Latency-Aware Data Replication for 5G Networks
Final Project Report

CS552 - Computer Networks

Harshankar Sai Reddy Gudur (hsg64)
Vishal Singh (vs872)

1

Abstract

This project addresses the critical challenge of latency in 5G networks, particularly for real-time applications like augmented reality (AR), virtual reality (VR), and online gaming, where even a few milliseconds of delay can severely impact user experience.

By designing an optimized data replication strategy, we aim to minimize the time it takes for users to access data, making these applications more responsive and scalable. This project is meaningful because it leverages edge computing to ensure that frequently accessed data is placed closer to users, thereby reducing network congestion and also improve service reliability as we ensure fault tolerance.

2
Introduction

This project focuses on evaluating the performance of a latency-aware data replication strategy tailored for 5G networks. Using edge nodes, the system aims to bring data closer to users, thereby reducing latency and improving the responsiveness of real-time applications. OMNeT++ serves as the primary tool for simulating network conditions, user behavior, and replication strategies. A custom replication manager is also implemented to determine the optimal data placement based on access patterns of the users and deem if it is worth it to replicate data based also on latency measurements.

2.1 Metrics

The project uses several metrics to evaluate the effectiveness of the proposed system. End-to-end latency is measured to understand the total time between a user request and data delivery, which is critical for applications like AR/VR and online gaming. Replication overhead is handled by ensuring no edge node contains more than 20 data items at a time. If replication is to occur, the least recently accessed data item is removed from the edge node. We also measure the cache hit/miss ratio for each of the edge nodes to see how effective the replication is. Finally, fault tolerance is evaluated to ensure that the system maintains data availability and access under simulated node failure scenarios. Other metrics, such as energy efficiency and throughput, are not considered as the primary focus of this project is to minimize latency.

2.2 System Description

The project involves the development of a latency-aware data replication system designed to optimize data placement in 5G networks. A replication manager dynamically determines where to replicate data based on real-time latency data and user behavior, while edge nodes store frequently accessed data to minimize dependency on core nodes. In this system, the core node contains all the data items, and the replication manager replicates data accordingly, from the core node to the edge node as and when it deems so during the simulation. This approach is particularly beneficial for latency-sensitive applications, including augmented reality (AR), virtual reality (VR), and online gaming, where responsiveness and minimal delay are essential.

We test this system for various configurations; by varying the total number of edge nodes, users, and data items in the system, and make note of the metrics in each of these configurations.

2.3 Intended Users and Potential Applications

The intended users of this system include application developers, who can take advantage of optimized data delivery for real-time applications, network engineers, who can implement the strategy to improve 5G network efficiency and reliability, and of course the end users, such as gamers and AR/VR enthusiasts, who also directly benefit from the improved application performance and reduced latency.

This technology has potential applications in various domains.

- **Augmented and Virtual Reality (AR/VR), and Online Gaming:** The system can be applied to AR/VR platforms to ensure responsive and immersive experiences, making it ideal for gaming, virtual training, or interactive simulations where minimal delay is important.

- **Smart Healthcare Systems:** Applications such as patient monitoring, real-time diagnostic systems, and wearable health devices can benefit from reduced latency, improving the accuracy and timeliness of healthcare services.

- **FinTech (Financial Technology):** In trading platforms and high-frequency financial applications, where milliseconds can determine outcomes, the system can reduce latency to improve transaction speeds and reliability.

- **Disaster Response and Public Safety:** Networks designed for emergency communication systems and public safety infrastructure can take advantage of the replication strategy to ensure reliable and timely information exchange during critical situations.

- **Edge Computing in IoT Networks:** The system can support IoT applications, such as smart grids and agriculture, by ensuring that data is processed and acted on quickly at the edge of the network. This system can be used for many more domains, any domain where minimizing latency is a critical metric.

In this project, OMNeT++ is utilized to simulate 5G network conditions and collect performance metrics. The results will demonstrate the efficacy of the proposed strategy in reducing latency and ensuring reliability in 5G networks.

3

Description/Methodology

3.1 Experimental Setup or User Interaction

As mentioned before, we use OMNeT++ for simulating our system. We decided to go with OMNeT++ as NS3 was not ideal for visualizing a proper simulation using our Windows Operating Systems. OMNeT++ mainly uses C++ as the base for implementing the functionality of the entities involved in simulations, and it has a few different files for changing the settings of the network itself. These are the types of files that we had to create/use in OMNeT++ for the simulation:

1. Network Description(.ned) File
2. Initialization (.ini) File
3. Message (.msg) Files
4. C++ header (.h) and Source Code (.cc) Files
5. Makefile
6. A JSON (.json) file to store synthetic data

Lets go through each of these files to make sure that a person following these steps can perfectly reproduce the simulation.

3.1.1 Pre-requisites

For replicating this project, we assume that the person reading this report is proficient in C++ and has the necessary technologies installed in their system. The technologies required for this would be:

1. The C++ compiler
2. Make, required to run the Makefile
3. The OMNeT++ IDE for C++.
4. A C++ library to handle JSON data. The one used in this project was Nholmann's library (which is also present in our project file under the *DataReplication/libs/single_include* directory).

3.1.2 The Data

For our project, we've created and used dummy data. This is for two reasons; the first being that various applications use a very diverse set of data. We wanted to store the parts of the data that mattered for the simulation instead of using actual

files that could span from 500MB to 5000MB each. The second reason is that it's easier for other people replicating this simulation with limited memory constraints. Using dummy data with only the metadata of the files that we're going to be sending allows us to focus more on the simulation itself, instead of other tasks like parsing the incoming data and the other practical considerations that would need to be made.

We create our data with a few important parameters in mind; an ID to uniquely identify a piece of data, the size of the data, the "type" of data it is (say it's data to load a webpage, or for a video; this would be useful while extending the system), and an access frequency counter initialized to 0. We store this data in a JSON file that will be used in our simulation. The sample data that we used is in the file submitted for the project with the name *synthetic_data.json*.

3.1.3 Using OMNeT++

Opening up the OMNeT++ IDE shows you the basic functionality available in any IDE. For running the project that submitted, you can simply open the folder using the IDE, go to any file and click on the run button. This should automatically run the simulation, that you can start by pressing the play button on the new window that popped up. Keep in mind that this isn't a tutorial on OMNeT++ . We explain the parts that are important for our simulation, but more detail on OMNeT++ can be found on their documentation online.

From here on, we'll be explaining the logic used in our project in case you want to create it from scratch, and how we accomplished that through OMNeT++ . Keep in mind that editing the source files can be done with any text editor, but running the simulation is something you'll need to use OMNeT++ for.

3.1.3.1 The NED file

The NED file is a Network Description file. It's where you define all the entities within your network, any information they should contain while being created (like an ID to uniquely identify them, for example) and how your network handles all of these entities.

We have 4 types of entities within our project, namely, the Replication Manager, the Core Node, the Edge Node, and the User. We will be creating these entities within the network in this NED file.

Let's start by defining our Core Node. We can do this by typing in:

```
simple CoreNode
{
    parameters:
        string type = "core";
        int nodeId = 0;
    gates:
```

```
        inout gate[];
}
```

Simple modules are atomic on NED level. They are also active components, and their behavior is implemented in C++. This declaration tells us that the Core Node has a `nodeId` of 0, has a `type` that stores the string "core", and has an *inout* gate (a gate that allows for data to be sent out, and to be received) vector, meaning that there are multiple connections that can be stored within this gate.

