the existing *ns-3* MobilityModel class. The LEO Satellite Mobility Model takes in the number of orbital planes in the network, the number of satellites per plane, the altitude of the satellite network, and the time at which the initial positions of the satellites were set. Fig. 2 below shows a high-level overview of the mobility model and its functionality.

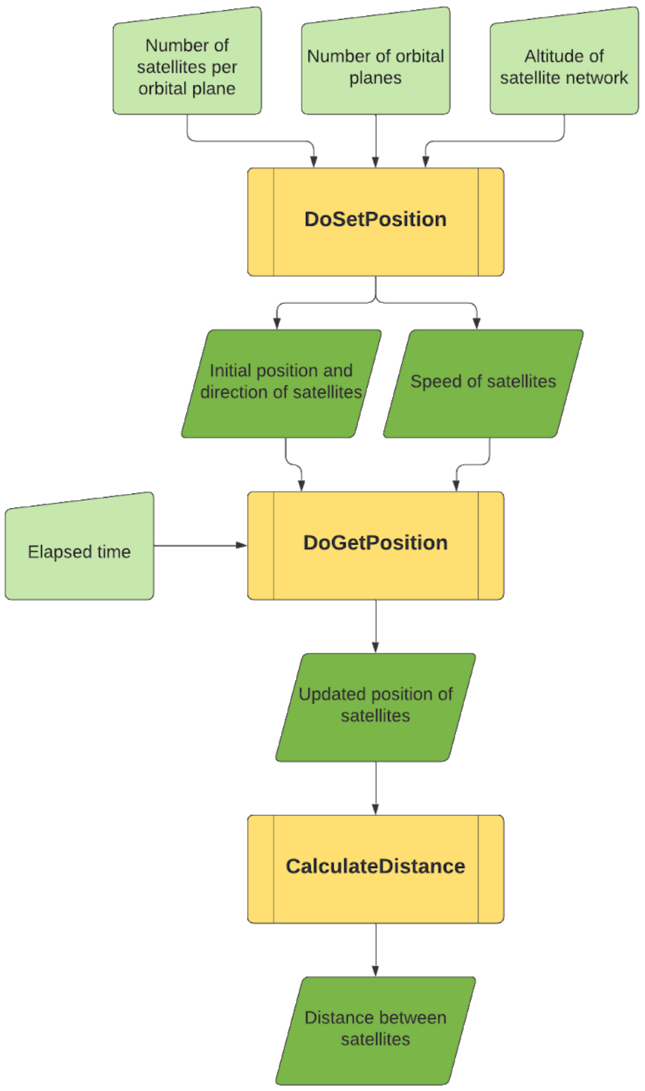
Number of satellites per orbital plane
Number of orbital planes
Altitude of satellite network
DoSetPosition
Initial position and direction of satellites
Speed of satellites
Elapsed time
DoGetPosition
Updated position of satellites
CalculateDistance
Distance between satellites

Figure 2: Functionality of LEO Satellite Mobility Model.

Given the number of planes and the number of satellites per plane, the DoSetPosition function configures orbital planes, and satellites within an orbital plane, equidistant from each other in a polar orbit configuration. Satellites in adjacent orbital planes move in opposite directions relative to each other.

Fig. 3 below shows an example of satellite positions configured by the LEO satellite mobility model for a network with nine orbital planes and eight satellites per orbital plane. As shown in Fig. 3, one full orbital plane consists of a longitude and the longitude 180 degrees apart from it (i.e., the longitudes -160^{o} and +20^{o} compose one full orbital plane) and all the intervening latitudes. The -180^{o} and +180^{o} longitudes are considered to be the same. Negative longitudes in Fig. 3 represent eastern longitudes, positive latitudes represent western latitudes, negative latitudes represent southern latitudes, and positive latitudes represent northern latitudes.

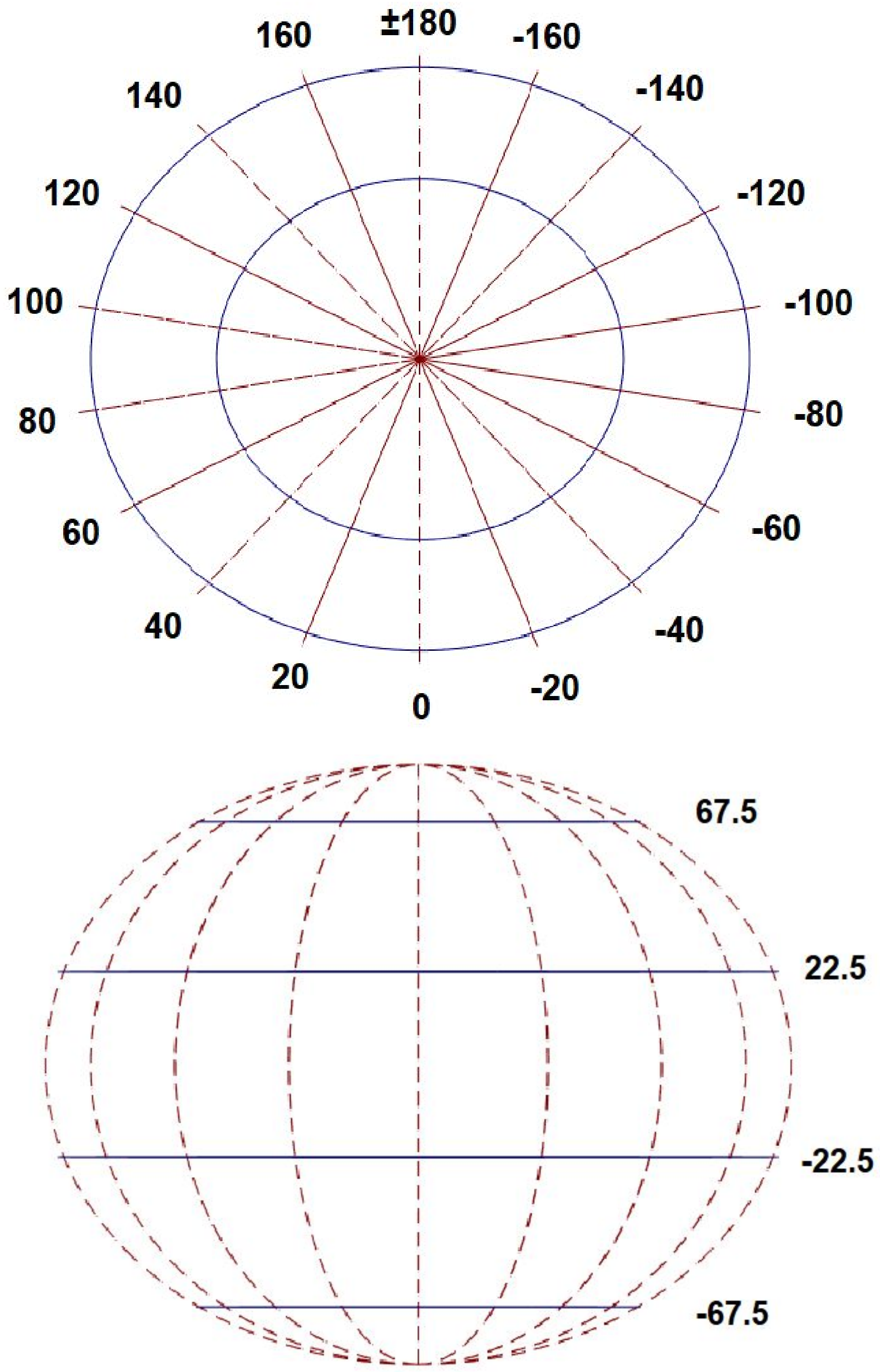

Figure 3: LEO satellite network topology with nine orbital planes and eight satellites per plane. Nodes are placed at the intersections of latitudes and longitudes shown.

At any point in the simulation time, the DoGetPosition function updates satellite positions based on initial satellite positions, the elapsed simulation time, and the speed of the satellites. It returns the updated satellite positions. The speed of the satellites is determined from their altitude using the following formula, where G is the gravitational constant ($6.673 \times 10^{-11} Nm^2/kg^2$), m_E is the mass of the Earth ($5.972 \times 10^{-24} kg$), and r is the radius of orbit [10].

$$v = \sqrt{\frac{G * m_E}{r}}$$

The CalculateDistance function calculates the distance between any two satellites using the Haversine formula, which is used to calculate the distance between two points on a sphere [11] using the latitudes ϕ and longitudes ψ of the two points. Haversine formula:

$$d = 2r\arcsin\sqrt{\sin^2(\frac{\phi_2 - \phi_1}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^2(\frac{\psi_2 - \psi_1}{2})}$$

2.1.2 Ground Station Mobility Model

The Ground Station Mobility Model is also derived from the existing *ns-3* MobilityModel class. The Ground Station Mobility Model takes in the number of orbital planes in the network to configure the position of a specified number of ground stations. Fig. 4 below shows a high-level overview of the mobility model and its functionality.

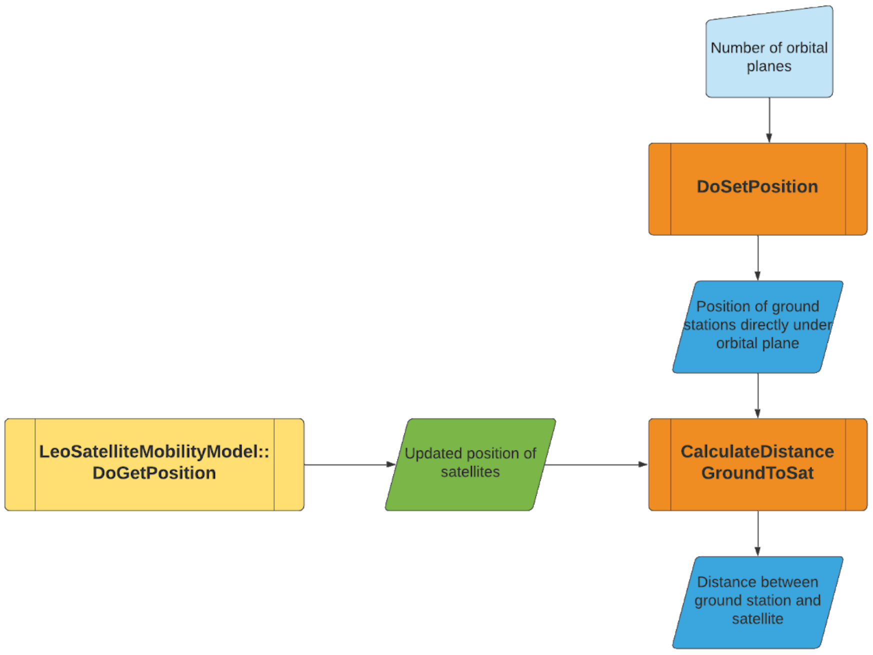
Number of orbital planes
DoSetPosition
Position of ground stations directly under orbital plane
LeoSatelliteMobilityModel:: DoGetPosition
Updated position of satellites
CalculateDistance GroundToSat
Distance between ground station and satellite

Figure 4: Functionality of Ground Station Mobility Model.

To simplify our implementation, the DoSetPosition function of the Ground Station Mobility Model currently configures two ground stations, and both ground stations are configured directly below satellite orbital planes. Fig. 5 below shows an example of ground station positions configured by the ground station mobility model for a network with nine orbital planes and eight satellites per orbital plane. As shown in Fig. 5, the ground stations are positioned strategically relative to each other such that a combination of inter- and intra-satellite links will be required to transmit a packet from one ground station to the other.

