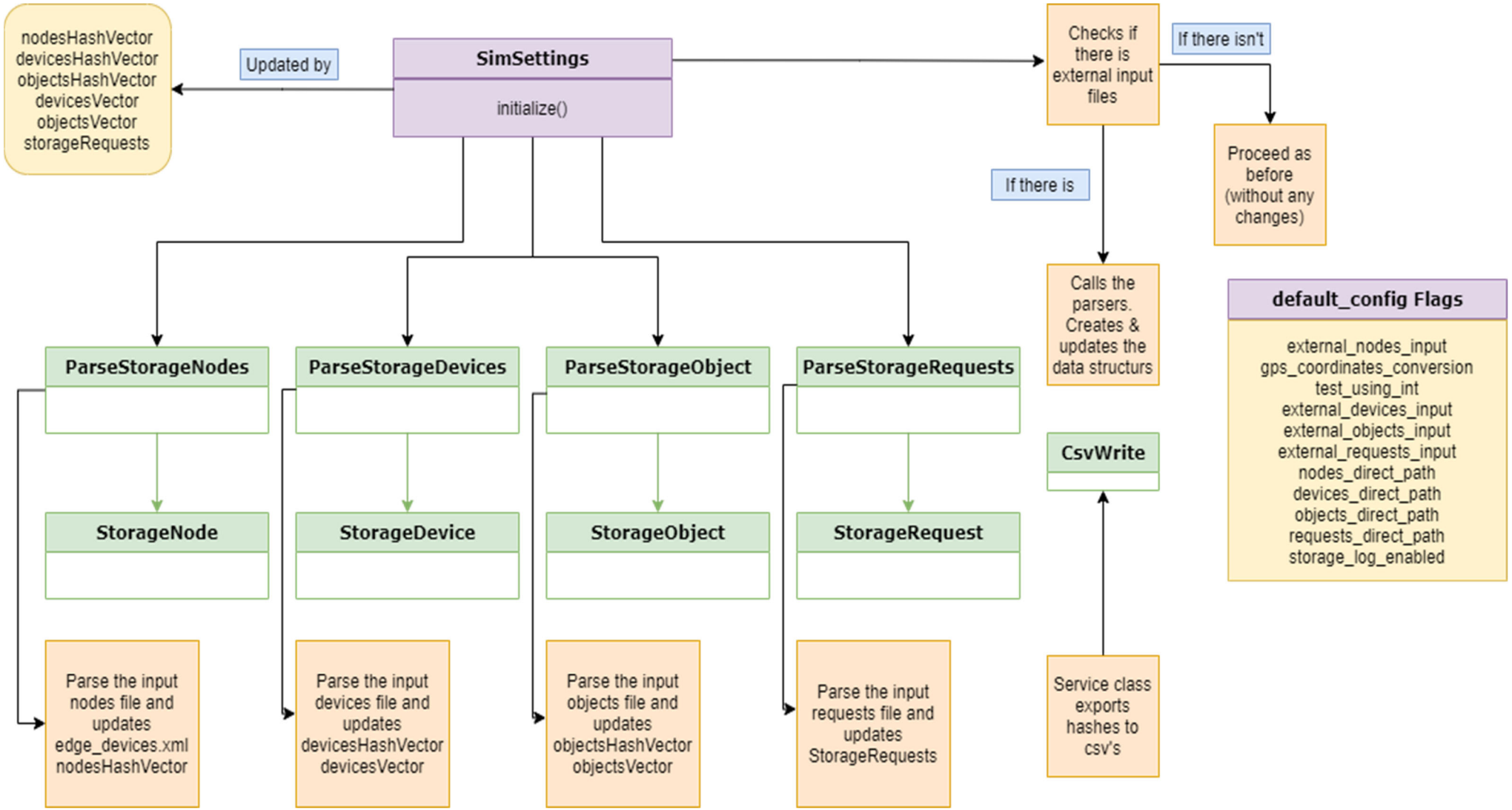
Project in Information Storage Systems – Adapting the SimCloud simulator for external input.

Presented by: Harel Karni

Table of contents

The First topic: The code flow (only outline)



The Second topic: Complete description of the new modules

Four modules had been written (classes and data structures) in additions to a service class designed for exporting the required data into csv files.

It should be mentioned that almost every module contains a parsing section and a data structure designed specifically to hold all the requirements and the data extracted by the parser section.