Similarly, let's define our Edge Node, User, and Replication Manager.

```
simple EdgeNode
{
    parameters:
        string type = "edge";
        int nodeId; // stores the nodeId for the node
    gates:
        inout gate[];
}

simple User
{
    parameters:
        string userId; // stores the userId
    gates:
        inout gate[];
}

simple ReplicationManager
{
    gates:
        inout gate[];
}
```

We have now defined the structure of every entity we need for our simulation, and the only thing left to describe is how all of these are connected together. Next, we define our network.

```
network DataReplicationNetwork
{
    parameters:
        int edgeNodeCount = default(5);
        int userCount = default(5);
        @display("bgb=520,467");
    submodules:
        // define the replication manager
        replicationManager: ReplicationManager {
            @display("p=249,53");  // Positioning the replication manager at the top
        }

        // define the Edge Nodes
        edgeNode1: EdgeNode {
            @display("p=50,300");
```

```
            nodeId = 1;
        }
        edgeNode2: EdgeNode {
            @display("p=150,300");
            nodeId = 2;
        }
        edgeNode3: EdgeNode {
            @display("p=250,300");
            nodeId = 3;
        }
        edgeNode4: EdgeNode {
            @display("p=350,300");
            nodeId = 4;
        }
        edgeNode5: EdgeNode {
            @display("p=450,300");
            nodeId = 5;
        }

        // define the core node
        coreNode1: CoreNode {
            @display("p=249,166");
        }

        // define the users
        user1: User {
            userId = "u1";
            @display("p=50,425");
        }

        user2: User {
            userId = "u2";
            @display("p=150,425");
        }

        user3: User {
            userId = "u3";
            @display("p=250,425");
        }

        user4: User {
            userId = "u4";
            @display("p=350,425");
        }

        user5: User {
            userId = "u5";
            @display("p=450,425");
        }

    connections allowunconnected:
        // Connect each edge node to the core node
        edgeNode1.gate++ <--> coreNode1.gate++;
        edgeNode2.gate++ <--> coreNode1.gate++;
        edgeNode3.gate++ <--> coreNode1.gate++;
        edgeNode4.gate++ <--> coreNode1.gate++;
```

```
72          edgeNode5.gate++ <--> coreNode1.gate++;
73
74          // Connect the replication manager to the core node
75          replicationManager.gate++ <--> coreNode1.gate++;
76
77          // Connect each user to an edge node
78          user1.gate++ <--> edgeNode1.gate++;
79          user2.gate++ <--> edgeNode1.gate++;
80          user3.gate++ <--> edgeNode2.gate++;
81          user4.gate++ <--> edgeNode2.gate++;
82          user5.gate++ <--> edgeNode3.gate++;
83
84          // Connect each edge node to the replication manager
85          edgeNode1.gate++ <--> replicationManager.gate++;
86          edgeNode2.gate++ <--> replicationManager.gate++;
87          edgeNode3.gate++ <--> replicationManager.gate++;
88          edgeNode4.gate++ <--> replicationManager.gate++;
89          edgeNode5.gate++ <--> replicationManager.gate++;
90
91          // Connect the users to the core node(s)
92          user1.gate++ <--> coreNode1.gate++;
93          user2.gate++ <--> coreNode1.gate++;
94          user3.gate++ <--> coreNode1.gate++;
95          user4.gate++ <--> coreNode1.gate++;
96          user5.gate++ <--> coreNode1.gate++;
97  }
```

In our `submodules` section, we create something akin to instances of the simple modules we defined above. In this configuration, we define 5 Edge Nodes and 5 Users. We assign a value to the `nodeId` and `userId` parameters in the Edge Node and User modules here as well. The `@display` function simply sets their positions in the simulation. This can also be changed by going to "Design" and moving the nodes manually.

Notice the connections made under the `connections allowunconnected` section. Since we defined our `gate` variable as an array of gates, each index can be used for a single connection. To made a second connection, we use the next index. The flow of data is described using the double arrow. Seeing the definition above, we see that the first connection of each Edge Node (at index 0) is to the Core node. This also means that the first 5 connections of the Core Node (index 0 through 4) are occupied for the Edge Nodes, and therefore index 5 gets used for the Replication Manager. Changing the number of these instances and connections allows you to try out multiple different configurations.

3.1.3.2 Node Functionality

We won't get into too much detail about each line of code written in our files, but instead define the basic functionality of each node and functions that assist us in implementing this functionality. We start by creating our *User.h*, a header file that creates the blueprint used by each user in the simulation. The functionality is coded

out in the *User.cc* file. Similarly, we have header files for the Edge Node, Core Node, and the Replication Manager.

To create our user class, we include the `omnetpp.h` header file, which allows us to inherit functionality defined by the people who created the simulator. Our file goes as follows:

```
#include "DataRequest_m.h"
#include "SendingUserData_m.h"
#include <omnetpp.h>
#include <unordered_map>
#include <string>
#include <vector>

using namespace omnetpp;

class User : public cSimpleModule {
private:
    std::string userId;            // to store the userId from our .ned file
    std::vector<std::string> dataItems; // List of available data items to request
    std::unordered_map<std::string, simtime_t> requests;

protected:
    virtual void initialize() override;
    virtual void handleMessage(cMessage* msg) override;
    void sendDataRequest();

public:
    User();  // Constructor
    virtual ~User(); // Destructor
};
```

We define the `userId` variable to store the *userId* assigned from our *data_replication.ned* file, the `dataItems` vector stores only the ID of the dummy data we created (so that they can make requests for it), and the `requests` map is used to store the exact time the request for a specific piece of data was requested. We use `simtime_t` as that's the defined datatype allowing us to properly store the time returned by the functions defined by the OMNeT++ header.

This `User` class that we created inherits from the `cSimpleModule` defined in `omnet++.h`. In our *User.cc* file, we go on to override the `initialize()` and the `handeMessage()` functions for our functionality. *User.cc*:

```
#include "User.h"
#include <fstream>
#include <omnetpp.h>
#include <string>
#include <vector>
#include <cstdlib>
#include <ctime>
#include <sstream>

```

```
using namespace omnetpp;

Define_Module(User);

User::User() {}

User::~User() {}

void User::initialize() {
    // Gets the userId defined in our .ned file
    userId = par("userId").stdstringValue();

    // Initialize available data items
    std::ifstream fJson("synthetic_data.json");
    std::stringstream buffer;
    buffer << fJson.rdbuf();
    auto json = nlohmann::json::parse(buffer.str());

    for (auto item: json) {
        dataItems.push_back(item["id"]);
    }

    // Schedule first data request
    scheduleAt(simTime() + 0.1, new cMessage("sendRequest"));
}

void User::handleMessage(cMessage* msg) {
    if (std::strcmp(msg->getName(), "sendingData") == 0) {
        SendingUserData *received = check_and_cast<SendingUserData *>(msg);
        std::string receivedData = received->getDataId();
        std::string receivedBubble = "Received " + receivedData;
        bubble(receivedBubble.c_str());

        // delete from our requests map
        requests.erase(receivedData);
    } else {
        sendDataRequest();

        // Schedule the next request
        float random_skip = float(intuniform(0, 1000)) / 500;
        EV << "Random skip of : " << random_skip << "s";

        scheduleAt(simTime() + random_skip, new cMessage("sendRequest"));
    }

    delete msg;
}

void User::sendDataRequest() {
    if (dataItems.empty()) {
        EV << "No data items available to request.\n";
        return;
    }

    // Randomly select a data item to request
    std::string requestedData = dataItems[intuniform(0, dataItems.size() - 1)];
```

```
66      std::string dest = "u_req_" + requestedData;
67      requests[requestedData] = simTime();
68
69      // Create and send custom defined message
70      char msgname[20];
71      snprintf(msgname, sizeof(msgname), dest.c_str());
72      DataRequest *request = new DataRequest(msgname);
73
74      request->setDataId(requestedData.c_str());
75      request->setUserId(userId.c_str());
76      send(request, "gate$o", 0);
77
78      std::string askingFor = "Requesting " + requestedData;
79      bubble(askingFor.c_str());
80      EV << "User " << userId << " sent request for data: " << requestedData << "\n";
81  }
```

Let's go through this:

1. **Define_Module**:
   We must use this `Define_Module` function to register the class in OMNeT++. This must be done for every class we create.