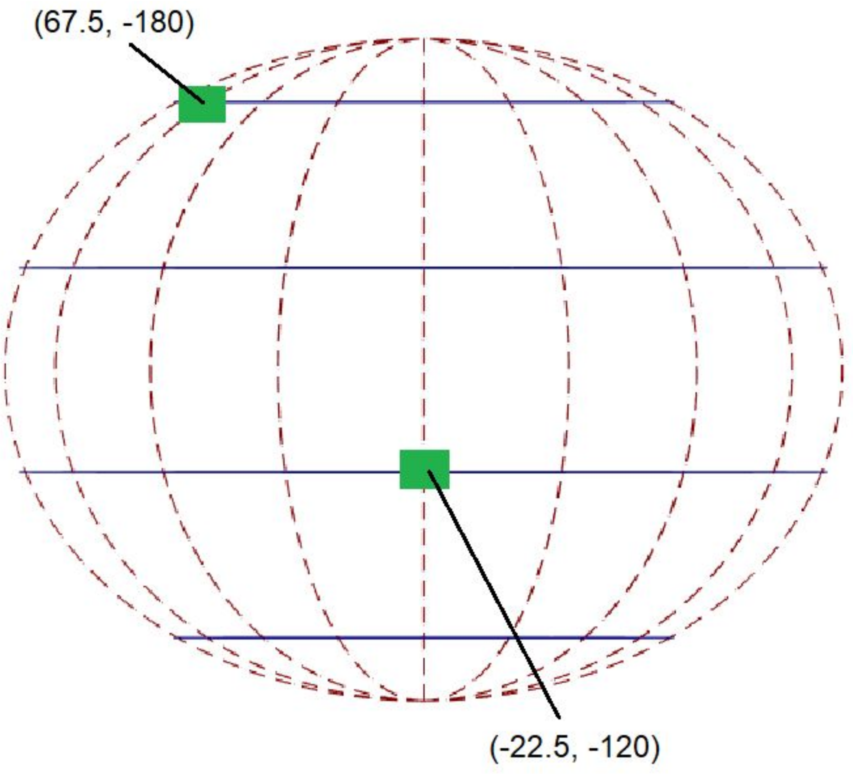

Figure 5: Configured position of two ground stations in LEO satellite network with nine orbital planes and eight satellites per orbital plane. Latitude and longitudes of ground stations are labelled.

The CalculateDistanceGroundToSat function calculates the distance between a satellite and the corresponding ground stations using the following formula, where the difference in latitudes between the satellite and ground station is represented by ϕ, and the difference in longitudes between the satellites and ground station is represented by ψ: [12]

$$d^2 = (R_s + R_e cos(\psi) cos(\varphi))^2 + (R_e sin(\psi) cos(\varphi))^2 + (R_e sin(\psi))^2$$

2.2 LEO Satellite Config Module

The LEO Satellite Config Module is the top level controller that maintains the complete LEO satellite network configuration — it takes in a given number of nodes, applies the relevant mobility models, and organizes these nodes into satellites in their respective orbital planes and ground station. The module implements two main additional functionalities: storing and updating the links between the various nodes in the network, and enabling the routing of packets through the networks. Fig. 6 below shows an overview of the top level controller module and how its tasks are organized. The bulk of the configuration occurs in the constructor of the LEO Satellite Config Module.

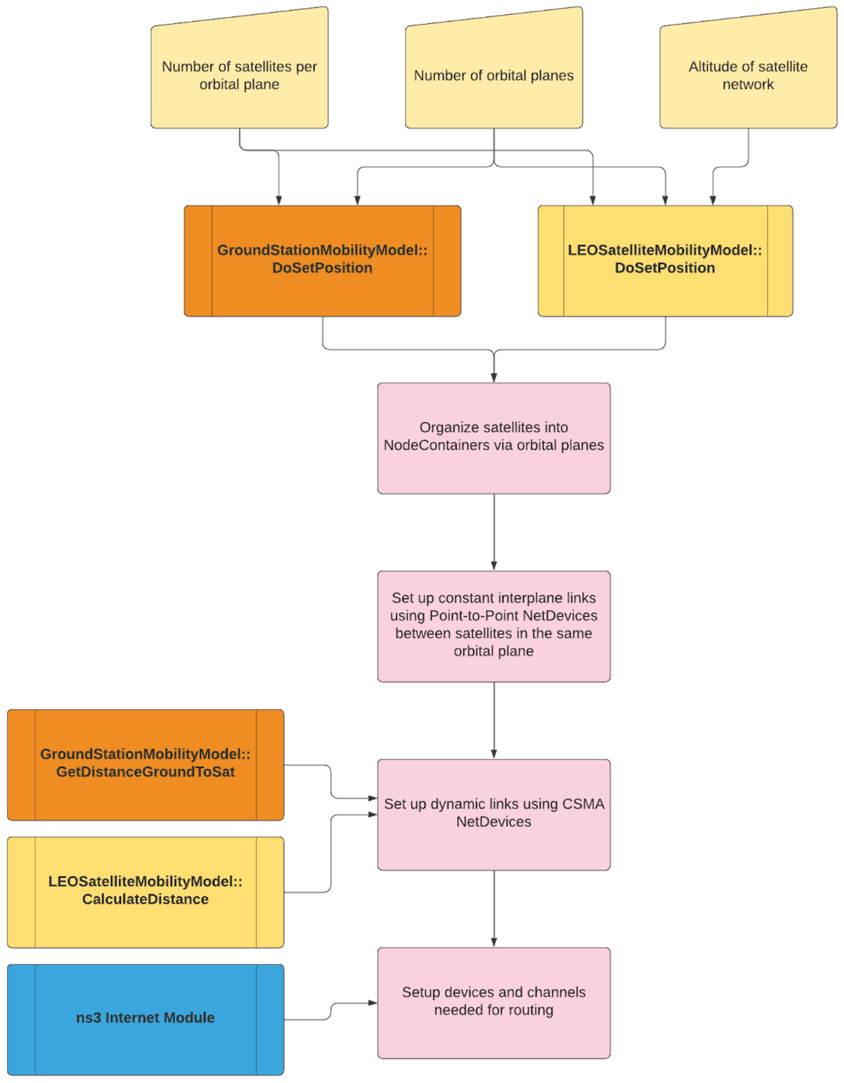
Number of satellites per orbital plane
Number of orbital planes
Altitude of satellite network
GroundStationMobilityModel:: DoSetPosition
LEOSatelliteMobilityModel:: DoSetPosition
Organize satellites into NodeContainers via orbital planes
Set up constant interplane links using Point-to-Point NetDevices between satellites in the same orbital plane
GroundStationMobilityModel:: GetDistanceGroundToSat
LEOSatelliteMobilityModel:: CalculateDistance
Set up dynamic links using CSMA NetDevices
ns3 Internet Module
Setup devices and channels needed for routing

Figure 6: Functionality of LEO Satellite Config module (top level controller).

2.2.1 Links between nodes

Each satellite in the LEO satellite constellation network has four active links to other satellites at any given point in time. Links between satellites can be static or dynamic, depending on the relative positions of the two satellites: two satellites within the same plane (intra-plane) have static links as the distance between those satellites do not change, two satellites in adjacent planes (inter-plane) have dynamic planes as these satellites move in opposite directions and find new satellites to create links with.

The initial configuration of links between satellites set up by the top level controller module is along the latitude and longitude lines — therefore the latitude and longitude lines in Fig. 3 can be viewed as an example of the initial inter- and intra-plane links for a network nine orbital planes and eight satellites per orbital plane.

As the satellites orbit across the Earth and ground stations remain stationary, hand-offs must occur when ground stations update which satellite they are connected to, to ensure they remain connected to the satellite closest to them. Therefore, links between ground stations and satellites are dynamic.

Intra-plane links, which are constant throughout the duration of the simulation, have been created between two nodes using *ns-3*'s Point-to-Point network devices [13]. Dynamic inter-plane satellite links, and links between ground stations and satellites, have been created using *ns-3*'s CSMA network devices [14]. CSMA network devices allow for dynamic links through attachment and detachment functions.

The UpdateLinks function in the LEO Satellite Config Module can be called periodically to update dynamic links. Pseudo-code describing the algorithm of the UpdateLinks function is shown below. The links are updated on a per-plane basis: for each plane, the satellite closest to the equator is chosen as the "reference satellite." The node displacement between the reference satellite and the closest satellite in the adjacent plane is used as an increment to automatically determine the closest satellite for every other satellite in the plane without evaluating the distance. This method of updating links ensures that no links are crossing over each other between the orbital planes, and that each satellite maintains exactly four connections to other satellites.

```
void LeoSatelliteConfig::UpdateLinks()
{
  For each plane:
    Find reference satellite /*satellite closest to the equator*/
    Find closest satellite to reference satellite in the plane east to the current
plane
    Calculate node displacement between reference satellite and its closest eastern
satellite
    For each node in this plane:
      Determine closest eastern satellite based on node displacement for reference
satellite
      If closest eastern satellite has changed since last update:
        Detach channel between node and its previous closest eastern neighbour
        Attach channel between node and new closest eastern neighbour
      Update link delay based on new distance

  For each ground station:
    Find closest satellite to ground station
    If closest satellite to ground station has changed since last update:
      Detach channel between ground station and previous closest satellite
      Attach channel between ground station and new closest satellite with link delay
    Update link delay based on new distance
}
```

Link delays used in the *ns-3* code for these links were calculated using the speed of light and the distance between nodes [15]. Data rates for all links was taken to be 5.36 Gbps, which was found to be a typical data rate value for LEO satellite constellation networks [16].

2.2.2 Routing

Routing was performed in the LEO Satellite Config Module using existing implementations of the IPv4 class in the Internet module in *ns-3* [17]. Fig. 7 shows a flow chart of how existing *ns-3* classes were used to perform routing.

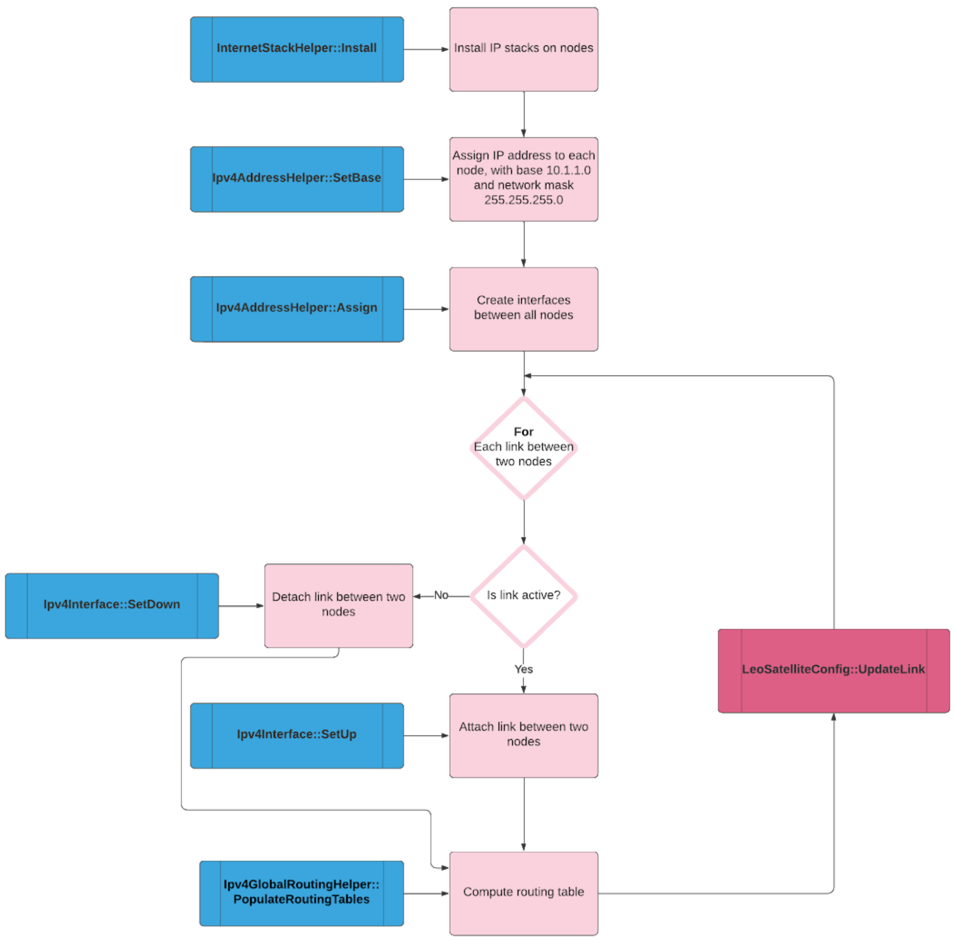
InternetStackHelper::Install
Install IP stacks on nodes
Ipv4AddressHelper::SetBase
Assign IP address to each node, with base 10.1.1.0 and network mask 255.255.255.0
Ipv4AddressHelper::Assign
Create interfaces between all nodes
For Each link between two nodes
Ipv4Interface::SetDown
Detach link between two nodes
No
Is link active?
Yes
LeoSatelliteConfig::UpdateLink
Ipv4Interface::SetUp
Attach link between two nodes
Ipv4GlobalRoutingHelper:: PopulateRoutingTables
Compute routing table

Figure 7: Routing functionality in LEO Satellite Config Module.