The first module is handling the Nodes. As it appears in the simulator, the nodes are practically the 'Edge centers' containing data and handling requests.

This module is implemented by two classes: the ParseStorageNodes and StorageNode.

The class StorageNode contains the data structure which hold every node in the system. Later, every StorageNode will be inserted to the simulator as an Edge center.

The data structure StorageNode contains the following attributes (partial list):

- nodeName: holds the conventional name of the node (integers in an ascending order).
- xPos and yPos: holds the location of the node by x and y coordinates. (The description of the grid and its alignment will be provided later)
- serviceClass: will indicate priorities/different sources of the nodes
- capacity: holds the maximum amount of data the node can hold.
- serviceRate: units: MB/s

Accept from getters and setters for the described attributes, there are no special functions in this class.

The class ParseStorageNodes contains the following attributes (partial list):

- xMin, yMin: holds the x and y coordinates of the bottom left point in the nodes grid. (It is not necessarily meaning that exists node with this coordinates)
- xRange, yRange: holds the x and y coordinates of the top right point in the nodes grid. (It is not necessarily meaning that exists node with this coordinates)

It also contains two functions (other than getters):

- prepareNodesHashVector
- xmlWrite

The prepareNodesHashVector function is getting an input file (in csv format) that contains the nodes, and parse it in a way that every node from the file is imported to the function and exported to three data structures:

1. nodesVector: a StorageNode is created for every node in the input file and inserted to that vector.
2. hashMap: every node has its own name (some string that imported from the input file). In this function every node gets a new name that comply with the requirements (integers in an ascending order).
3. Edge_devices.xml: this file contains the Edge centers for the simulator. After importing the nodes from the file, they exported to the xml file by the xmlWrite function that will be described later on.

In addition to all that, this function does some adjustments to the nodes grid. Because the bottom left point of the nodes' grid does not necessarily (0, 0), an adjustment is made to all the nodes coordinates to make it so and keep the relative locations of the nodes. Also, the top right point on the adjusted grid is stored for later use.

At the end, after all the nodes had been imported successfully, the hashMap (between the imported names and the conventional ones that were given to them) is exported to the service class CsvWrite that opens a new csv file (that by default is called "Nodes_Hash.csv") and writes the hashMap to that file.

Note: the name of each imported node must be unique. If we encounter two nodes with the same name, the simulation will stop immediately, and a proper error message will be presented to the use stating the cause to the error (the node's name and line in the input file).

The xmlWrite function is straight forward. It gets the nodesVector made by the previous function and, using the conventions and requirements needed by the simulator, writes (rewrites if necessary) a new file that is called "Edge_devices.xml".

The second module is handling the Devices. As it appears in the simulator, the devices are the 'clients' that send requests to the data centers (they are mobile devices from any kind and as such, the simulator calls them that).

This module is implemented by two classes: the ParseStorageDevices and StorageDevice.

The class StorageDevice contains the data structure which hold every device in the system. Later, every StorageDevice will be inserted to the simulator as a mobile device.

The data structure StorageDevice contains the following attributes (partial list):

- deviceName: holds the conventional name of the node (integers in an ascending order).
- xPos and yPos: holds the location of the device by x and y coordinates. (The description of the grid and its alignment will be provided later)
- time: a time stamp parameter (will be used later to indicate that the device had change its location)

Accept from getters and setters for the described attributes, there are no special functions.

The class ParseStorageDevices contains the following attribute (partial list):

- devicesVector: holds all the devices imported by the class (updated by the prepareDevicesVector function).

It also contains one function (other than getters):

- prepareDevicesVector

The prepareDevicesVector function is getting an input file (in csv format) that contains the devices, and parse it in a way that every device from the file is imported to the function and exported to two data structures:

1. devicesVector: a StorageDevice is created for every device in the input file and inserted to that vector.
2. hashMap: every device has its own name (some string that imported from the input file). In this function every node gets a new name that comply with the requirements (integers in an ascending order).

The function checks several things such as that every device was declared before use (declaration before use of a device is its presence in the devicesVector in time 0 before accepting its presence in the devicesVector with time other then 0).

At the end, after all the devices had been imported successfully, the hashMap (between the imported names and the conventional ones that were given to them) is exported to the service class CsvWrite that opens a new csv file (that by default is called "Devices_Hash.csv") and writes the hashMap to that file.