2. **initialize function**:
   We use this function to initialize a User. The `par` function is used for getting a specific parameter defined in the .ned file. Within this function, we read from our JSON file and store the ID of each data item into the `dataItems` vector declared in the header file.

   At the end of this function, we use the `scheduleAt` function (a pre-existing function with OMNeT++) to schedule a self-message. We send ourself a cMessage (a simple, pre-defined message type) with the name "sendRequest" set to be sent (and received) at 0.1s since the current simulation time, which at the beginning is 0s. The message we receive from this is handled in the `handleMessage` function.

3. **handleMessage**:
   This function is called whenever a message is received by this User instance. Since we sent ourself a self-message when initializing the class, this function gets called 0.1s into the simulation. Here, we check the message name. In our implementation, receiving a message with the name "sendingData" is when the Edge Node sends data that the User requested. If this is not the case, then the only other case is due to our self-message. Here call a function that sends a data request to the Edge Node, and schedule the next request.

4. **sendDataRequest**:
   A function we created that randomly selects a data ID, and requests that data from our Edge Node. Here, we use two new functions, `send` and `bubble`.
   The `bubble` function is purely for simulation purposes. It puts a text bubble with a message we set above the node in the simulation. This makes it easier to see what's going on.

   The `send` function is more interesting here. This function sends a specific mes-

sage from our node to another. Here, we send a message of type `DataRequest` to the node connected at our gate at index 0. From the NED file, we see that the first connection for every user (at index 0) is to the Edge Node, meaning this message is being sent to the Edge Node connected to this entity.

The `EV` is OMNet's equivalent of `cout` in the simulation console.

There are more lines of code in our *User.cc* file, but those are meant purely for collecting statistics and do not affect the functionality of the User.

Now that the functionality for the User is complete, let's move to the Edge Node. Similar to how we created our User class, here we create a class for our Edge Node and implement its functionality in the *EdgeNode.cc* file.

```
#include "DataRequest_m.h"
#include "CoreRequest_m.h"
#include "ReplicateRequest_m.h"
#include "ManagerMessage_m.h"
#include "SendingUserData_m.h"
#include "DataItem.h"
#include "json.hpp"
#include <fstream>
#include <map>
#include <omnetpp.h>
#include <string>
#include <climits>

using namespace omnetpp;

class EdgeNode : public cSimpleModule {
  private:
    std::map<std::string, DataItem> cachedData; // Data items stored locally
    int edgeId;
    int cacheLimit = 20;

  protected:
    virtual void initialize() override {

        // get the edge id
        edgeId = par("nodeId");

        // Caching some data in this edge node
        std::ifstream fJson("synthetic_data.json");
        std::stringstream buffer;
        buffer << fJson.rdbuf();
        auto json = nlohmann::json::parse(buffer.str());

        // a poor attempt to randomly store some data into each edge node
        int num_items = 0;
        int random_skip = intuniform(0, 70);
        int i = 0;
        for (auto item: json) {
            while (i++ < random_skip)
                continue;
            if (num_items >= cacheLimit || i >= 100)
```

```
                break;

            DataItem data;
            data.id = item["id"];
            data.type = item["type"];
            data.size = item["size"];
            data.access_frequency = 0;

            cachedData[data.id] = data;
            ++num_items;
        }

        return;
    }

    virtual void handleMessage(cMessage *msg) override {
        // this is the core node telling the edge node to replicate the data
        if (strcmp(msg->getName(), "replicate") == 0)
        {
            ReplicateRequest *rep_req = check_and_cast<ReplicateRequest *>(msg);
            // implement logic for replicating the sent data into the cache

            // send the data to the user who requested it
            sendDelayed(user_msg, time, "gate$o", userId);
        }

        // this is the user asking for some data
        else
        {
            DataRequest *ttmsg = check_and_cast<DataRequest *>(msg);

            if (cachedData.find(requestedData) != cachedData.end()) {
                // if the edge node has the data, send it to the user

                // send a message to the replication manager for informing it
                ManagerMessage *inform = new ManagerMessage(msgname);
                inform->setEdgeId(edgeId);
                inform->setDataId(requestedData.c_str());
                inform->setUserId(ttmsg->getUserId());

                send(inform, "gate$o", 2); // gate 2 is the replication manager for a
    single user

            } else {
                // the edge node doesn't have the data, request the core node
                CoreRequest *c_req = new CoreRequest(msgname);
                c_req->setEdgeId(edgeId);
                c_req->setUserId(ttmsg->getUserId());
                c_req->setDataId(requestedData.c_str());

                send(c_req, "gate$o", 0); // gate 0 is core node
            }
        }
    }
};

```

```
97 Define_Module(EdgeNode);
```

The fine details of the implementation are in the code file. Let's discuss the logic used here:

1. We use a map as a cache to store our data. This stores data of the type `DataItem`, a `struct` we created that contains all the attributes of the data mentioned before. The `cacheLimit` variable is used to decide the maximum number of items that the Edge Node can keep in its memory.
2. **check_and_cast**:
   When the Core Node sends data for the Edge Node to replicate, it sends a message with the name "replicate" of type `ReplicateRequest` which is a message type we've defined. If the name matches, we cast the `cMessage` received into a `ReplicateRequest` type to get further details from it. We then store this in the cache and send this data to the user using the `sendDelayed` function. This function sends a message through the connection we choose after some set time. Since OMNeT++ is a simulator and not an emulator, we have to decide the time it takes based on the data, and this time is based on the size of the data and the practical considerations of a distributed edge system. Therefore, larger data files have a higher transfer time than data files of smaller size.
3. **User Request**:
   If the message is not a replication command from the Core Node, then it's a User requesting some data. The Edge Node then checks if it already has this data. If it does, then it sends it to the user. If it doesn't, it requests the Core Node for this data using a message of the type `CoreRequest`. If it does have the data, it informs the Replication Manager of the request made for the specific data item using the `ManagerMessage` type.

Using the functions explained in the files above, we can create our Core Node and Replication Manager. The functionalities of both of these are as follows:

1. **Core Node**:
   - If it receives a request from the Edge Node for a piece of data, it sends it over.
   - It informs the Replication Manager of the data item that has been requested, and the Edge Node that requested it.
2. **Replication Manager**:
   - It receives messages from the Edge and Core Node informing it of the data items they've been requested. It keeps track of all the items requested, and the Edge Node it's been requested from.
   - If a data item has been requested a specific number of times (beyond some threshold), it tells the Core Node to replicate it at that Edge Node.

3.1.3.3 Message Files

We've created a variety of message files for this simulation. This is done because the data sent between any two entities could be different. For example, when a

User requests data from an Edge Node, it only knows the data ID and it's own ID (the userID), so it sends just these two. However, when the Edge Node requests data from the Core Node, it also needs to send it's own ID (edgeId) so that the Core Node knows which Edge Node to send the data to. Similarly, there are various scenarios where we need different message types, and these are the different message files we've created:

1. **CoreRequest**: When an Edge Node requests some data from the Core Node.
2. **DataRequest**: For when the User requests some data from the Edge Node.
3. **ManagerMessage**: For when the Edge/Core Nodes send the Replication Manager a message, informing them of the data item received, and for when the Replication Manager wants to tell the Core Node to replicate some data at a specific Edge Node.
4. **ReplicateRequest**: The Core Node telling the Edge Node to store this data in its cache.
5. **SendingUserData**: When the Edge Node sends the User the requested data.

Let's look at the *DataRequest* message file to see how to create a message.

OMNeT++ allows us to create message types in a *.msg* file. A message type available in the *omnet++.h* header file is `cMessage`, which holds no data. To create a message file, we create a new file within our root folder with the *.msg* extension. This file is used when sending the user some data, so it would only contain the dataId of the data that its sending. Additionally, you could choose to add some actual values corresponding to the data that is being sent in this file, but we don't do this as it has no impact on the simulation itself. A message file is as follows:

```
message DataRequest {
    string dataId;    // ID of the requested data
    string userId;    // ID of the user making the request
}
```

Similarly, all the other message files would contain the details that need to be sent. For example, the *CoreRequest* file would contain the dataId of the data requested, the edgeId of the Edge Node that received this data, and the userId of the user that requested it.