As shown in Fig. 7 above, the IPv4Interface SetDown and SetUp functions were used to attach or detach links in the case of dynamic links. The IPv4GlobalRoutingHelper was used to populate routing tables on each of the satellites. This method of routing assumes a global knowledge of the system and finds the shortest path to all other nodes in the system. Each time the dynamic links are updated, the routing table is recomputed.

3 Simulations and Data Collection

The simulations conducted in this project involved two ground stations and satellite networks of varying sizes, or at varying altitudes. The ground stations in the simulations were configured on opposite sides of the globe at different latitudes, such that a combination of inter- and intra-plane link traversals were required to transmit packets between the two. The UdpClientServer application [18] in *ns-3* was used: UDPEchoServerHelper and UDPEchoClientHelper were used to configure the two ground stations as client and server, and to send UDP packets between them.

Fig. 8 shows a flow chart for how simulations were set up and conducted to collect data. The network was first set up with LEO satellites, and two ground stations configured as a UDP client and a UDP server. The simulations were run for a total of 2000 seconds, with links being updated and packets being sent every 100 seconds.

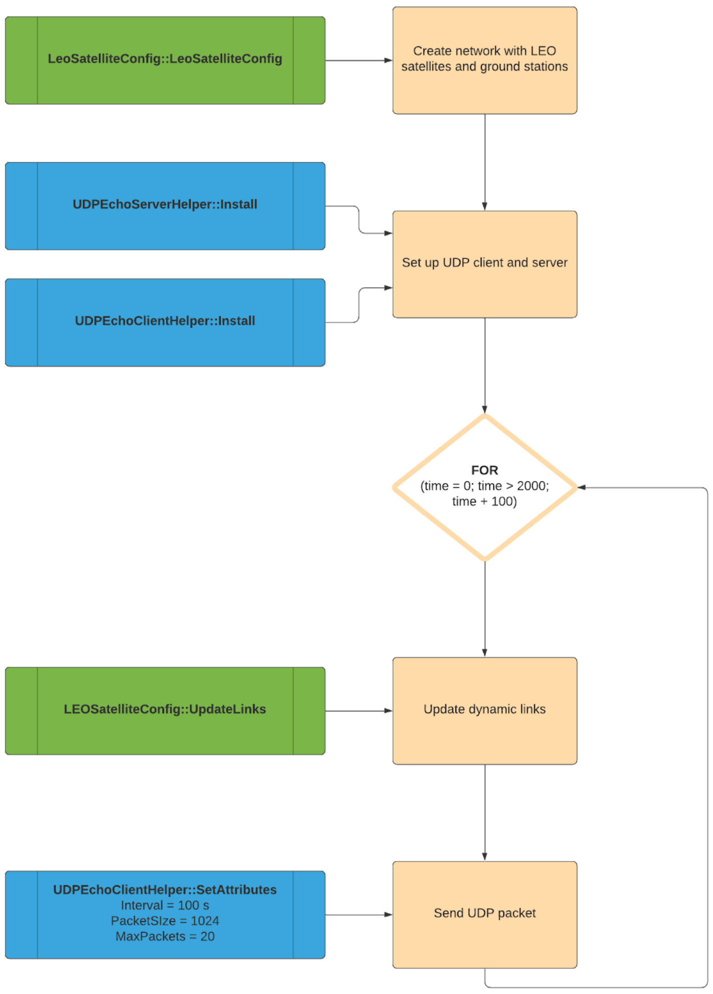
LeoSatelliteConfig::LeoSatelliteConfig
Create network with LEO satellites and ground stations
UDPEchoServerHelper::Install
Set up UDP client and server
UDPEchoClientHelper::Install
FOR (time = 0; time > 2000; time + 100)
LEOSatelliteConfig::UpdateLinks
Update dynamic links
UDPEchoClientHelper::SetAttributes Interval = 100 s PacketSIze = 1024 MaxPackets = 20
Send UDP packet

Figure 8: Simulation flow.

Three different experiments were conducted, and the results are displayed in the subsequent sections. In the results shown below, round-trip time (RTT) is calculated as the average time required for all packets to be sent from client to server and back during the full simulation. The number of hops is taken to be the average number of nodes traversed from client to server and back during the full simulation.

3.1 Number of hops versus Size of Satellite Network

The first experiment compares the average number of hops in a round-trip per size of the satellite network. Table 1 lists the data collected, and Fig. 9 displays the results in the form of a line graph organized according to the number of planes in the LEO satellite network.

Number of satellites per orbital plane	**Average Number of Hops**			
	5 planes	**7 planes**	**9 planes**	**11 planes**
4	9	11	11	13
8	11.3	13.2	13.5	15.5
12	13.4	15.3	15.6	17.6
16	15.4	17.3	17.7	19.7
20	17.4	19.3	19.8	21.8
24	19.5	21.4	21.7	23.8
28	21.5	23.4	23.7	25.8

Table 1: Data for average number of hops in a round-trip versus size of satellite network.

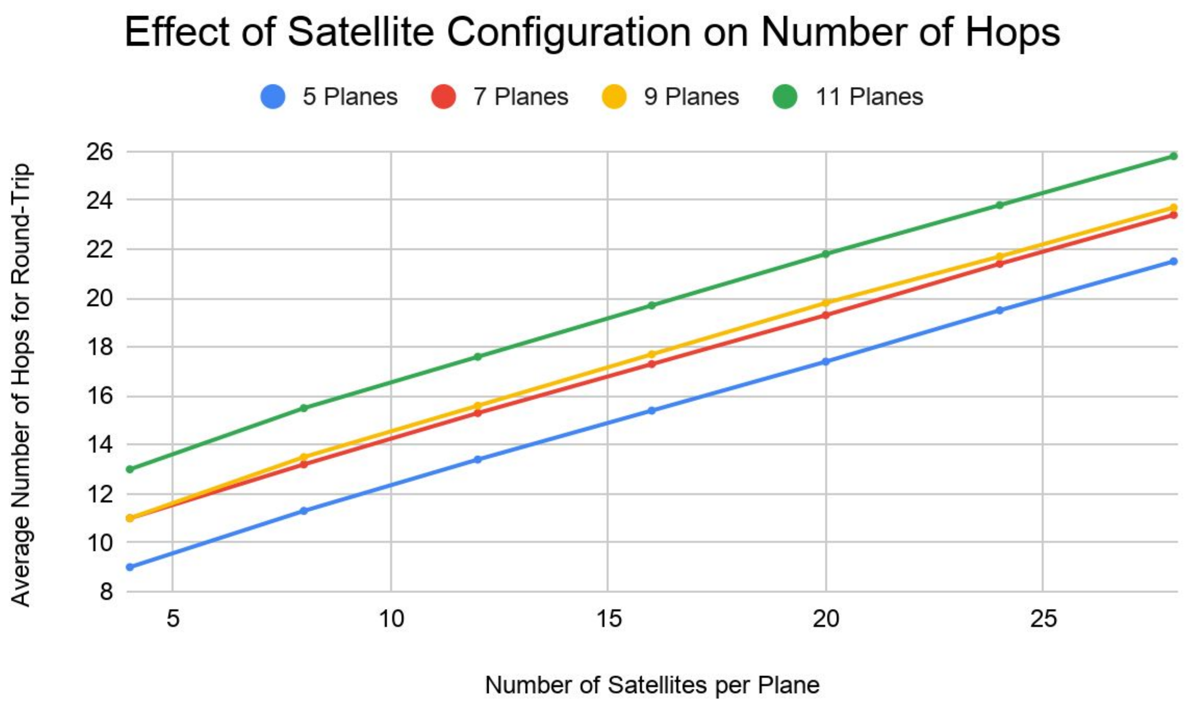

Figure 9: Average number of hops in a round-trip versus size of satellite network.

From Fig. 9, it can be observed that increasing the number of satellites per orbital plane leads to an increase in the number of hops required for a packet to be transmitted between ground stations. The trend can be seen to be nearly linear.

This trend stands to reason as increasing the number of satellites per orbital plane leads to a greater number of satellites between the ground stations. Since, in the LEO satellite network configuration presented here, each satellite is connected to exactly four other satellites (its directly-adjacent neighbors), the packet is forced to be transmitted between all of the satellites in its path — it cannot "skip" satellites between it and its destination as there are no links available for it to do so.

3.2 RTT versus Size of Satellite Network

The next experiment compares average RTT versus the size of the satellite network. Table 2 lists the data collected, and Fig. 10 displays the results of data collected in the form of a line graph organized according to the number of planes in the LEO satellite network.

Number of satellites per orbital plane	Average RTT (s)			
	5 planes	7 planes	9 planes	11 planes
4	0.3163557165	0.3735906093	0.3625927415	0.4123042336
8	0.2836814303	0.311222296	0.2963554839	0.3247936079
12	0.276367334	0.2895328663	0.2683191534	0.2904637724
16	0.290454808	0.2930258743	0.2600693072	0.2803170556
20	0.293654645	0.3042221518	0.2734021586	0.2949358806
24	0.2908444814	0.304402985	0.2667672368	0.2925206788
28	0.2969210555	0.3092615448	0.2692735063	0.2895764836

Table 2: Data for average RTT versus size of satellite network.

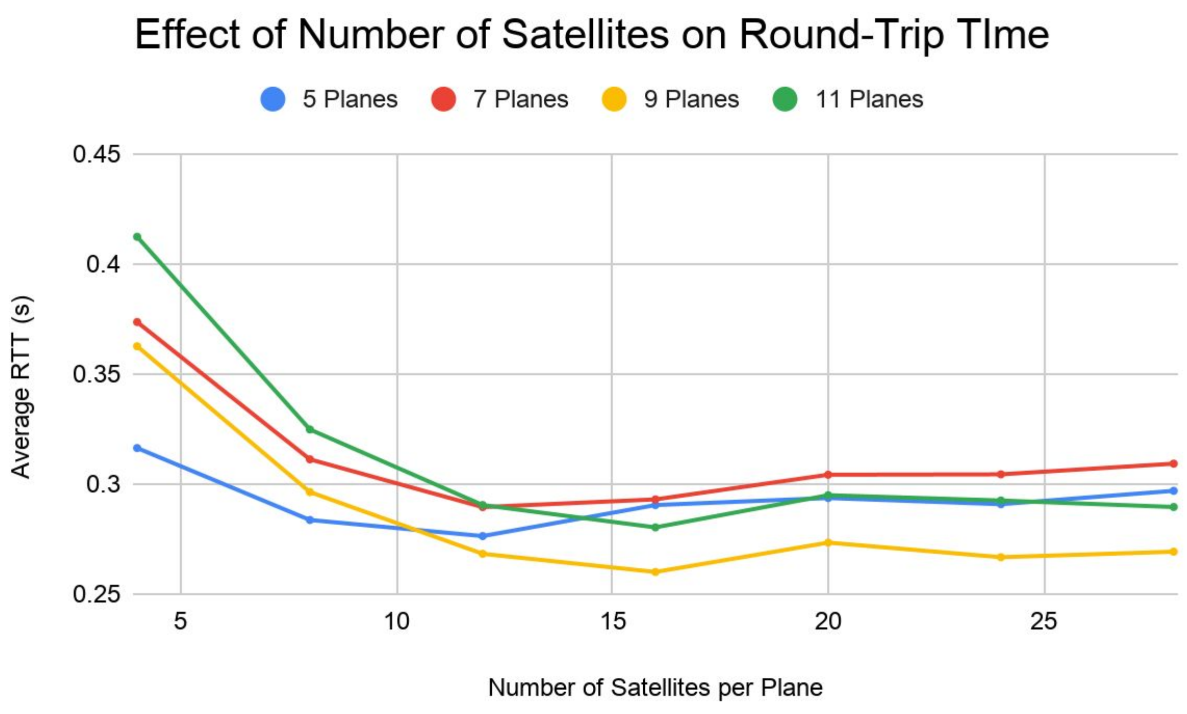

Figure 10: Average RTT versus size of satellite network.