Note: as well as the nodes, the devices grid needs to be adjusted to the nodes grid by the same parameters. It will be explained later how it has done and where.

The third module is handling the Objects. As it appears in the simulator, the objects are the data requested by the devices from the nodes (they have the same terminology in the simulator as well).

This module is implemented by two classes: the ParseStorageObjects and StorageObject.

The class StorageObject contains the data structure which hold every object in the system.

The data structure StorageObject contains the following attributes (partial list):

- objName: holds the conventional name of the object (prefix 'd' with integers in an ascending order).
- objSize: holds the size of the object (the units are the same as the node's capacity attribute).
- objLocations: holds a list of nodes that contains the object on them (can be empty).
- objLocationsProb: holds a list of probabilities (sum of all probabilities must be 1 at most) its use will be determined later (by Oleg). Also, can be empty.
- objClass: holds the class of the object.
- type: holds the objects' type (data by default, may be changed later).

Accept from getters and setters for the described attributes, there are no special functions in this class.

The class ParseStorageObjects contains the following attribute (partial list):

- objectsVector: holds all the objects imported by the class (updated by the prepareObjectsVector function).

It also contains one function (other than getters):

- prepareObjectsVector

The prepareObjectsVector function is getting an input file (in csv format) that contains the objects, and parse it in a way that every object from the file is imported to the function and exported to two data structures:

1. objectsVector: a StorageObject is created for every object in the input file and inserted to that vector.
2. hashMap: every object has its own name (some string that imported from the input file). In this function every object gets a new name that comply with the requirements (prefix 'd' with integers in an ascending order).

In addition to all that, the function checks that the objLocations has the same number of elements as the objLocationsProb. If they do not contain the same number of elements, the simulation will stop, and a proper error message will be presented to the user. Additionally, the object name needs to be unique and the elements of the objLocations must appear as nodes in the nodesHash. If one (or more) of those criteria does not met, as before, the simulation will stop, and a proper error message will be presented to the user.

The fourth module is handling the requests. As it appears in the simulator, the requests are an attempt to access data objects stored on the nodes, by the devices (they have the same terminology in the simulator as well).

This module is implemented by two classes: the ParseStorageRequests and StorageRequest.

The class StorageRequest contains the data structure which hold every request in the system.

The data structure StorageRequest contains the following attributes (partial list):

- deviceName: holds the conventional name of the device requesting this object.
- time: holds the time the request was made.
- objectID: holds the name of the object requested.
- ioTaskID: the ID of the task
- taskPriority: holds the priority of the task. Can be empty.
- taskDeadLine: holds the deadline (time stamp) by which the request should be processed. Can be empty.

Note: this class has more than one constructor.

Accept from getters and setters for the described attributes, there are no special functions in this class.

The class ParseStorageRequests contains one function (other than getters):

- prepareRequests

The prepareRequests function is getting an input file (in csv format) that contains the requests and parse it in a way that every request from the file is imported to the function and exported to the requestsVector data structure. The requestsVector data structure holds all the StorageRequests imported from the requests input file.

Note: the function gets two hashMaps containing the nodes and devices.

A numerous check is done:

1. Check that the objectID is compatible with one of the objects in the objects hash vector.
2. Check that the deviceName is compatible with one of the devices in the devices hash vector.
3. Check that the time of the requests is an ascending order.

If one (or more) of those criteria does not met, as before, the simulation will stop, and a proper error message will be presented to the user.

The Third topic: Changes made to the original simulator

In this section, the classes and changes made to them will be named. Only substantial changes will be specified here.

The SimSettings class:

General section:

I added some attributes (to the class itself) that indicate if we are currently wanting to use external input or not. Each specific attribute will be updated from the default_config file. (It will be done automatically so the right parameters should be inserted to the default_config file before running the simulation)

Also, if we wish to give the simulator external files, we need to put them in the "input_files" directory by default. If we wish to read the input files from anywhere else (by full path of course) we can do that by giving the default_config file, the correct parameters which will be imported to that section in the class as well.

In addition, four hash-maps were added to the class:

1. nodesHashVector – holds the mapping between the original names of the nodes and the conventional one given to them earlier.
2. devicesHashVector – holds the mapping between the original names of the devices and the conventional one given to them earlier.
3. objectsHashVector - holds the mapping between the original names of the objects and the conventional one given to them earlier.
4. reversedHashVector - holds a reverse mapping between the conventional names and the original names of the objects.