3.1.3.4 Makefile

The Makefile is automatically created when you create a new project, and needs to be modified according to the extra files you've created. This Makefile is what tells OMNeT++ the classes that need to be registered, the message files to be built, and any extra libraries that need to be linked for the functionality (like the Nholmann JSON library). We **modify** our Makefile as below:

```
# add this line
INET_LIBS += -ljsoncpp

# Object files for local .cc, .msg and .sm files
OBJS = \
```

```
  $O/ReplicationManager.o \
  $O/CoreNode.o \
  $O/User.o \
  $O/EdgeNode.o \
  $O/DataRequest_m.o \
  $O/CoreRequest_m.o \
  $O/ManagerMessage_m.o \
  $O/ReplicateRequest_m.o \
  $O/SendingUserData_m.o

# Message files
MSGFILES = \
  DataRequest.msg \
  CoreRequest.msg \
  ManagerMessage.msg \
  ReplicateRequest.msg \
  SendingUserData.msg
```

Here, we inform OMNeT++ of all our message files. These message files (along with our source code) get built and converted into an object file, which are then linked together to create our executable file (that OMNeT++ runs). The *.cc* files are set to be built by the other commands already mentioned in the Makefile, so we include these object files as well (how we've included EdgeNode.o, as EdgeNode.cc is used to make the object file) to tell the compiler to link them together.

The message files get two things; an object file (.o), and a header file (.h) that you include in your source code (.cc). You can see we've included these message files in the code for the *User.cc* and the *EdgeNode.cc* above. The compiler creates header files for the *.msg* files in the format *original_name_of_msg_file_m.h*, and the object files are created in the same format.

The headers of these message files come with getters and setters for each variable you've included in your original *.msg* file. Keep in mind that **OMNeT++ doesn't use strings, but const char***. This means that setting a value for a string in your message file has to be done by converting it to a `const char*` format, which can be done by using the `c_str()` function.

3.1.3.5 Initialization (.ini) file

Finally, we have to add to our *.ini* file. This file contains the name of the network we wish to use (defined in our .ned file), how long we would like the simulation to run, the metrics we wish to collect, etc. Our *.ned* file is as follows:

```
[General]
network = DataReplicationNetwork    # Name of the network defined in the NED file
sim-time-limit = 2000s              # Simulation time limit, you can adjust as needed

*.seed = 42                         # for random number generation

check-signals = true                # used for recording metrics
```

With this, one should easily be able to reproduce and possibly extend our simulation project.

The project can be run by initially building the entire folder, and then running the simulation. The simulation page is as follows:

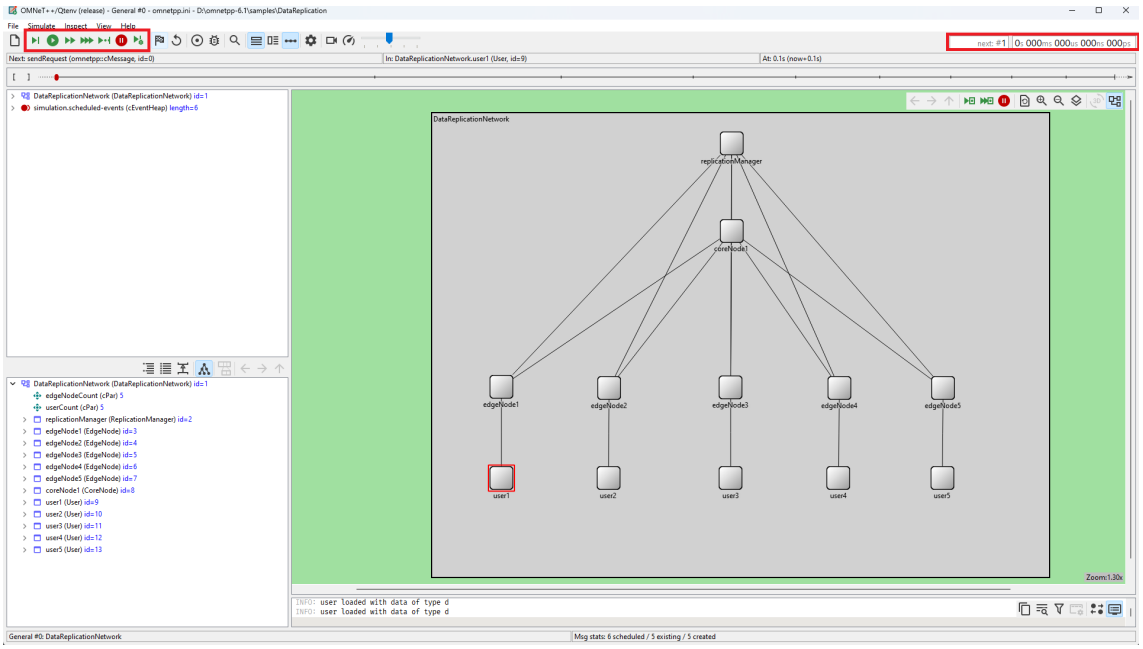

Figure 3.1: OMNeT++ Simulation

As shown in Figure 3.1, The buttons on the top left corner are used to run the simulation, and the top right shows the simulation time.

3.2 Architectural Diagrams

The architectural diagrams used for each configuration are outlined below, with the connections made as mentioned in the previous sections.