From Fig. 10, it can be observed that, initially, increasing the number of satellites per orbital plane decreases the average RTT. This trend ceases after a certain number of satellites per orbital plane for each configuration consisting of a fixed number of orbital planes, and further increase in the number of satellites per orbital plane can be seen to slightly increase the average RTT.

Therefore, we can identify that there is an optimal number of satellites per orbital plane for each configuration consisting of a fixed number of orbital planes. The decrease in link delays due a greater number of satellites per plane are only beneficial in lowering RTT until this optimal number is reached.

The data in Fig. 9 has been collected on a simplified LEO satellite network simulation which does not account for the overhead involved in satellite transmission. When combined with the results seen in Fig. 8, which show that an increased number of satellites per plane lead to an increase number of hops from source to destination, we hypothesize that RTT will increase more dramatically after the optimal number of satellites has been reached for each configuration consisting of a fixed number of orbital planes. This increase would be due to the tradeoff involved in decreasing link delays by increasing the number of satellites in the network, and increasing total delay due to satellite transmission by increasing the number of hops required from source to destination.

3.3 RTT versus Altitude of Satellite Network

The final experiment compares average RTT versus the altitude of the satellite network. Table 3 lists the data collected, and Fig. 11 displays the results of the data collected in the form of a line graph.

Altitude (km)	Average RTT (s)
500	0.2327437758
600	0.2374440146
700	0.2469918745
800	0.2446166877
900	0.2449374887
1000	0.2560940156
1100	0.2657430958
1200	0.2583438511
1300	0.2633701737
1400	0.2659976394
1500	0.2694497151
1600	0.2726561003
1700	0.2770539662
1800	0.2863040995
1900	0.294162827
2000	0.2963554839

Table 3: Data for average RTT versus altitude of LEO satellite network.

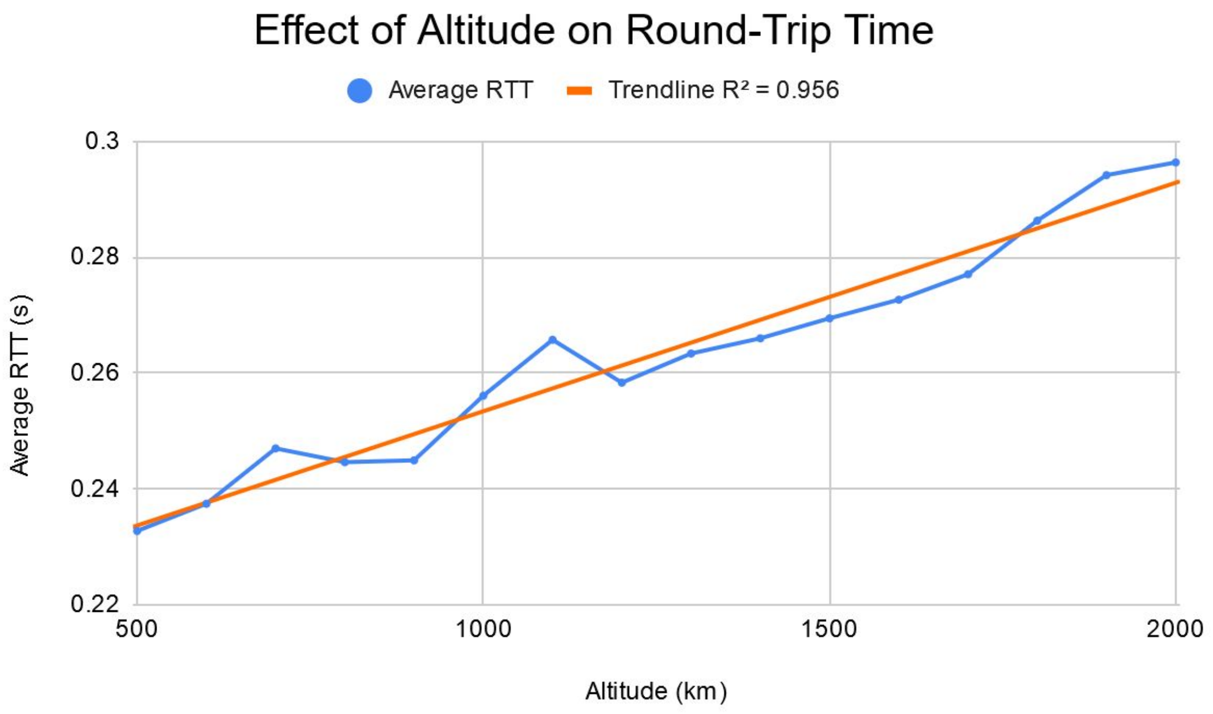

Figure 11: Average RTT versus altitude of LEO satellite network.

From Fig. 11, we can observe that increasing the altitude of the LEO satellite network results in an increase in average RTT. The trend is observed to be nearly-linear as the goodness-of-fit of linear regression for this data, R^2, is very close to 1.

In his Ph.D. thesis, Gvozdjak commented on a tradeoff involved in lower altitude satellite networks, which provide lower latencies for transmission between ground stations and satellites but also require more frequent hand-offs between ground stations [8]. As our LEO satellite network model implements changing dynamic links (performing hand-offs) as an almost-instantaneous process, the tradeoff is not observed. Only the benefit of a lower altitude satellite network is observed, as it shows a lower average RTT.

In a more robust model which takes into account any overhead that may be involved in modifying links between satellites, a tradeoff may be observed when this experiment is repeated. This tradeoff may become even more prominent in a model which accounts for overhead involved in modifying links between satellites while simulating a congested network.

3.4 Data Collection

As previously described in this section, simulations included setting up a LEO satellite constellation with a specific number of planes, satellites per plane, and altitude. Then UDP traffic was set to run through this network topology. To collect all of the relevant data illustrated and discussed above, we made use of the *ns-3* Flow Monitor module [19]. The FlowMonitorHelper allows flow monitoring to be enabled on all nodes within an *ns-3* network topology. This monitors all traffic sent and received on a per IP-address basis. Data collected using flow monitoring includes packet arrival times, delay, jitter, lost packets, and times forwarded. The FlowMonitorHelper then allows for this data to be serialized to an XML file at the end of a simulation, from which the information necessary to our analysis may be extracted.

Running simulations with a smaller number of satellites typically resulted in real processing times of approximately one to two minutes. As we increased the number of planes and satellites per plane, however, this processing time greatly increased. The largest simulation that we ran consisted of a LEO satellite constellation with a topology of 11 planes and 28 satellites per plane, thus a total number of satellites of 308 satellites. This simulation resulted in a processing time of 47 minutes. The vast majority of this processing time was determined to be accounted for by the population of routing tables for each satellite. As the number of satellites increase, the network becomes more complex and so do the routing tables. Due to this processing time, we were constrained in the amount of data that we were able to collect within a reasonable amount of time.

All simulation results were run on an Intel i7 quad-core processor.

4 Discussion

The simulation results presented above validate the LEO satellite constellation network we have created in *ns-3*, and also demonstrate fundamental properties of LEO satellite networks. In a network with a fixed number of connections between satellites, increasing the number of satellites in the network is demonstrated to increase the number of hops required for a packet to be transmitted from client to server and back. We have also determined that there is an optimal network size which results in the lowest average RTT while requiring the least number of satellites — increasing the number of satellites in the network beyond this optimal point does not yield significant benefits in average RTT. Finally, we observed that lower altitude LEO satellite networks experience lower average RTTs as the propagation distance between ground stations and satellites, and among satellites, is smaller.

There are multiple ways to implement a LEO satellite constellation that may differ from what we have presented in this paper. Alternative approaches to this solution involve a LEO satellite constellation with a different orbital configuration, for example an inclined orbit instead of a polar orbit. A different configuration of channels among satellites may be implemented, or alternative mathematical models for configuring and maintaining satellite positions in GPS coordinates may be explored.

Future work may also involve creating a more robust LEO satellite constellation than the one presented in this paper. For instance, satellite transmission overhead may be inserted into the model, which is predicted to show more accurate data for RTT versus the number of satellites in the network. More robust models may take into consideration the radio frequencies used for satellite communication, and account for any possible interference caused by neighboring satellites using the same radio frequencies. Support for collisions in polar orbits may be added into a model, for instance by disabling satellites as they move past poles. A model may be configured to place ground stations anywhere on the globe, instead of restricting the position to be directly underneath an orbital plane.

Future work may also involve expanding the scope of simulation experiments conducted. For example, a simulation with a greater number of ground stations may be conducted, or a congested network may be investigated.

5 Conclusion

As LEO satellite constellation networks emerge as technologies for communication and Internet connections, it is useful to be able to simulate these network topologies and analyze their properties. In this project, we designed and implemented a LEO satellite constellation network in *ns-3*, and then ran experiments on the implementation to evaluate properties of the network.

Construction of this model was a non-trivial task as it involved creating modules in *ns-3* that did not previously exist. To construct the model, additions to the existing *ns-3* source code were added in the form of two mobility models — a LEO Satellite Mobility Model and a Ground Station Mobility Model — and a top level LEO Satellite Config Module. When designing the algorithms behind the code, difficulties arose in implementing the LEO Satellite Mobility Model due to operation in GPS coordinates, and the organization and maintenance of all the dynamic components of the satellite network configuration were not trivial.

The experiments conducted on our LEO satellite constellation network functioned to both validate the LEO satellite constellation network created, as well as generated primary analysis on these networks. Much remains to be designed and explored to create a more robust LEO satellite network model in *ns-3* such as overhead involved with satellite transmission, consideration of the radio frequencies used by these satellites, and support for collisions.

Throughout this project, our group members learned a great deal. We gained knowledge regarding satellites and satellites networks, such as LEO satellite network configurations, the mathematics behind satellite motion, and factors involved in satellite communication. We also became familiar with the *ns-3* tool as we created our own module, created unique models extending from existing *ns-3* classes, and ran simulations and collected data.

References

[1] M. Sheetz and M. Petrova. “Why in the next decade companies will launch thousands more satellite than in all of history,” *CNBC*, para. 2, Dec. 15, 2019. [Online]. Available: https://www.cnbc.com/2019/12/14/spacex-oneweb-and-amazon-to-launch-thousands-more-satellites-in-2020s.html. [Accessed: Apr. 11, 2020].

[2] “StarLink Mission,” *SpaceX*, Mar., 2020. [Online]. Available: https://www.spacex.com/sites/spacex/files/sixth_starlink_mission_overview_0.pdf. [Accessed: Apr. 11, 2020].