Also, three vectors were added to the class:

1. devicesVector – holds the devices vector created by the ParseStorageDevices class.
2. objectsVector – holds the objects vector created by the ParseStorageObject class.
3. storageRequests – hold the requests vector created by the ParseStorageRequests class.

Last, five variables were added to the class:

1. minXpos – holds the x coordinate of the far-left node.
2. minYpos – holds the y coordinate of the far-down node.
3. xRange – holds the x coordinate of the far-right node.
4. yRange – holds the y coordinate of the far-up node.
5. numOfExternalTasks – holds the number of tasks (requests) imported by the ParseStorageRequests class.

For all the attributes above, getters were added if we need them. (Not all the attributes have getters and almost no-one has setters).

The initialize function:

As mentioned before, all the attributes above need to be updated before we can use them. The updating process is done here (like other flags and attributes that the simulator already has). We read the specific flags in the default_config relevant to the attribute and updating it accordingly.

Before the changes, at the end of this function, the simulator parses the two xml files and after that, finishes the initialization process. After parsing the first xml, a code was added that checks the flags indicating that the simulator will get the data externally instead of generating its own data. During the checks, the proper attributes and data structures are updated by the classes added to the simulator and the parameters (as specified in the requirements document) are being checked accordingly. If some input file is incorrect, or does not meet the requirements provided, the simulator will stop and present a proper message to the user indicating the error that occur. (Includes line number and what went wrong)

In this section the user can decide which part of the simulation will be provided externally. For example, the user can decide that only the nodes and devices will be provided externally but the objects and requests will be generated by the simulator. (Basically, the user can mix between the external input and the synthetic data created by the simulator as long is the proper flags in the default_config file is updated)

Note: It is important to mention that all the previous functionality was save. If the user chooses to run the simulator without giving it external files to process, the simulators will run normally as before.

The reverseHash function:

This function was added to reverse the objects hash and save the reversed hash in the proper attribute mentioned before. It will save time and calculations later in the code.

The ObjectGenerator class:

The constructor checks if the simulator is in external objects mode and if it is, the function importObjectsFromFile is called to create the data objects list. If its not, the simulator will resume the original function generates synthetic objects as before.

The importObjectsFromFile function:

The function goes over the data objects imported from the input file and insert them to a list of data objects and returns that list.

The Task class:

I have created new constructor to comply with the additional attributes required but it turns out that the Task class is abstract and uses the constructor of the parent (super - Cloudlet) therefor it cannot be changed (the Cloudelet class is in the binary classes).

The IdleActiveStorageLoadGenerator class:

The initializeModel function:

When the function begins, it will check if the user wanted the simulator to generate the tasks (requests) or they will be provided by external input. If the tasks will be imported from file (the proper flag will indicate this), the function initializeModelWithRequestsFromInput will be called. As mentioned before, if the user does not want to provide the tasks (from input), the simulator will continue this function without calling the other one.

It is important to say that that function was very long, so it was divided to two functions. At the end of the current one, the checkModeAfterInit function is called to complete all the tasks the previous longer function was doing before the changes.

The initializeModelWithRequestsFromInput function:

Creates the task list for the simulator using the tasks (requests) provided externally. Also, updates some system variables indicating the number of tasks etc.

The function creates a taskProperty for each request in the requests vector and updates the internal variables required for creating a taskProperty object. Some data (attributes) was added to the taskProperty object that was not in the previous version of the simulator. For the sake if that, a new constructor was created at the class TaskProperty. The new constructor can get all the new parameters and updates the required attributes accordingly. At the end of the function, the checkModeAfterInit function is called to complete all the tasks the original function had.

The createParityTask function:

A switch has been added to indicate to the simulator that the user wants to use external data. If the tasks (requests) are imported from an external file, the function calls a different TaskProperty constructor (with different parameters) and insert it to the task list. If the simulator generates the tasks synthetically, the old code will take place with no changes.

Note: There are probably other places in that class that were changed so the new TaskProperty constructor will have its all required parameters. If the function adds a task to the task list, it was probably changed since the original code.

The TaskProperty class:

The following attributes were added to the class:

1. taskPriority – allow the user to give the tasks priorities.
2. taskDeadline – allow the user to give the tasks deadline.

The attributes were added according to the requirements to provide the user flexibility when assigning tasks.