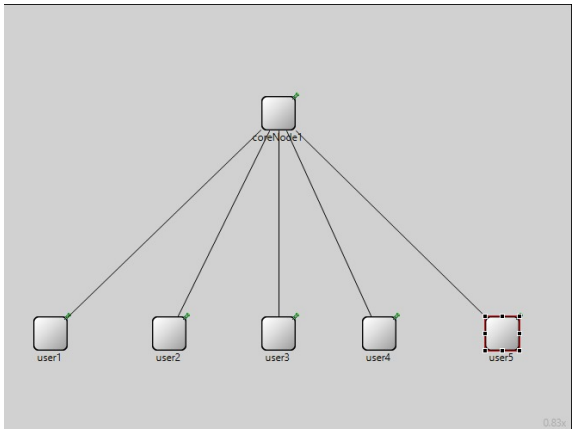

Figure 3.2: 5 Users Without Edge Nodes

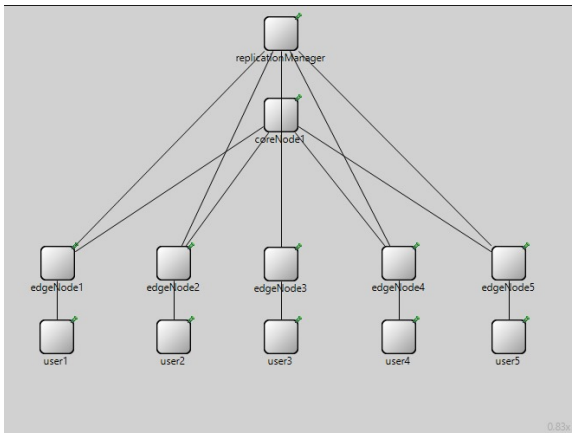

Figure 3.3: 5 Users With 5 Edge Nodes

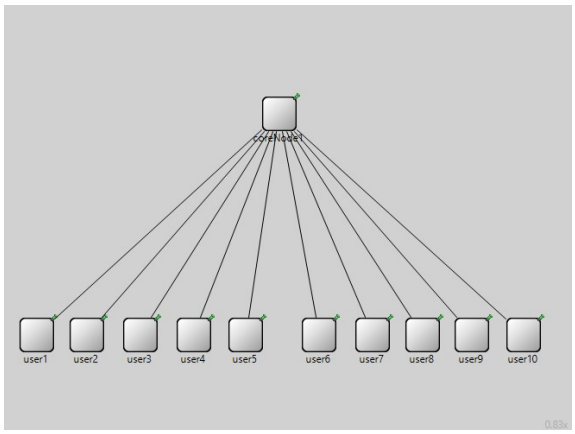

Figure 3.4: 10 Users Without Edge Nodes

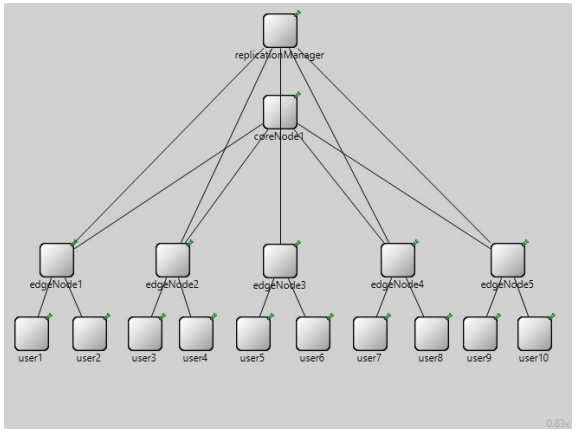

Figure 3.5: 10 Users With 5 Edge Nodes

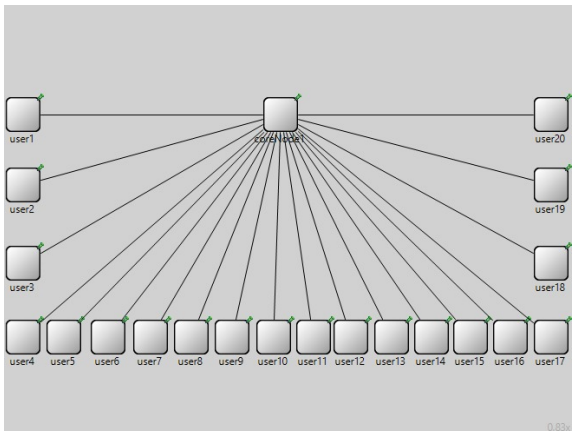

Figure 3.6: 20 Users Without Edge Nodes

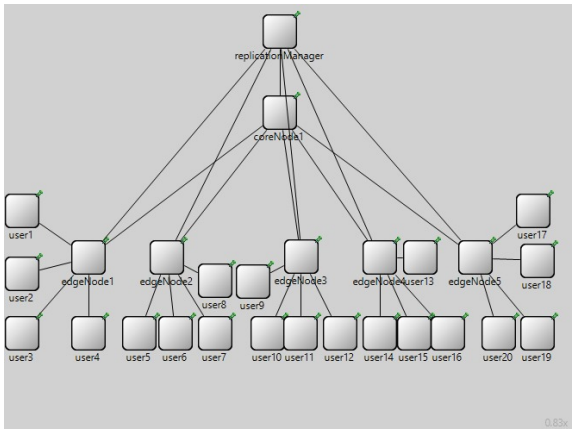

Figure 3.7: 20 Users With 5 Edge Nodes

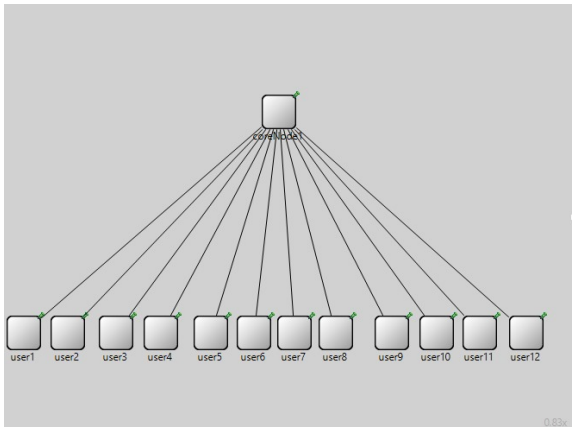

Figure 3.8: 12 Users Without Edge Nodes

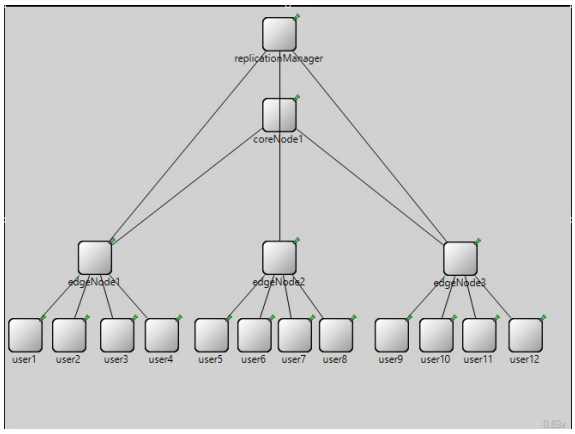

Figure 3.9: 12 Users With 3 Edge Nodes

3.3 Fault Tolerance

Here, we simulate an Edge Node failure, as shown below. In this scenario, the user just connects to the next nearest edge node, where their requests can then be serviced through that particular edge node, greatly improving the reliability of the system.