[3] M. Sheetz, “Amazon wants to launch thousands of satellites so it can offer broadband internet from space,” *CNBC,* para.
2, Apr. 4, 2019. [Online]. Available: https://www.cnbc.com/2019/04/04/amazon-project-kuiper-broadband-internet-small-satellite-network.html. [Accessed: Apr. 11, 2020].

[4] J. O’Callaghan, “SpaceX Rival OneWeb Launches 34 Satellites In Space Internet Race With Starlink Mega Constellation, *Forbes*, para. 3, Feb. 6, 2020. [Online]. Available: https://www.forbes.com/sites/jonathanocallaghan/2020/02/06/spacex-rival-oneweb-launches-34-satellites-in-space-internet-race-with-starlink-mega-constellation/#498a76a33d81. [Accessed: Apr 11. 2020].

[5] “About Us,” *Iridium*, 2019. [Online]. Available: https://www.iridium.com/. [Accessing: Mar. 5, 2020].

[6] G. Ritchie, “Why Low-Earth Orbit Satellites Are the New Space Race,” *The Washington Post*, para. 3, Aug. 15, 2019. [Online]. Available: https://www.washingtonpost.com/business/why-low-earth-orbit-satellites-are-the-new-space-race/2019/08/15/6b224bd2-bf72-11e9-a8b0-7ed8a0d5dc5d_story.html. [Accessed: Mar. 5, 2020].

[7] S. J. Campanella and T. J. Kirkwood, “Faster than fibre: Advantages and challenges of LEO communication satellite systems,” *AIP Conference Proceedings,* vol. 325, no. 39, May 12, 2008. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.47249. [Accessed: Mar. 5, 2020].

[8] P. Gvozdjak, “Modelling Communications in Low-Earth Orbit Satellite Networks,” Ph.D. dissertation, School of Computing Science, Simon Fraser Univ., Burnaby, 2000. [Online]. Available: https://www.collectionscanada.gc.ca/obj/s4/f2/dsk1/tape3/PQDD_0011/NQ61647.pdf. [Accessed: Apr. 11, 2020].

[9] M. Handley, “Delay is Not an Option: Low Latency Routing in Space,” in *Proc. of the 17th ACM Workshop on Hot Topics in Networks, November, 2018, Washington, USA* [Online]. Available: ACM Digital Library, https://dl.acm.org/doi/10.1145/3286062.3286075. [Accessed: Apr. 11, 2020].

[10] E. Zeleny, “Orbital Speed and Period of a Satellite,” *Wolfram Demonstrations Project*, Mar.

2011. [Online]. Available: https://demonstrations.wolfram.com/OrbitalSpeedAndPeriodOfASatellite/. [Accessed: Apr. 11, 2020].

[11] M. Basyir, M. Nasir, S. Suryati, and W. Mellyssa, "Determination of Nearest Emergency Service Office Using Haversine Formula Based on Android Platform, *Emitter International Journal of Engineering Technology*, vol, 5, no. 2, pp. 270-278, Jan. 13, 2018. [Online]. Available: https://emitter.pens.ac.id/index.php/emitter/article/view/220. [Accessed: Apr. 11, 2020].

[12] "How to calculate distance from a given latitude and longitude on the earth to a specific geostationary satellite," *Stack Exchange Mathematics,* Aug. 28, 2019. [Online]. Available: https://math.stackexchange.com/questions/301410/how-to-calculate-distance-from-a-given-latitude-and-longitude-on-the-earth-to-a. [Accessed: Apr. 11, 2020].

[13] "Point-To-Point Network Device," *ns-3*. [Online]. Available: https://www.nsnam.org/doxygen/group__point-to-point.html. [Accessed: Apr. 11, 2020].

[14] "CSMA Network Device," *ns-3*. [Online]. Available: https://www.nsnam.org/doxygen/group__csma.html. [Accessed: Apr. 11, 2020].

[15] R.L. Smith-Rose, "The Speed of Radio Waves and Its Importance in Some Applications," *Proceedings of the IRE*, vol. 38, no. 1, pp. 16-20, Jan., 1950. [Online]. Available: IEEE Xplore, https://ieeexplore.ieee.org/document/1701081/authors#authors. [Accessed: Apr. 11, 2020].

[16] I. del Portillo, B. G. Cameron, and E. F. Crawley, "A technical comparison of three low earth orbit satellite constellation systems to provide global broadband," *Acta Astronautica*, vol. 159, pp. 123-135, June, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0094576518320368. [Accessed: Apr. 5, 2020].

[17] "Internet," *ns-3*. [Online]. Available:
https://www.nsnam.org/doxygen/group__internet.html.
[Accessed: Apr. 11, 2020].

[18] "UdpClientServer," *ns-3*. [Online]. Available: https://www.nsnam.org/doxygen/group__udpclientserver.html. [Accessed: Apr. 11, 2020].

[19] "Flow Monitor," *ns-3*. [Online]. Available: https://www.nsnam.org/docs/models/html/flow-monitor.html. [Accessed: Apr. 12, 2020].

Appendix A: Code Listing

leo-satellite-mobility.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Mobility model subclass
 * Keeps track of current position and velocity of LEO satellites
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */

#ifndef LEO_SATELLITE_MOBILITY_H
#define LEO_SATELLITE_MOBILITY_H

#include "ns3/object.h"
#include "ns3/ptr.h"
#include "ns3/mobility-model.h"
#include "ns3/vector.h"

namespace ns3 {

/**
 * \ingroup leo-satellite
 * \brief leo-satellite mobility model.
 *
 * Each satellite moves in a polar orbit within its plane
 * Satellites move with a fixed velocity determined by their altitude
 * Satellites in adjacent planes move in opposing directions
 */
class LeoSatelliteMobilityModel : public MobilityModel
{
public:
  /**
   * Register this type with the TypeId system.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);
  LeoSatelliteMobilityModel();

private:
  virtual Vector DoGetPosition (void) const;
  virtual void DoSetPosition (const Vector &position);
  virtual Vector DoGetVelocity (void) const;
  friend double CalculateDistance (const Vector &a, const Vector &b);
  uint32_t m_current; // current node
  double m_nPerPlane; // number of satellites per plane -> m_nPerPlane/2 must be even
number
  double m_numPlanes; // number of planes -> must be an odd number
  mutable double m_time; // time when current m_latitude, m_longitude, and m_direction
were set
  double m_altitude; // [km]
  // The following variables are calculated automatically given the above parameters
  mutable double m_latitude; // latitude of satellite at m_time
                   // negative value indicates southern latitude, positive value indicates
northern latitude
  mutable double m_longitude; // initial longitude of satellite
                   // negative value indicates western longitude, positive value indicates
eastern longitude
  mutable bool m_direction; // each adjacent plane will be orbiting in an opposite
direction
                   // 1 = S to N, 0 = N to S
  double m_speed; // [m/s]
};

} // namespace ns3

#endif /* LEO_SATELLITE_MOBILITY_H */
```

leo-satellite-mobility.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Mobility model subclass
 * Keeps track of current position and velocity of LEO satellites, calculates distance
between satellites
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */

#include "leo-satellite-mobility.h"
#include "ns3/vector.h"
#include "ns3/double.h"
#include "ns3/boolean.h"
#include "ns3/log.h"
#include "ns3/simulator.h"
#include "ns3/integer.h"
#define _USE_MATH_DEFINES
#include <cmath>
#include <algorithm>

namespace ns3 {
```

```
NS_LOG_COMPONENT_DEFINE ("LeoSatelliteMobility");

NS_OBJECT_ENSURE_REGISTERED (LeoSatelliteMobilityModel);

double earthRadius = 6378.1; // radius of Earth [km]
uint32_t currentNode = 0; // for initialization only

TypeId
LeoSatelliteMobilityModel::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::LeoSatelliteMobilityModel")
      .SetParent<MobilityModel> ()
      .SetGroupName ("Mobility")
      .AddConstructor<LeoSatelliteMobilityModel> ()
      .AddAttribute ("NPerPlane", "The number of satellites per orbital plane.",
                  IntegerValue (1),
                  MakeIntegerAccessor (&LeoSatelliteMobilityModel::m_nPerPlane),
                  MakeIntegerChecker<uint32_t> ())
      .AddAttribute ("NumberofPlanes", "The total number of orbital planes.",
                  IntegerValue (1),
                  MakeIntegerAccessor (&LeoSatelliteMobilityModel::m_numPlanes),
                  MakeIntegerChecker<uint32_t> ())
      .AddAttribute ("Latitude",
                  "Latitude of satellite.",
                  DoubleValue(1.0),
                  MakeDoubleAccessor (&LeoSatelliteMobilityModel::m_latitude),
                  MakeDoubleChecker<double> ())
      .AddAttribute ("Longitude",
                  "Longitude of satellite. Constant for satellites in same plane.",
                  DoubleValue(1.0),
                  MakeDoubleAccessor (&LeoSatelliteMobilityModel::m_longitude),
                  MakeDoubleChecker<double> ())
      .AddAttribute ("Time",
                  "Time when initial position of satellite is set.",
                  DoubleValue (1.0),
                  MakeDoubleAccessor (&LeoSatelliteMobilityModel::m_time),
                  MakeDoubleChecker<double> ())
      .AddAttribute ("Altitude",
                  "Altitude of satellite. Used to determine velocity.",
                  DoubleValue(1.0),
                  MakeDoubleAccessor (&LeoSatelliteMobilityModel::m_altitude),
                  MakeDoubleChecker<double> ())
      .AddAttribute ("Direction",
                  "Direction of satellite relative to other satellites.",
                  BooleanValue(1),
                   MakeBooleanAccessor (&LeoSatelliteMobilityModel::m_direction),
                  MakeBooleanChecker ())
  ;

  return tid;
}

LeoSatelliteMobilityModel::LeoSatelliteMobilityModel()
```

```
{
  currentNode++;
  m_current = currentNode;
}

/* To be called after MobilityModel object is created to set position.
   Input should be a NULL vector as position is determined by number of orbital planes and
number of satellites per
   orbital plane

   Node positions are set as follows:
   Node 1 is the closest to latitude = 90, longitude = -180
   Nodes in the same plane are set by decrementing latitude until -90 is reached,
longitude kept the same
   The first node in the next plane is set by setting latitude = 90 and incrementing the
longitude
   ...
   Node N is closest to latitude = -90, longitude = 180

   The latitude edges of +90 and -90 are populated
   The longitude edge of -180 is populated (180 is not populated as it is equivalent to
-180)
 */
void
LeoSatelliteMobilityModel::DoSetPosition (const Vector &position)
{
  // Determine speed of satellite from altitude
  double G = 6.673e-11; // gravitational constant [Nm^2/kg^2]
  double earthMass = 5.972e24; // mass of Earth [kg]
  double altitude = m_altitude;

  m_speed = std::sqrt(G*earthMass/(earthRadius*1000 + altitude*1000));

  // Set latitude and longitude of satellite from number of orbital planes and number of
satellites per orbital plane
  // First satellite in plane will have a longitude that is a half-step down from 90
degrees
  if (m_current == 1)
  {
      m_latitude = 90 - 180/(m_nPerPlane/2)/2;
  }
  else if ((fmod(m_current - 1, m_nPerPlane/2) == 0) && (m_current > m_nPerPlane))
  {
      m_latitude = 90 - 180/(m_nPerPlane/2)/2;
  }
  else
  {
      m_latitude = 90 - 180/(m_nPerPlane/2)/2 - 180/(m_nPerPlane/2)*fmod(m_current - 1,
m_nPerPlane/2);
  }
  m_longitude = -180 + 360/(m_numPlanes*2)*floor((m_current - 1)/(m_nPerPlane/2));

  // Set direction based on which orbital plane satellite belongs to
  uint32_t plane = floor((m_current - 1)/(m_nPerPlane/2));
  (plane % 2 == 1) ? m_direction = 0: m_direction = 1;
}

Vector
LeoSatelliteMobilityModel::DoGetPosition (void) const
{
   double altitude = m_altitude;
   double latitude = m_latitude;
   double longitude = m_longitude;
   bool direction = m_direction;
   double currentTime = Simulator::Now().GetSeconds();
   double radius = earthRadius + altitude;
   double newLatitude;

   // How many orbital periods have been completed, then converted to degree displacement
   double orbitalPeriod = 2*M_PI*radius/(m_speed/1000); // [seconds]
   double orbitalPeriodTravelled = (currentTime - m_time)/orbitalPeriod;
   double degreeDisplacement = fmod(orbitalPeriodTravelled*360, 360);

   if (direction == 1)
   {
      if ((latitude + degreeDisplacement) > 90)
      {
      degreeDisplacement = degreeDisplacement - (90 - latitude); // We've accounted for
the degrees taken to get to north pole
      latitude = 90;
      if (longitude < 0)
            longitude = longitude + 180;
      else
            longitude = longitude - 180;
      direction = 0;

      if ((latitude - degreeDisplacement) < -90)
      {
      degreeDisplacement = degreeDisplacement - 180; // We've accounted for the degrees
taken to get to south pole
      latitude = -90;
      if (longitude < 0)
             longitude = longitude + 180;
      else
            longitude = longitude - 180;
      direction = 1;
      }
      }
      (direction == 1) ? newLatitude = latitude + degreeDisplacement : newLatitude =
latitude - degreeDisplacement;
   }

   else if (direction == 0)
   {
      if ((latitude - degreeDisplacement) < -90)
      {
```

```
      degreeDisplacement = degreeDisplacement - (latitude - (-90)); // We've accounted for
the degrees taken to get to south pole
      latitude = -90;
      if (longitude < 0)
            longitude = longitude + 180;
      else
            longitude = longitude - 180;
      direction = 1;

      if ((latitude + degreeDisplacement) > 90)
      {
      degreeDisplacement = degreeDisplacement - 180; // We've accounted for the degrees
taken to get to north pole
      latitude = 90;
      if (longitude < 0)
            longitude = longitude + 180;
      else
            longitude = longitude - 180;
      direction = 0;
      }
      }
      (direction == 1) ? newLatitude = latitude + degreeDisplacement : newLatitude =
latitude - degreeDisplacement;
   }

   // Update latitude, longitude, direction, and time values for this object
   m_latitude = newLatitude;
   m_longitude = longitude;
   m_time = currentTime;
   m_direction = direction;

   // Create vector to return
   Vector currentPosition = Vector(m_latitude, m_longitude, altitude);
   return currentPosition;
}

/* Args "a" and "b" to be obtained from LeoSatelliteMobilityModel::DoGetPosition for each
argument
   Distance calculated using Haversine formula for distance of two points on spherical
surface
   Ignoring the slight ellipsoidal effects of Earth */
double
CalculateDistance (const Vector &a, const Vector &b)
{
  double altitude = a.z;
  double radius = earthRadius + altitude;
  //a.x = latitude1
  //a.y = longitude1
  //b.x = latitude2
  //b.x = latitude2
  // Convert to radians for use in Haversine formula
  double latitude1 = a.x*M_PI/180;
  double latitude2 = b.x*M_PI/180;
  double deltaLatitude = (b.x - a.x)*M_PI/180;
  double deltaLongitude;
  if (b.y == -180 && a.y > 0)
  {
      deltaLongitude = (180 - a.y)*M_PI/180;
  }
  else if (a.y == -180 && b.y > 0)
  {
      deltaLongitude = (b.y - 180)*M_PI/180;
  }
  else
  {
      deltaLongitude = (b.y - a.y)*M_PI/180;
  }

  // Haversine formula
  double y = pow(sin(deltaLatitude/2), 2) +
cos(latitude1)*cos(latitude2)*pow(sin(deltaLongitude/2), 2);
  double z = 2*atan(sqrt(y)/sqrt(1-y));
  double distance = radius*z;

  return distance;
}

Vector
LeoSatelliteMobilityModel::DoGetVelocity (void) const
{
  Vector null = Vector(0.0, 0.0, 0.0);
  return null;
}

} // namespace ns3
```