As mentioned before, a new constructor was made for the additional attributes to be inserted to the simulator. In that constructor the new attributes are being imported to the proper fields in the taskProperty object.

The StaticRangeMobility class:

The initialize function:

A switch was added to determine if the simulator should generate synthetic devices or use the ones provided by an external file. If the devices are imported, the function realPlaceDevice is called upon with the number of the device to check that the device is in range of a least one node. After returning from the function above, the new location object is inserted to a treeMapArray that holds all the devices.

The realPlaceDevice function:

Gets an index of a device from the caller and checks if the device is in range of at least one host (node). If not, stop the simulation and prints a proper message to the user.

The function creates a location object for the device and places it on the hosts' (nodes) grid. After the placement, the checkLegalPlacement function is called. (As the original code does) It checks how many hosts contains the device in their range and if there is none, stop the simulation and prints a proper message to the user. If there is more then one host available for the device, the function chooses the nearest host (As the original code does). The function returns the new location object created for the device.

The MainApp class in sample_app6:

Holds the entire changes made to the simulator in order to get it to work with external inputs.

If the changes need to be reverted for any reason or the original simulation is required, try to run the MainApp class from sample_app5 directory. The changes made to the current version are not affecting the original runs. (As requested)

The Fourth topic: How to use the adapted simulator

The changes required for running the simulator with external input files are:

1. The **default_config** file properties changes (the Input properties sector):
    a. **external_nodes** input parameter should be true if the nodes are imported from a file.
    b. **test_using_int** parameter should be true to keep using integer coordinates instead of real ones. (In order to use real coordinates, some changes need to be implemented in the code)
    c. **external_devices_input** parameter should be true if the devices are imported from a file.
    d. **external_objects_input** parameter should be true if the objects are imported from a file.
    e. **external_requests_input** parameter should be true if the requests are imported from a file.
    f. If you wish the input files will be read from the "input_files" directory in the simulator package, leave the parameters: **nodes_direct_path**, **devices_direct_path**, **objects_direct_path**, **requests_direct_path** empty. If you wish to give the simulator a path to one or more of the files, insert the full path of the proper input file to the proper parameter. (For example: if you want to get the nodes from the file Nodes.csv with the full path be "scripts/sample_app6/input_files/Nodes.csv" the proper line will be: **nodes_direct_path =scripts/sample_app6/input_files/Nodes.csv** without spaces)

2. The input file should be in the "**input_files**" directory (for other oaths for the input files, check the section above) by default and the hashes will be exported to the "**hash_tables**" directory in the simulator package.

3. The original simulator creates a grid for the hosts (nodes) and devices that goes between $(0,0)$ to $(xRange, yRange)$. To adapt the grid created by the nodes and devices given by the input files, some conversion is required (the real-world coordinates can be negative). The conversion made is the following: after the simulator gets the hosts (nodes) from the external file it finds the host with the far-left x coordinate and the host with the far-down y coordinate and convert the original grid to a new grid with the coordinates found will be converted to $(0,0)$. The simulation also finds the host with the far-right x coordinate and the host with the far-up y coordinate and that coordinates will become $(xRange, yRange)$ for the simulator after the conversion.

    Because of the conversion mentioned above, the same adjustments needed for the devices on the grid. (So, their locations will much the hosts locations) the converted coordinates are exported to an attribute in the simulator (mentioned above) and they will be used later to convert the grid of the devices as well.

In order to do the conversion (the simulator will not run with external files without using the coordinate conversion) the flag **gps_coordinates_conversion** needs to be true.

The outputs of the original simulator (files and on-screen) will be presented in the same directories and same formats as the synthetic ones.

Be advised:

1. The input files need to meet the requirements provided (included at the end of this document). If one or more of the files is not in the format provided, the simulator will stop and provide a proper error message indicate the first line that cause the error.
2. If a combination between external input and synthetic data generated by the simulator required, it can be done by applying the correct flags at the default_config file. With that being said, the changes allow the user to combine between the two with the following restriction: it can be done only from the top, down.
   For example, if the user wants to enter external devices input file, an external nodes input file is also required. That is because they based one on the other. The same goes for the objects (requires the nodes and the devices). The requests (if the user wants to get them from an external input file) require from the user to enter all the four files externally.
   The reason behind this is that after the simulator creates the data (lets say the nodes for example) when the user enters other data externally, the inserted data would not be compatible with the data thae simulator generates because the user does not have access to it before the simulation starts.