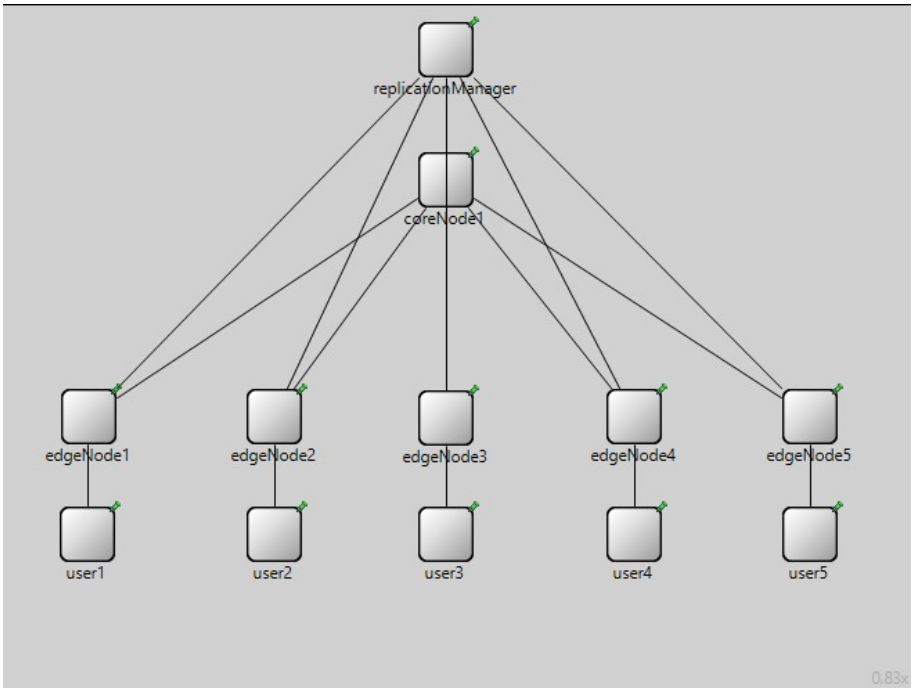

Figure 3.10: Before Simulating Edge Node Failure

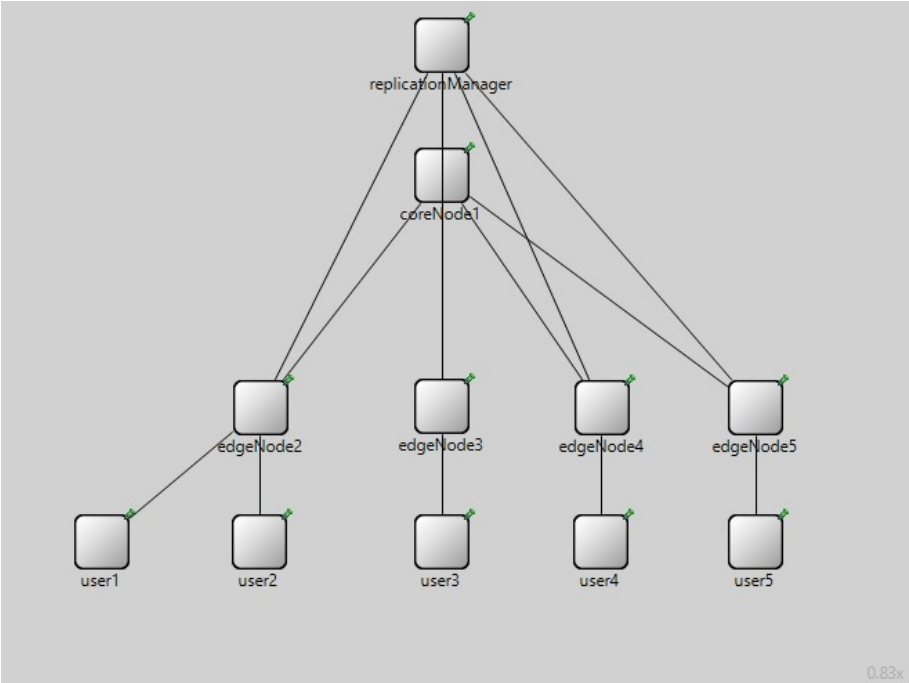

Figure 3.11: After Simulated Edge Node Failure; Fault Tolerance

4

Results

4.1 5 Users, 5 Edge Nodes

For a configuration with 5 edge nodes and 5 users, these are the results, run over 50 simulation seconds.

Name	Count	Mean	StdDev
coreTimeTaken:vector	249	~ 259.689	~ 150.89
userRequestedData:vector	50	25.5	~ 14.577 4
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 2.113 41	~ 0.873 305
userRequestedData:vector	50	25.5	~ 14.577 4
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 2.098 4	~ 0.770 731
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 2.093 9	~ 0.791 803
userRequestedData:vector	52	26.5	~ 15.154 8
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 2.283 39	~ 2.302 91
userRequestedData:vector	46	23.5	~ 13.422 6
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 3.211 94	~ 6.724 06

Figure 4.1: 5 Users Without Edge Nodes

userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 1.664 57	~ 0.941 598
userRequestedData:vector	49	25	~ 14.288 7
userDataReceived:vector	47	24	~ 13.711 3
userRequestDelay:vector	47	~ 2.763 6	~ 6.157 45
userRequestedData:vector	48	24.5	14
userDataReceived:vector	46	23.5	~ 13.422 6
userRequestDelay:vector	46	~ 1.793 97	~ 0.790 042
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 1.987 68	~ 0.749 854
userRequestedData:vector	54	27.5	~ 15.732 1
userDataReceived:vector	52	26.5	~ 15.154 8
userRequestDelay:vector	52	~ 3.027 05	~ 6.698 26

Figure 4.2: 5 Users With 5 Edge Nodes

The highlighted rows are the rows that show the latency, Figure 4.1 shows the latency without edge nodes, and Figure 4.2 shows the latency with edge nodes. The following results follow a similar structure. The second column is the number of packets sent, the third column is the latency (in seconds), and the fourth is the standard deviation.

In this particular scenario, we see a decrease in latency from an average of 2.360208 seconds to an average of 2.247374 seconds showing a decrease of 112.834ms, with a percentage decrease of 4.78% using our system.

4.2 10 Users, 5 Edge Nodes

For a configuration with 5 edge nodes and 10 users, these are the results, run over 50 simulation seconds.

userRequestedData:vector	59	30	~ 17.175 6
userDataReceived:vector	56	28.5	~ 16.309 5
userRequestDelay:vector	56	~ 4.439 84	~ 6.649 99
userRequestedData:vector	48	24.5	14
userDataReceived:vector	46	23.5	~ 13.422 6
userRequestDelay:vector	46	~ 3.723 2	~ 5.923 53
userRequestedData:vector	46	23.5	~ 13.422 6
userDataReceived:vector	42	21.5	~ 12.267 8
userRequestDelay:vector	42	~ 4.185 52	~ 7.504 63
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 2.811 14	~ 1.236 19
userRequestedData:vector	48	24.5	14
userDataReceived:vector	46	23.5	~ 13.422 6
userRequestDelay:vector	46	~ 3.605 87	~ 4.540 93
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	44	22.5	~ 12.845 2
userRequestDelay:vector	44	~ 3.715 68	~ 4.330 66
userRequestedData:vector	56	28.5	~ 16.309 5
userDataReceived:vector	55	28	~ 16.020 8
userRequestDelay:vector	55	~ 4.281 96	~ 7.335 36
userRequestedData:vector	50	25.5	~ 14.577 4
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 2.765 75	~ 1.305 94
userRequestedData:vector	56	28.5	~ 16.309 5
userDataReceived:vector	53	27	~ 15.443 4
userRequestDelay:vector	53	~ 4.749 81	~ 8.517 01
userRequestedData:vector	48	24.5	14
userDataReceived:vector	46	23.5	~ 13.422 6
userRequestDelay:vector	46	~ 3.359 04	~ 1.062 97

Figure 4.3: 10 Users Without Edge Nodes

userRequestedData:vector	106	53.5	~ 30.7436
userDataReceived:vector	104	52.5	~ 30.1662
userRequestDelay:vector	104	~ 2.27228	~ 8.94986
userRequestedData:vector	89	45	~ 25.836
userDataReceived:vector	88	44.5	~ 25.5473
userRequestDelay:vector	88	~ 2.45266	~ 7.74342
userRequestedData:vector	108	54.5	~ 31.3209
userDataReceived:vector	106	53.5	~ 30.7436
userRequestDelay:vector	106	~ 1.82226	~ 2.83438
userRequestedData:vector	101	51	~ 29.3002
userDataReceived:vector	98	49.5	~ 28.4341
userRequestDelay:vector	98	~ 2.61676	~ 6.54478
userRequestedData:vector	107	54	~ 31.0322
userDataReceived:vector	106	53.5	~ 30.7436
userRequestDelay:vector	106	~ 1.62733	~ 0.618062
userRequestedData:vector	96	48.5	~ 27.8568
userDataReceived:vector	94	47.5	~ 27.2794
userRequestDelay:vector	94	~ 3.50981	~ 11.6339
userRequestedData:vector	90	45.5	~ 26.1247
userDataReceived:vector	89	45	~ 25.836
userRequestDelay:vector	89	~ 1.74579	~ 3.20244
userRequestedData:vector	108	54.5	~ 31.3209
userDataReceived:vector	106	53.5	~ 30.7436
userRequestDelay:vector	106	~ 3.27274	~ 10.2614
userRequestedData:vector	103	52	~ 29.8775
userDataReceived:vector	119	60	~ 34.4964
userRequestDelay:vector	119	~ 8.26697	~ 19.222
userRequestedData:vector	100	50.5	~ 29.0115
userDataReceived:vector	83	42	~ 24.1039
userRequestDelay:vector	83	~ 1.94332	~ 4.61695

Figure 4.4: 10 Users With 5 Edge Nodes

Similarly, in this particular scenario, we see a decrease in latency from an average of 3.763781 seconds to an average of 2.952992 seconds showing a **decrease of 810.789ms**, with a percentage decrease of 21.55% using our system.