ground-station-mobility.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Mobility model subclass
 * Keeps track of current position of ground stations and distance from satellites
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */
#ifndef GROUND_STATION_MOBILITY_H
#define GROUND_STATION_MOBILITY_H

#include "ns3/object.h"
#include "ns3/mobility-model.h"
#include "ns3/vector.h"
```

```
namespace ns3 {

/**
 * \ingroup leo-satellite
 * \brief ground station mobility model.
 *
 * For a simplified simulation, ground stations will be placed along the
 * longitude of satellites orbiting above and at varying latitudes
 * Currently supports two ground stations
 */
class GroundStationMobilityModel : public MobilityModel
{
public:
  /**
   * Register this type with the TypeId system.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);
  GroundStationMobilityModel();

private:
  virtual Vector DoGetPosition (void) const;
  virtual void DoSetPosition (const Vector &position);
  virtual Vector DoGetVelocity (void) const;
  friend double CalculateDistanceGroundToSat (const Vector &a, const Vector &b); //
Vectors must correspond to a ground station and a LEO satellite
  double m_nPerPlane; // number of satellites per plane
  double m_numPlanes; // number of planes
  // The following variables are calculated automatically given the above parameteres
  double m_latitude; // latitude of ground station
                    // negative value indicates southern latitude, positive value indicates
northern latitude
  double m_longitude; // longitude of ground station
                    // negative value indicates western longitude, positive value indicates
eastern longitude
};

} // namespace ns3

#endif /* GROUND_STATION_MOBILITY_H */
```

ground-station-mobility.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * Mobility model subclass
 * Keeps track of current position of ground stations and distance from satellites
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */

#include "ground-station-mobility.h"
#include "ns3/vector.h"
#include "ns3/double.h"
#include "ns3/log.h"
#include "ns3/integer.h"
#define _USE_MATH_DEFINES
#include <cmath>

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("GroundStationMobility");

NS_OBJECT_ENSURE_REGISTERED (GroundStationMobilityModel);

uint32_t current = 0; // to know if we are setting up first or second ground station

TypeId
GroundStationMobilityModel::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::GroundStationMobilityModel")
      .SetParent<MobilityModel> ()
      .SetGroupName ("Mobility")
      .AddConstructor<GroundStationMobilityModel> ()
      .AddAttribute ("NPerPlane", "The number of satellites per orbital plane.",
                     IntegerValue (1),
                     MakeIntegerAccessor (&GroundStationMobilityModel::m_nPerPlane),
                     MakeIntegerChecker<uint32_t> ())
      .AddAttribute ("NumberofPlanes", "The total number of orbital planes.",
                     IntegerValue (1),
                     MakeIntegerAccessor (&GroundStationMobilityModel::m_numPlanes),
                     MakeIntegerChecker<uint32_t> ())
      .AddAttribute ("Latitude",
                     "Latitude of ground station.",
                     DoubleValue(1.0),
                     MakeDoubleAccessor (&GroundStationMobilityModel::m_latitude),
                     MakeDoubleChecker<double> ())
      .AddAttribute ("Longitude",
                     "Longitude of ground station.",
                     DoubleValue(1.0),
                     MakeDoubleAccessor (&GroundStationMobilityModel::m_longitude),
                     MakeDoubleChecker<double> ())
  ;

  return tid;
}

GroundStationMobilityModel::GroundStationMobilityModel()
{
}
```

```
/* To be called after MobilityModel object is created to set position.
   Input should be a NULL vector as position is determined by number of orbital planes
and number of satellites per
   orbital plane
   Both ground stations are set along the longitude of a satellite's orbit (not at the
same longitude), and at random
   latitudes
 */
void
GroundStationMobilityModel::DoSetPosition (const Vector &position)
{
  current++;
  if (current == 1) // first ground station
  {
      m_latitude = 90 - 180/(m_nPerPlane/2)/2 ;
      m_longitude = -180;
  }
  else // second ground station
  {
      m_latitude = 90 - 180/(m_nPerPlane/2)/2 - 180/(m_nPerPlane/2)*(m_nPerPlane/4);
      m_longitude = -180 + 360/(m_numPlanes*2)*floor(3*m_numPlanes/7);
  }
}

Vector
GroundStationMobilityModel::DoGetPosition (void) const
{
  Vector currentPosition = Vector(m_latitude, m_longitude, 0);
  return currentPosition;
}

/* Vector a is the position of the ground station
   Vector b is the position of the LEO satellite */
double
CalculateDistanceGroundToSat (const Vector &a, const Vector &b)
{
  double earthRadius = 6378.1; // radius of Earth [km]
  double distance;
  double deltaLatitude = (b.x - a.x)*M_PI/180;
  double deltaLongitude;
  if((b.y == -180 && a.y == -180) || (b.y == 0 && a.y == 0))
  {
      deltaLongitude = abs(a.y - b.y)*M_PI/180;
  }
  else if (b.y == -180)
  {
      deltaLongitude = std::min(std::abs(b.y - a.y), std::abs(0-a.y))*M_PI/180;
  }
  else if (a.y == -180)
  {
      deltaLongitude = std::min(std::abs(b.y - a.y), std::abs(b.y - 0))*M_PI/180;
  }
  else if (b.y == 0)
  {
      deltaLongitude = std::min(std::abs(b.y - a.y), std::abs(180 - a.y))*M_PI/180;
  }
  else if (a.y == 0)
  {
      deltaLongitude = std::min(std::abs(b.y - a.y), std::abs(b.y - 180))*M_PI/180;
  }
  else
  {
      deltaLongitude = abs(a.y - b.y)*M_PI/180;
  }

  distance = pow(earthRadius + b.z - earthRadius*cos(deltaLongitude)*cos(deltaLatitude),
2) + pow(earthRadius*sin(deltaLongitude)*cos(deltaLatitude), 2) +
pow(earthRadius*sin(deltaLatitude), 2);
  distance = sqrt(distance);

  return distance;
}

Vector
GroundStationMobilityModel::DoGetVelocity (void) const
{
  Vector null = Vector(0.0, 0.0, 0.0);
  return null;
}

} // namespace ns3
```

leo-satellite-config.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */

/*

 * LEO Satellite Constellation Config

 * Creates and maintains all satellites and links within a satellite communication
network
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */
#ifndef LEO_SATELLITE_CONFIG_H
#define LEO_SATELLITE_CONFIG_H

#include "ns3/vector.h"
```

```
#include "ns3/object.h"
#include "ns3/ptr.h"
#include "ns3/core-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/leo-satellite-mobility.h"
#include "ns3/ground-station-mobility.h"
#include <vector>
#include "ns3/mobility-module.h"
#include "ns3/csma-module.h"
#include <cmath>
#include "ns3/internet-module.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/applications-module.h"

namespace ns3 {

class LeoSatelliteConfig : public Object
{
public:
  /**
   * Register this type with the TypeId system.
   * \return the object TypeId
   */
  static TypeId GetTypeId (void);

  LeoSatelliteConfig (uint32_t num_planes, uint32_t num_satellites_per_plane, double
altitude);

  virtual ~LeoSatelliteConfig ();
  virtual TypeId GetInstanceTypeId (void) const;

  void UpdateLinks (); //update the intersatellite links

  NodeContainer ground_stations; //node container to hold ground stations
  std::vector<Ipv4InterfaceContainer> ground_station_interfaces;

private:
  uint32_t num_planes;
  uint32_t num_satellites_per_plane;
  double m_altitude;

  std::vector<NodeContainer> plane; //node container for each plane
  std::vector<NetDeviceContainer> intra_plane_devices; //contains net devices for all
P2P links for all planes
  std::vector<NetDeviceContainer> inter_plane_devices;
  std::vector<Ptr<CsmaChannel>> inter_plane_channels;
  std::vector<uint32_t> inter_plane_channel_tracker; //this will have the node from the
adjacent plane that is currently connected
  std::vector<NetDeviceContainer> ground_station_devices;
  std::vector<Ptr<CsmaChannel>> ground_station_channels;
  std::vector<uint32_t> ground_station_channel_tracker;
  std::vector<Ipv4InterfaceContainer> intra_plane_interfaces;
  std::vector<Ipv4InterfaceContainer> inter_plane_interfaces;
};