The Fifth topic: Challenges I ran into during the project

Overall, the main challenge was trying to figure out the how the simulator and all its components works and trying to understand where to put my modules and how they are affecting the existing code. It is important to understand that every piece of new or modified code inserted to the original one had impact and required changes in various entries in the simulator. Because the simulator is complex and has three layers (the basic simulator, the additional code required to support simulating the edge centers and the changes Oleg made for the storage part), in the beginning some deep understanding was required. Also, some of the code cannot be changed (the basic simulator that comes in a format of binary classes only), so the changes required some flexibility in order not to cause damage to the original simulator as well as adapting it to the provided requirements.

There were not test files for checking that the adapted simulator is doing what it supposed to, so some test files needed to be written to check the correctness of the modules before inserting them into the simulator. It took a while because Oleg needed to explain to me some storage concepts and how he implemented them in the code (in order not to damage the adaptations he already made to the simulator, I needed to figure out how to integrate the new modules into the code without damaging the storage properties he tried to achieve in the original simulator).

In the next few lines, I will explain some (not all) of the concepts or code challenges I ran into and what I decided to do to overcome the difficulty:

1. The edge_devices file:
   After getting the nodes from the files, some methods were considered by Oleg and me regarding to the insertion of the nodes into the simulator. The file holding the nodes is the edge_devices file and it needs to be changed. We thought (in the beginning) that we will use the existing method (the parser in the SimSettings initialize function) in order to do so, but as it turns out, it was very complex compared to the final implementation. I decided to create a function in the module parses the nodes that overwrites the edge_devices file and by doing so, import all the nodes from the input file into the simulator.

2. In order to check all the requirements (such as that every request is done to an existing object file, or every object is located on a viable legitimate node) some data exchange between the modules importing the data from the input files was required. I implemented the necessary methods and attributes in the modules I wrote but as it turns out, some methods in the simulator required the same data structures during its run. The data structures were changed, and the necessary attributes and methods were added to the simulator in order to support access to the data from the simulator too.
3. Some of the objects generated in the original code were to be changed so I created new constructors for the classes that created the matching objects. Some of the objects are made from an abstract class and the constructor that creates them is in the binary classes so it can not ne changed. (For example, the class Task)
4. In order to know if the simulator will run on synthetic generated data or on data imported from input files, we needed some flags that will be updated right from the start, and we needed that the way to change between the two options will be relatively easy to use. Some flags were created in the default_config file that the user can change easily (before running the simulation) and indicates the simulator the mode we are in. The user also can change the paths to the input files from the same default_config file. After updating the simulator internal flags, every function that had changed checks the mode we are in and do a different set of orders accordingly.
5. The original synthetic grid of hosts is made from $(0,0)$ to $(xRange, yRange)$ so it always was with positive values only. The new updated grid (with hosts imported from the nods input file) can have positive and negative values. Because Oleg said that some parts of the simulator does not comply with negative value on the grid, we needed to find a way to convert it to the first quarter grid. But, after doing so, some devices located in the negative zone yet with in radius of one of the hosts, will be out of range because of the positivity of the grid. In order to fix the problem, we extended the grid by host radius and adjusted the nodes grid and the devices grid (the same one but in different functions and classes) accordingly.

Only the main challenges were described in this section. There were many additional challenges that were handled with Oleg assistance during mails or video calls via zoom.

The Sixth topic: Tasks left to do and recommendations

There are mainly two things that was required by Oleg and have not completed:

1. The Task class new constructor that is required to hold all the new parameters described by Oleg:
   Every request needs to have more fields which mean the StorageRequest has more attributes to it. In order to insert the new attributes provided by the input files into the simulator task list, the data structure Task must be updated (needs to add additional attributes to it). Because the Task class constructor uses the Cloudlet class constructor (super) in order to construct a new Task object, and we can not change the Cloudlet class (because it is binary class) we cannot change the form of the Task object. Even so, I wrote the required constructor and checked if I can get the simulator to use it instead of using the original but without any success.
2. The location of the hosts (nodes) needs to be able to be real and not only integers:
   That task was part of the main requirements but after I started to work on it, I discovered it is too vast to complete with everything else in the project. The main change is to the class Location. It needs to be changed in order to support real numbers in addition to integers. The Location class is responsible for the objects on the grid (hosts and devices) and therefor has a lot of dependencies in the original simulator. Many methods in the original simulator uses the fact that the coordinates are integers so changing that required a widely deep change across the simulator. I tried to make the necessary changes regarding the Location class and got a bunch of errors from across the simulator classes. That change in my opinion needs to be very precise and completely cover all the classes and relevant functions in the simulator and would require changes in the main simulator. (For example, the Task class uses a Location object in its constructor. In order to update the location of the task, the function setSubmittedLocation is called in SampleMobileDeviceManager class. If we change the object (we will create a different constructor with float attributes instead of integer) all the calculation with such objects will need to be duplicated and replaced with the proper variables.)
   The code I wrote is handling a float coordinates (when it will be supported by the proper changes to the simulator), but in the meantime there is a flag that cause