4.3 20 Users, 5 Edge Nodes

For a configuration with 5 edge nodes and 20 users, these are the results, run over 50 simulation seconds.

userRequestedData:vector	53	27	~ 15.443 4
userDataReceived:vector	50	25.5	~ 14.577 4
userRequestDelay:vector	50	~ 3.051 3	~ 2.289 33
userRequestedData:vector	55	28	~ 16.020 8
userDataReceived:vector	50	25.5	~ 14.577 4
userRequestDelay:vector	50	~ 4.393 74	~ 6.590 15
userRequestedData:vector	48	24.5	14
userDataReceived:vector	46	23.5	~ 13.422 6
userRequestDelay:vector	46	~ 3.170 24	~ 1.821 05
userRequestedData:vector	49	25	~ 14.288 7
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 3.268 89	~ 2.266 35
userRequestedData:vector	48	24.5	14
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 3.280 38	~ 1.280 2
userRequestedData:vector	48	24.5	14
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 2.946 47	~ 1.262 39
userRequestedData:vector	55	28	~ 16.020 8
userDataReceived:vector	50	25.5	~ 14.577 4
userRequestDelay:vector	50	~ 3.111 48	~ 3.829 19
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 4.234 64	~ 7.458 83
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 3.293 8	~ 3.522 05
userRequestedData:vector	56	28.5	~ 16.309 5
userDataReceived:vector	52	26.5	~ 15.154 8
userRequestDelay:vector	52	~ 3.789 27	~ 4.513 91
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 3.732 87	~ 2.983 6
userRequestedData:vector	44	22.5	~ 12.845 2
userDataReceived:vector	41	21	~ 11.979 1
userRequestDelay:vector	41	~ 4.395 88	~ 7.350 7
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 2.784 27	~ 1.433

Figure 4.5: 20 Users Without Edge Nodes

userRequestedData:vector	52	26.5	~ 15.154 8
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 3.341 39	~ 2.842 12
userRequestedData:vector	53	27	~ 15.443 4
userDataReceived:vector	50	25.5	~ 14.577 4
userRequestDelay:vector	50	~ 3.437 98	~ 2.735 34
userRequestedData:vector	41	21	~ 11.979 1
userDataReceived:vector	37	19	~ 10.824 4
userRequestDelay:vector	37	~ 3.096 51	~ 1.384 83
userRequestedData:vector	55	28	~ 16.020 8
userDataReceived:vector	51	26	~ 14.866 1
userRequestDelay:vector	51	~ 3.861 06	~ 5.336 86
userRequestedData:vector	56	28.5	~ 16.309 5
userDataReceived:vector	53	27	~ 15.443 4
userRequestDelay:vector	53	~ 3.109 36	~ 1.975 23
userRequestedData:vector	50	25.5	~ 14.577 4
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 3.409 44	~ 2.999 1
userRequestedData:vector	53	27	~ 15.443 4
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 3.087 96	~ 1.228 88

Figure 4.6: 20 Users Without Edge Nodes, continued.

userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 2.224 59	~ 0.956 925
userRequestedData:vector	55	28	~ 16.020 8
userDataReceived:vector	53	27	~ 15.443 4
userRequestDelay:vector	53	~ 3.018 4	~ 5.966 07
userRequestedData:vector	48	24.5	14
userDataReceived:vector	44	22.5	~ 12.845 2
userRequestDelay:vector	44	~ 2.522 07	~ 5.172 56
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 3.319 14	~ 4.714 25
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	44	22.5	~ 12.845 2
userRequestDelay:vector	44	~ 2.349 58	~ 0.888 454
userRequestedData:vector	45	23	~ 13.133 9
userDataReceived:vector	43	22	~ 12.556 5
userRequestDelay:vector	43	~ 2.229 62	~ 1.097 98
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 2.166 31	~ 0.829 552
userRequestedData:vector	52	26.5	~ 15.154 8
userDataReceived:vector	48	24.5	14
userRequestDelay:vector	48	~ 2.237 52	~ 0.987 615
userRequestedData:vector	59	30	~ 17.175 6
userDataReceived:vector	56	28.5	~ 16.309 5
userRequestDelay:vector	56	~ 2.447 37	~ 0.831 178
userRequestedData:vector	55	28	~ 16.020 8
userDataReceived:vector	53	27	~ 15.443 4
userRequestDelay:vector	53	~ 2.587 03	~ 4.582 59

Figure 4.7: 20 Users With 5 Edge Nodes

userRequestedData:vector	54	27.5	~ 15.732 1
userDataReceived:vector	51	26	~ 14.866 1
userRequestDelay:vector	51	~ 2.532 14	~ 2.391 39
userRequestedData:vector	52	26.5	~ 15.154 8
userDataReceived:vector	51	26	~ 14.866 1
userRequestDelay:vector	51	~ 3.019	~ 6.588 29
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	44	22.5	~ 12.845 2
userRequestDelay:vector	44	~ 1.991 64	~ 0.944 797
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	47	24	~ 13.711 3
userRequestDelay:vector	47	~ 2.997 22	~ 5.852 81
userRequestedData:vector	57	29	~ 16.598 2
userDataReceived:vector	55	28	~ 16.020 8
userRequestDelay:vector	55	~ 2.134 41	~ 1.650 88
userRequestedData:vector	48	24.5	14
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 2.393 37	~ 1.552 64
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	46	23.5	~ 13.422 6
userRequestDelay:vector	46	~ 2.200 59	~ 0.964 605
userRequestedData:vector	47	24	~ 13.711 3
userDataReceived:vector	45	23	~ 13.133 9
userRequestDelay:vector	45	~ 1.847 12	~ 0.945 559
userRequestedData:vector	52	26.5	~ 15.154 8
userDataReceived:vector	49	25	~ 14.288 7
userRequestDelay:vector	49	~ 2.524 69	~ 3.616 65
userRequestedData:vector	51	26	~ 14.866 1
userDataReceived:vector	50	25.5	~ 14.577 4
userRequestDelay:vector	50	~ 1.969 67	~ 0.996 883

Figure 4.8: 20 Users With 5 Edge Nodes, continued.

In this particular scenario, we see a decrease in latency from an average of 3.4398465 seconds to an average of 2.435574 seconds showing **a decrease of 1004.2725ms**, with a percentage decrease of 29.21% using our system.

4.4 12 Users, 3 Edge Nodes

For another configuration with 3 edge nodes and 12 users, run over 50 simulation seconds, the results are as follows.

userRequestedData:vector	57	29	~ 16.598 2	
userDataReceived:vector	55	28	~ 16.020 8	
userRequestDelay:vector	55	~ 4.126 29	~ 7.310 4	
userRequestedData:vector	47	24	~ 13.711 3	
userDataReceived:vector	44	22.5	~ 12.845 2	
userRequestDelay:vector	44	~ 3.718 82	~ 5.278 29	
userRequestedData:vector	53	27	~ 15.443 4	
userDataReceived:vector	49	25	~ 14.288 7	
userRequestDelay:vector	49	~ 3.721 12	~ 4.163 84	
userRequestedData:vector	45	23	~ 13.133 9	
userDataReceived:vector	44	22.5	~ 12.845 2	
userRequestDelay:vector	44	~ 3.444 36	~ 2.808 35	
userRequestedData:vector	48	24.5	14	
userDataReceived:vector	47	24	~ 13.711 3	
userRequestDelay:vector	47	~ 3.022 96	~ 1.209 82	
userRequestedData:vector	48	24.5	14	
userDataReceived:vector	45	23	~ 13.133 9	
userRequestDelay:vector	45	~ 3.467 62	~ 4.113 77	

Figure 4.9: 12 Users Without Edge Nodes

userRequestedData:vector	54	27.5	~ 15.732 1	
userDataReceived:vector	50	25.5	~ 14.577 4	
userRequestDelay:vector	50	~ 4.007 68	~ 6.396 94	
userRequestedData:vector	53	27	~ 15.443 4	
userDataReceived:vector	51	26	~ 14.866 1	
userRequestDelay:vector	51	~ 3.894 49	~ 5.271 82	
userRequestedData:vector	45	23	~ 13.133 9	
userDataReceived:vector	43	22	~ 12.556 5	
userRequestDelay:vector	43	~ 3.773 35	~ 4.792 11	
userRequestedData:vector	53	27	~ 15.443 4	
userDataReceived:vector	47	24	~ 13.711 3	
userRequestDelay:vector	47	~ 2.611 74	~ 1.502 23	
userRequestedData:vector	47	24	~ 13.711 3	
userDataReceived:vector	44	22.5	~ 12.845 2	
userRequestDelay:vector	44	~ 2.970 18	~ 1.185 18	
userRequestedData:vector	58	29.5	~ 16.886 9	
userDataReceived:vector	53	27	~ 15.443 4	
userRequestDelay:vector	53	~ 5.026 43	~ 7.878 19	