}

#endif /* LEO_SATELLITE_CONFIG_H */
```

leo-satellite-config.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * LEO Satellite Constellation Config
 * Creates and maintains all satellites and links within a satellite communication
network
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */

#include "leo-satellite-config.h"

namespace ns3 {

NS_OBJECT_ENSURE_REGISTERED (LeoSatelliteConfig);
NS_LOG_COMPONENT_DEFINE ("LeoSatelliteConfig");

extern double CalculateDistanceGroundToSat (const Vector &a, const Vector &b);

double speed_of_light = 299792458; //in m/s

//typeid
TypeId LeoSatelliteConfig::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::LeoSatelliteConfig")
  .SetParent<Object> ()
  .SetGroupName("LeoSatellite")
  ;
  return tid;
}

LeoSatelliteConfig::~LeoSatelliteConfig ()
{
}

TypeId LeoSatelliteConfig::GetInstanceTypeId (void) const
{
  TypeId tid = this->GetTypeId();
  return tid;
}
```

```
//constructor
LeoSatelliteConfig::LeoSatelliteConfig (uint32_t num_planes, uint32_t
num_satellites_per_plane, double altitude)
{
  this->num_planes = num_planes;
  this->num_satellites_per_plane = num_satellites_per_plane;
  this->m_altitude = altitude;

  uint32_t total_num_satellites = num_planes*num_satellites_per_plane;
  NodeContainer temp;
  temp.Create(total_num_satellites);

  //assign mobility model to all satellites
  MobilityHelper mobility;
  mobility.SetMobilityModel ("ns3::LeoSatelliteMobilityModel",
                             "NPerPlane", IntegerValue (num_satellites_per_plane),
                             "NumberofPlanes", IntegerValue (num_planes),
                             "Altitude", DoubleValue(altitude),
                             "Time", DoubleValue(Simulator::Now().GetSeconds()));
  mobility.Install(temp);

  for (NodeContainer::Iterator j = temp.Begin ();
       j != temp.End (); ++j)
    {
      Ptr<Node> object = *j;
      Ptr<MobilityModel> position = object->GetObject<MobilityModel> ();
      Vector null = Vector(0.0, 0.0, 0.0);
      position->SetPosition(null); // needs to be done to initialize
      NS_ASSERT (position != 0);
    }

  //assigning nodes to e/ plane's node container as necessary
  for (uint32_t i=0; i<num_planes; i++)
  {
     NodeContainer temp_plane;
     for(uint32_t j=0; j<num_satellites_per_plane/2; j++)
     {
       Vector pos = temp.Get(i*num_satellites_per_plane/2 +
j)->GetObject<MobilityModel> ()->GetPosition();
       std::cout << Simulator::Now().GetSeconds() << ": plane # "<< i << " node # "
<<j<< ": x = " << pos.x << ", y = " << pos.y << ", z = " << pos.z << std::endl;
       temp_plane.Add(temp.Get(i*num_satellites_per_plane/2 + j));
     }
     for(uint32_t j=num_satellites_per_plane/2; j> 0; j--)
     {
       Vector pos = temp.Get(total_num_satellites/2 + i*num_satellites_per_plane/2 + j
- 1)->GetObject<MobilityModel> ()->GetPosition();
       std::cout << Simulator::Now().GetSeconds() << ": plane # "<< i << " node # "
<<num_satellites_per_plane - j<< ": x = " << pos.x << ", y = " << pos.y << ", z = " <<
pos.z << std::endl;
       temp_plane.Add(temp.Get(total_num_satellites/2 + i*num_satellites_per_plane/2 +
j - 1));
```

```
      }
      InternetStackHelper stack;
      stack.Install(temp_plane);
      this->plane.push_back(temp_plane);
   }

   //setting up all intraplane links
   Vector nodeAPosition =
this->plane[0].Get(0)->GetObject<MobilityModel>()->GetPosition();
   Vector nodeBPosition =
this->plane[0].Get(1)->GetObject<MobilityModel>()->GetPosition();
   double distance = CalculateDistance(nodeAPosition, nodeBPosition);
   double delay = (distance * 1000)/speed_of_light; //should get delay in seconds
   PointToPointHelper intraplane_link_helper;
   intraplane_link_helper.SetDeviceAttribute ("DataRate", StringValue ("5.36Gbps"));
   intraplane_link_helper.SetChannelAttribute ("Delay", TimeValue(Seconds (delay)));

   std::cout<<"Setting up intra-plane links with distance of "<<distance<<" km and delay
of "<<delay<<" seconds."<<std::endl;

   for (uint32_t i=0; i<num_planes; i++)
   {
     for (uint32_t j=0; j<num_satellites_per_plane; j++)
     {

this->intra_plane_devices.push_back(intraplane_link_helper.Install(plane[i].Get(j),
plane[i].Get((j+1)%num_satellites_per_plane)));
       std::cout<<"Plane "<<i<<": channel between node "<<j<<" and node
"<<(j+1)%num_satellites_per_plane<<std::endl;
     }
   }

   //setting up interplane links
   std::cout<<"Setting up inter-plane links"<<std::endl;
   for (uint32_t i=0; i<num_planes; i++)
   {
     for (uint32_t j=0; j<num_satellites_per_plane; j++)
     {
       uint32_t nodeBIndex;
       (i == num_planes - 1) ? nodeBIndex = num_satellites_per_plane - j - 1: nodeBIndex
= j;
       Vector nodeAPos =
this->plane[i].Get(j)->GetObject<MobilityModel>()->GetPosition();
       Vector nodeBPos =
this->plane[(i+1)%num_planes].Get(nodeBIndex)->GetObject<MobilityModel>()->GetPosition(
);
       double distance = CalculateDistance(nodeAPos, nodeBPos);
       double delay = (distance*1000)/speed_of_light;
       CsmaHelper interplane_link_helper;
       interplane_link_helper.SetChannelAttribute("DataRate", StringValue ("5.36Gbps"));
       interplane_link_helper.SetChannelAttribute("Delay", TimeValue(Seconds(delay)));

       std::cout<<"Channel open between plane "<<i<<" satellite "<<j<<" and plane
```

```
"<<(i+1)%num_planes<<" satellite "<<nodeBIndex<< " with distance "<<distance<< "km and
delay of "<<delay<<" seconds"<<std::endl;

      NodeContainer temp_node_container;
      temp_node_container.Add(this->plane[i].Get(j));
      temp_node_container.Add(this->plane[(i+1)%num_planes]);
      NetDeviceContainer temp_netdevice_container;
      temp_netdevice_container = interplane_link_helper.Install(temp_node_container);
      Ptr<CsmaChannel> csma_channel;
      Ptr<Channel> channel;
      channel = temp_netdevice_container.Get(0)->GetChannel();
      csma_channel = channel->GetObject<CsmaChannel> ();

      for (uint32_t k=0; k<num_satellites_per_plane; k++)
      {
        if (j != k)
        {
csma_channel->Detach(temp_netdevice_container.Get(k+1)->GetObject<CsmaNetDevice> ());
        }
      }

      this->inter_plane_devices.push_back(temp_netdevice_container);
      this->inter_plane_channels.push_back(csma_channel);
      this->inter_plane_channel_tracker.push_back(nodeBIndex);
    }
  }

  //setting up two ground stations for now
  std::cout << "Setting up two ground stations" << std::endl;
  ground_stations.Create(2);
  //assign mobility model to ground stations
  MobilityHelper groundMobility;
  groundMobility.SetMobilityModel ("ns3::GroundStationMobilityModel",
                                   "NPerPlane", IntegerValue (num_satellites_per_plane),
                                   "NumberofPlanes", IntegerValue (num_planes));
  groundMobility.Install(ground_stations);
  //Install IP stack
  InternetStackHelper stack;
  stack.Install(ground_stations);
  for (int j = 0; j<2; j++)
  {
    Vector temp = ground_stations.Get(j)->GetObject<MobilityModel> ()->GetPosition();
    std::cout << Simulator::Now().GetSeconds() << ": ground station # " << j << ": x =
" << temp.x << ", y = " << temp.y << std::endl;
  }
  //setting up links between ground stations and their closest satellites
  std::cout<<"Setting links between ground stations and satellites"<<std::endl;
  for (uint32_t i=0; i<2; i++)
  {
    Vector gndPos = ground_stations.Get(i)->GetObject<MobilityModel> ()->GetPosition();
    uint32_t closestAdjSat = 0;
    uint32_t closestAdjSatDist = 0;
    uint32_t planeIndex;
    if (i == 0)
      planeIndex = 0;
    else
      planeIndex = floor(3*num_planes/7);
    //find closest adjacent satellite for ground station
    for (uint32_t j=0; j<this->num_satellites_per_plane; j++)
    {
      Vector pos =
this->plane[planeIndex].Get(j)->GetObject<MobilityModel>()->GetPosition();
      double temp_dist = CalculateDistanceGroundToSat(gndPos,pos);
      if((temp_dist < closestAdjSatDist) || (j==0))
      {
        closestAdjSatDist = temp_dist;
        closestAdjSat = j;
      }
    }
    double delay = (closestAdjSatDist*1000)/speed_of_light;
    CsmaHelper ground_station_link_helper;
    ground_station_link_helper.SetChannelAttribute("DataRate", StringValue
("5.36Gbps"));
    ground_station_link_helper.SetChannelAttribute("Delay", TimeValue(Seconds(delay)));

    std::cout<<"Channel open between ground station " << i << " and plane " <<
planeIndex << " satellite "<<closestAdjSat<<" with distance "<<closestAdjSatDist<< "km
and delay of "<<delay<<" seconds"<<std::endl;

    NodeContainer temp_node_container;
    temp_node_container.Add(ground_stations.Get(i));
    temp_node_container.Add(this->plane[planeIndex]);
    NetDeviceContainer temp_netdevice_container;
    temp_netdevice_container = ground_station_link_helper.Install(temp_node_container);
    Ptr<CsmaChannel> csma_channel;
    Ptr<Channel> channel;
    channel = temp_netdevice_container.Get(0)->GetChannel();
    csma_channel = channel->GetObject<CsmaChannel> ();

    for (uint32_t k=0; k<num_satellites_per_plane; k++)
    {
      if (closestAdjSat != k)
      {

csma_channel->Detach(temp_netdevice_container.Get(k+1)->GetObject<CsmaNetDevice> ());
      }
    }

    this->ground_station_devices.push_back(temp_netdevice_container);
    this->ground_station_channels.push_back(csma_channel);
    this->ground_station_channel_tracker.push_back(closestAdjSat);
  }

  //Configure IP Addresses for all NetDevices
  Ipv4AddressHelper address;
  address.SetBase ("10.1.0.0", "255.255.255.0");

  //configuring IP Addresses for IntraPlane devices
  for(uint32_t i=0; i< this->intra_plane_devices.size(); i++)
  {
    address.NewNetwork();

this->intra_plane_interfaces.push_back(address.Assign(this->intra_plane_devices[i]));
  }

  //configuring IP Addresses for InterPlane devices
  for(uint32_t i=0; i< this->inter_plane_devices.size(); i++)
  {
    address.NewNetwork();

this->inter_plane_interfaces.push_back(address.Assign(this->inter_plane_devices[i]));
    for(uint32_t j=1; j<= this->num_satellites_per_plane; j++)
    {
      if(j != this->inter_plane_channel_tracker[i] + 1)
      {
        std::pair< Ptr< Ipv4 >, uint32_t > interface =
this->inter_plane_interfaces[i].Get(j);
        interface.first->SetDown(interface.second);
      }
    }
  }

  //configuring IP Addresses for Ground devices
  for(uint32_t i=0; i< this->ground_station_devices.size(); i++)
  {
    address.NewNetwork();

this->ground_station_interfaces.push_back(address.Assign(this->ground_station_devices[i
]));
    for(uint32_t j=1; j<= this->num_satellites_per_plane; j++)
    {
      if(j != this->ground_station_channel_tracker[i] + 1)
      {
        std::pair< Ptr< Ipv4 >, uint32_t > interface =
this->ground_station_interfaces[i].Get(j);
        interface.first->SetDown(interface.second);
      }
    }
  }

  //Populate Routing Tables
  std::cout<<"Populating Routing Tables"<<std::endl;
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
  std::cout<<"Finished Populating Routing Tables"<<std::endl;

  // Set up packet sniffing for entire network
  /*CsmaHelper csma;
  for (uint32_t i=0; i< this->inter_plane_devices.size(); i++)
  {
    csma.EnablePcap("inter-sniff", this->inter_plane_devices[i].Get(1), true);
  }
  PointToPointHelper p2p;
  for(uint32_t i=0; i< this->intra_plane_devices.size(); i++)
  {
    p2p.EnablePcap("intra-sniff", this->intra_plane_devices[i].Get(1), true);
  }
  for(uint32_t i=0; i< this->ground_station_devices.size(); i++)
  {
    p2p.EnablePcap("ground-sniff", this->ground_station_devices[i].Get(1), true);
  }*/
}

void LeoSatelliteConfig::UpdateLinks()
{
  std::cout<<std::endl<<std::endl<<std::endl<<"Updating Links"<<std::endl;

  std::vector<NodeContainer> update_links_plane = this->plane;
  NodeContainer final_plane;
  for(uint32_t j=0; j<num_satellites_per_plane; j++)
  {
    final_plane.Add(this->plane[0].Get(num_satellites_per_plane - j - 1));
  }
  update_links_plane.push_back(final_plane);

  for (uint32_t i=0; i<this->num_planes; i++)
  {
    Vector refSatPos;
    uint32_t refSat = 0;
    //find reference satellite (closest to equator)
    for (uint32_t j=0; j<this->num_satellites_per_plane; j++)
    {
      Vector pos =
update_links_plane[i].Get(j)->GetObject<MobilityModel>()->GetPosition();
      if ((std::abs(pos.x) < std::abs(refSatPos.x)) || j == 0)
      {
        refSatPos = pos;
        refSat = j;
      }
    }

    //find the closest adjacent satellite to the reference satellite
    uint32_t closestAdjSat = 0;
    double closestAdjSatDist = 0;
    Vector adjPos; //debug
    for (uint32_t j=0; j<this->num_satellites_per_plane; j++)
    {
      Vector pos =
update_links_plane[i+1].Get(j)->GetObject<MobilityModel>()->GetPosition();
      double temp_dist = CalculateDistance(refSatPos,pos);
      if((temp_dist < closestAdjSatDist) || (j==0))
      {
        closestAdjSatDist = temp_dist;
        closestAdjSat = j;
        adjPos = pos; //debug
      }
    }

    //calculate the reference increment factor for adjacent satellites in a plane
    uint32_t ref_incr;
    (refSat <= closestAdjSat) ? (ref_incr = closestAdjSat - refSat) : (ref_incr =
this->num_satellites_per_plane - refSat + closestAdjSat);

    //update all adjacent satellites for this plane
    for (uint32_t j=0; j<this->num_satellites_per_plane; j++)
    {
      uint32_t access_idx = i*(this->num_satellites_per_plane) + j;
      uint32_t currAdjNodeID = this->inter_plane_channel_tracker[access_idx];
      uint32_t nextAdjNodeID = (j + ref_incr)%(this->num_satellites_per_plane);
      double nextAdjNodeDist;

      Vector constNodePos =
update_links_plane[i].Get(j)->GetObject<MobilityModel>()->GetPosition();
      Vector nextAdjNodePos =
update_links_plane[(i+1)/*%(this->num_planes)*/].Get(nextAdjNodeID)->GetObject<Mobility
Model>()->GetPosition();

      nextAdjNodeDist = CalculateDistance(constNodePos, nextAdjNodePos);

      if (i == this->num_planes - 1)
        nextAdjNodeID = num_satellites_per_plane - nextAdjNodeID - 1;

      if(currAdjNodeID == nextAdjNodeID)
      {
        double new_delay = (nextAdjNodeDist*1000)/speed_of_light;
        this->inter_plane_channels[access_idx]->SetAttribute("Delay",
TimeValue(Seconds(new_delay)));
        std::cout<<"Channel updated between plane "<<i<<" satellite "<<j<<" and plane
"<<(i+1)%num_planes<<" satellite "<<nextAdjNodeID<< " with distance
"<<nextAdjNodeDist<< "km and delay of "<<new_delay<<" seconds"<<std::endl;
      }
      else
      {

this->inter_plane_channels[access_idx]->Detach(this->inter_plane_devices[access_idx].Ge
t(currAdjNodeID+1)->GetObject<CsmaNetDevice> ());
        std::pair< Ptr< Ipv4 >, uint32_t> interface =
this->inter_plane_interfaces[access_idx].Get(currAdjNodeID+1);
        interface.first->SetDown(interface.second);

this->inter_plane_channels[access_idx]->Reattach(this->inter_plane_devices[access_idx].
Get(nextAdjNodeID+1)->GetObject<CsmaNetDevice> ());
        interface = this->inter_plane_interfaces[access_idx].Get(nextAdjNodeID+1);
        interface.first->SetUp(interface.second);
        this->inter_plane_channel_tracker[access_idx] = nextAdjNodeID;
        double new_delay = (nextAdjNodeDist*1000)/speed_of_light;
        this->inter_plane_channels[access_idx]->SetAttribute("Delay",
TimeValue(Seconds(new_delay)));
        std::cout<<"New channel between plane "<<i<<" satellite "<<j<<" and plane
"<<(i+1)%num_planes<<" satellite "<<nextAdjNodeID<< " with distance
"<<nextAdjNodeDist<< "km and delay of "<<new_delay<<" seconds"<<std::endl;
      }
    }
  }

  //updating links between ground stations and their closest satellites
  for (uint32_t i=0; i<2; i++)
  {
    Vector gndPos = ground_stations.Get(i)->GetObject<MobilityModel> ()->GetPosition();
    uint32_t closestAdjSat = 0;
    uint32_t closestAdjSatDist = 0;
    uint32_t planeIndex;
    if (i == 0)
      planeIndex = 0;
    else
      planeIndex = floor(3*num_planes/7);
    //find closest adjacent satellite for ground station
    for (uint32_t j=0; j<this->num_satellites_per_plane; j++)
    {
      Vector pos =
this->plane[planeIndex].Get(j)->GetObject<MobilityModel>()->GetPosition();
      double temp_dist = CalculateDistanceGroundToSat(gndPos,pos);
      if((temp_dist < closestAdjSatDist) || (j==0))
      {
        closestAdjSatDist = temp_dist;
        closestAdjSat = j;
      }
    }

    uint32_t currAdjNodeID = this->ground_station_channel_tracker[i];
    if(currAdjNodeID == closestAdjSat)
    {
      double new_delay = (closestAdjSatDist*1000)/speed_of_light;
      this->ground_station_channels[i]->SetAttribute("Delay",
TimeValue(Seconds(new_delay)));
      std::cout<<"Channel updated between ground station "<<i<<" and plane
"<<planeIndex<<" satellite "<<closestAdjSat<< " with distance "<<closestAdjSatDist<<
"km and delay of "<<new_delay<<" seconds"<<std::endl;
      }
      else
      {

this->ground_station_channels[i]->Detach(this->ground_station_devices[i].Get(currAdjNod
eID+1)->GetObject<CsmaNetDevice> ());
        std::pair< Ptr< Ipv4 >, uint32_t> interface =
this->ground_station_interfaces[i].Get(currAdjNodeID+1);
        interface.first->SetDown(interface.second);
```

```
this->ground_station_channels[i]->Reattach(this->ground_station_devices[i].Get(closestA
djSat+1)->GetObject<CsmaNetDevice> ());
        interface = this->ground_station_interfaces[i].Get(closestAdjSat+1);
        interface.first->SetUp(interface.second);
        this->ground_station_channel_tracker[i] = closestAdjSat;
        double new_delay = (closestAdjSatDist*1000)/speed_of_light;
        this->ground_station_channels[i]->SetAttribute("Delay",
TimeValue(Seconds(new_delay)));
        std::cout<<"New channel between ground station "<<i<<" and plane
"<<planeIndex<<" satellite "<<closestAdjSat<< " with distance "<<closestAdjSatDist<<
"km and delay of "<<new_delay<<" seconds"<<std::endl;
      }
  }

  //Recompute Routing Tables
  std::cout<<"Recomputing Routing Tables"<<std::endl;
  Ipv4GlobalRoutingHelper::RecomputeRoutingTables ();
  std::cout<<"Finished Recomputing Routing Tables"<<std::endl;
}

}
```