some converting for testing purposes with integer coordinates. That flag as well as the other flags can be changed in the default_config file before running the simulator.

The Seventh topic: Oleg's requirements for the project

Nodes

List of edge node (hosts). Will replace edge_ devices.xml

- nodeName
  - Name is unique for each host – Need to check
  - Host names in EdgeCloudSim are numerical (0,1,2…). Need to rename and output hash table.
- xPos
  - x, y coordinates are based on the geographic coordinate system (Latitude and longitude).
  - EdgeCloudSim currently uses a grid between 0 and x_range, y_range.
  - Need to change EdgeCloudSim such that grid will be in range of $(xRange_0, yRange_0)$ and $(xRange_1, yRange_1)$.
  - Type double (can be negative)
- yPos
  - Same as xPos
- serviceClass
  - Will indicate priorities/different sources of the nodes.
  - Currently does not exist in EdgeCloudSim
  - Probably will not require any special handling at the moment
  - Integer
- capacity
  - Will replace “storage” field in edge_devices.xml
  - Integer
  - Units: KB
- serviceRate
  - Integer
  - Units: MB/s

Devices

List of users (mobile devices).
Initially all devices are declared at time 0. Mobility is possible – device can move at time>0.

- deviceName
  - Name is unique for each device at time 0 – Need to check
  - Need to check that device is declared at time 0 if it appears at time>T
  - Host names in EdgeCloudSim are numerical (0,1,2…). Need to rename and output hash table.
  - Start time might not be 0
- xPos
  - x, y coordinates are based on the geographic coordinate system (Latitude and longitude).
  - EdgeCloudSim currently uses a grid between 0 and x_range, y_range.
  - Need to change EdgeCloudSim such that grid will be in range of $(xRange_0, yRange_0)$ and $(xRange_1, yRange_1)$.
  - Type double (can be negative)
- yPos
  - Same as xPos
- time
  - Type double
  - Units: seconds
  - Need to verify time is in ascending order

Objects

Currently synthetically generated in ObjectGenerator class and inserted at this point:

dataObjects = createDataObjects(numOfDataObjects, Integer.toString(this.objectSize))

- objectName
  - Name is unique for each object - Need to check
  - Naming convention is numerical + "d" prefix (d0, d1, d2…). Need to rename and output hash table.
- size
  - String
  - Units: B
- locationVector
  - List of nodeName (should match names in Nodes file – need to check)
- locationProbVector
  - List of probabilities to be in each node of locationVector. Position indices match locationVector.
  - Lengths of locationVector and locationProbVector are equal – Need to check
  - Sum should be <=1
  - These 2 vectors currently do not exist in the object HashMap. Needs to be added.
- class
  - Will indicate priorities/different sources.
  - These 2 vectors currently do not exist in the object HashMap. Needs to be added.

Requests

The list of events.
Currently generated synthetically in IdleActiveStorageLoadGenerator and inserted in:
taskList.add(new TaskProperty(i,randomTaskType, virtualTime, objectID, ioTaskID, 0,expRngList));

- deviceName
  - Matches names from Devices file. Need to check.
- time
  - Time of the event
  - Type Double
  - Units: seconds
  - Need to verify time is in ascending order
- objectID
  - Matches Objects file naming. Need to check.
- ioTaskID
  - The ID of the task. Name convention is 0,1,2…
  - Integer
- taskPriority
  - Currently doesn't exist in TaskProperty. Needs to be added.
  - Can be empty
  - Integer
- taskDeadline
  - Currently doesn't exist in TaskProperty. Needs to be added.
  - Double
  - Can be empty
  - Units: seconds