leo-satellite-example.cc

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
/*
 * LEO Satellite Example
 * Runs traffic through a configurable LEO satellite constellation
 *
 * ENSC 427: Communication Networks
 * Spring 2020
 * Team 11
 */

#include "ns3/core-module.h"
#include "ns3/leo-satellite-config.h"
#include "ns3/applications-module.h"
#include "ns3/flow-monitor-helper.h"
#include <string>
#include <sstream>

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("LeoSatelliteExample");

int
main (int argc, char *argv[])
{
  uint32_t n_planes = 3;
  uint32_t n_sats_per_plane = 4;
  double altitude = 2000;

  CommandLine cmd;
  cmd.AddValue ("n_planes", "Number of planes in satellite constellation", n_planes);
  cmd.AddValue ("n_sats_per_plane", "Number of satellites per plane in the satellite 
constellation", n_sats_per_plane);
  cmd.AddValue ("altitude", "Altitude of satellites in constellation in kilometers ... 
must be between 500 and 2000", altitude);

  cmd.Parse (argc,argv);

  LogComponentEnable("UdpEchoClientApplication", LOG_LEVEL_INFO);
  LogComponentEnable("UdpEchoServerApplication", LOG_LEVEL_INFO);

  LeoSatelliteConfig sat_network(n_planes, n_sats_per_plane, altitude);

  UdpEchoServerHelper echoServer (9);

  ApplicationContainer serverApps = 
echoServer.Install(sat_network.ground_stations.Get(1));
  serverApps.Start (Seconds (1.0));
  serverApps.Stop (Seconds (2000.0));

  UdpEchoClientHelper echoClient 
(sat_network.ground_station_interfaces[1].GetAddress(0), 9);
  echoClient.SetAttribute("MaxPackets", UintegerValue (20));
  echoClient.SetAttribute("Interval", TimeValue(Seconds(100.0)));
  echoClient.SetAttribute("PacketSize", UintegerValue (1024));

  ApplicationContainer clientApps = echoClient.Install 
(sat_network.ground_stations.Get(0));
  clientApps.Start (Seconds (2.0));
  clientApps.Stop (Seconds (2000.0));

  FlowMonitorHelper flowmonHelper;
  flowmonHelper.InstallAll();

  for(uint32_t i=0; i<19; i++)
  {
      Simulator::Stop(Seconds(100));
      Simulator::Run();
      sat_network.UpdateLinks();
  }
  Simulator::Stop(Seconds(100));
  Simulator::Run();

  Simulator::Destroy ();

  std::stringstream ss;
  std::string file_title;
  ss << 
"leo-satellite-example-"<<n_planes<<"-"<<n_sats_per_plane<<"-"<<altitude<<".flowmon";
  ss >> file_title;
  flowmonHelper.SerializeToXmlFile (file_title, false, false);
  return 0;
}
```