Figure 4.10: 12 Users Without Edge Nodes, continued.

userRequestedData:vector	48	24.5	14	
userDataReceived:vector	46	23.5	~ 13.422 6	
userRequestDelay:vector	46	~ 2.515 87	~ 2.593 36	
userRequestedData:vector	50	25.5	~ 14.577 4	
userDataReceived:vector	49	25	~ 14.288 7	
userRequestDelay:vector	49	~ 2.146 33	~ 0.971 932	
userRequestedData:vector	51	26	~ 14.866 1	
userDataReceived:vector	48	24.5	14	
userRequestDelay:vector	48	~ 1.922 02	~ 1.053 85	
userRequestedData:vector	49	25	~ 14.288 7	
userDataReceived:vector	48	24.5	14	
userRequestDelay:vector	48	~ 3.018 02	~ 6.363 94	
userRequestedData:vector	48	24.5	14	
userDataReceived:vector	46	23.5	~ 13.422 6	
userRequestDelay:vector	46	~ 1.911 66	~ 0.789 103	
userRequestedData:vector	55	28	~ 16.020 8	
userDataReceived:vector	53	27	~ 15.443 4	
userRequestDelay:vector	53	~ 3.327 47	~ 5.846 53	

Figure 4.11: 12 Users With 3 Edge Nodes

userRequestedData:vector	48	24.5	14	
userDataReceived:vector	46	23.5	~ 13.422 6	
userRequestDelay:vector	46	~ 2.559 96	~ 3.618 2	
userRequestedData:vector	51	26	~ 14.866 1	
userDataReceived:vector	50	25.5	~ 14.577 4	
userRequestDelay:vector	50	~ 1.949 81	~ 0.914 054	
userRequestedData:vector	49	25	~ 14.288 7	
userDataReceived:vector	48	24.5	14	
userRequestDelay:vector	48	~ 2.096 39	~ 0.899 732	
userRequestedData:vector	57	29	~ 16.598 2	
userDataReceived:vector	54	27.5	~ 15.732 1	
userRequestDelay:vector	54	~ 2.172 59	~ 1.023 47	
userRequestedData:vector	50	25.5	~ 14.577 4	
userDataReceived:vector	49	25	~ 14.288 7	
userRequestDelay:vector	49	~ 2.535 94	~ 2.225 07	
userRequestedData:vector	52	26.5	~ 15.154 8	
userDataReceived:vector	48	24.5	14	
userRequestDelay:vector	48	~ 2.057 22	~ 0.851 339	

Figure 4.12: 12 Users With 3 Edge Nodes, continued.

In this particular scenario, we see a decrease in latency from an average of 3.64875333 seconds to 2.35110667 seconds showing a **decrease of 1297.64666ms**, with a per-

centage decrease of 35.56% using our system.

The trend we observe here is that as the number of users increase, the latency while using edge nodes decreases proportionally. This system can be simulated for much longer than the time we used to obtain greater insights into the workings of our code.

4.5 Cache Hit/Miss Ratio

This is the initial cache hit/miss ratio for 5 users, when the users are randomly accessing data, run just for 50 simulation seconds. The results are about what you'd expect, around 26.87%; it is essentially just random chance.

edgeRequestCore:vector	35	18	~ 10.247
edgeHaveData:vector	12	6.5	~ 3.605 55
edgeHitMissRatio:vector	82	~ 0.249 918	~ 0.105 914
edgeRequestCore:vector	38	19.5	~ 11.113 1
edgeHaveData:vector	11	6	~ 3.316 62
edgeHitMissRatio:vector	86	~ 0.183 197	~ 0.071 407 6
edgeRequestCore:vector	37	19	~ 10.824 4
edgeHaveData:vector	11	6	~ 3.316 62
edgeHitMissRatio:vector	85	~ 0.253 651	~ 0.102 741
edgeRequestCore:vector	42	21.5	~ 12.267 8
edgeHaveData:vector	9	5	~ 2.738 61
edgeHitMissRatio:vector	93	~ 0.141 17	~ 0.086 346 1
edgeRequestCore:vector	45	23	~ 13.133 9
edgeHaveData:vector	9	5	~ 2.738 61
edgeHitMissRatio:vector	98	~ 0.108 066	~ 0.055 357 4

Figure 4.13: Initial Cache Hit/Miss Ratio

After implementing a proactive/predictive approach, where the users are more likely to access certain 'popular' data items (given by a popularity index for the data, where the higher the index the more often they are accessed, which is more akin to a real-world system), the cache hit/miss ratio greatly improved, and continues to improve the longer the simulation is run; the results are as follows.

edgeRequestCore:vector	32	16.5	~ 9.380 83
edgeHaveData:vector	17	9	~ 5.049 75
edgeHitMissRatio:vector	81	~ 0.209 402	~ 0.073 353 8
edgeRequestCore:vector	29	15	~ 8.514 69
edgeHaveData:vector	23	12	~ 6.782 33
edgeHitMissRatio:vector	81	~ 0.208 758	~ 0.152 07
edgeRequestCore:vector	31	16	~ 9.092 12
edgeHaveData:vector	25	13	~ 7.359 8
edgeHitMissRatio:vector	86	~ 0.332 443	~ 0.084 685 6
edgeRequestCore:vector	29	15	~ 8.514 69
edgeHaveData:vector	19	10	~ 5.627 31
edgeHitMissRatio:vector	77	~ 0.283 547	~ 0.082 735 3
edgeRequestCore:vector	24	12.5	~ 7.071 07
edgeHaveData:vector	24	12.5	~ 7.071 07
edgeHitMissRatio:vector	72	~ 0.233 665	~ 0.178 594

Figure 4.14: Improved Cache Hit/Miss Ratio; 50 simulation seconds

edgeRequestCore:vector	51	26	~ 14.866 1
edgeHaveData:vector	46	23.5	~ 13.422 6
edgeHitMissRatio:vector	148	~ 0.312 829	~ 0.128 043
edgeRequestCore:vector	43	22	~ 12.556 5
edgeHaveData:vector	56	28.5	~ 16.309 5
edgeHitMissRatio:vector	142	~ 0.335 528	~ 0.187 98
edgeRequestCore:vector	50	25.5	~ 14.577 4
edgeHaveData:vector	52	26.5	~ 15.154 8
edgeHitMissRatio:vector	151	~ 0.393 634	~ 0.095 849 7
edgeRequestCore:vector	50	25.5	~ 14.577 4
edgeHaveData:vector	52	26.5	~ 15.154 8
edgeHitMissRatio:vector	152	~ 0.395 053	~ 0.129 917
edgeRequestCore:vector	45	23	~ 13.133 9
edgeHaveData:vector	58	29.5	~ 16.886 9
edgeHitMissRatio:vector	148	~ 0.387 554	~ 0.195 647

Figure 4.15: Improved Cache Hit/Miss Ratio; 100 simulation seconds

edgeRequestCore:vector	94	47.5	~ 27.279 4
edgeHaveData:vector	110	55.5	~ 31.898 3
edgeHitMissRatio:vector	297	~ 0.412 185	~ 0.134 9
edgeRequestCore:vector	81	41	~ 23.526 6
edgeHaveData:vector	116	58.5	~ 33.630 3
edgeHitMissRatio:vector	277	~ 0.460 952	~ 0.186 278
edgeRequestCore:vector	97	49	~ 28.145 5
edgeHaveData:vector	111	56	~ 32.187
edgeHitMissRatio:vector	305	~ 0.455 111	~ 0.091 118
edgeRequestCore:vector	87	44	~ 25.258 7
edgeHaveData:vector	132	66.5	~ 38.249 2
edgeHitMissRatio:vector	305	~ 0.484 167	~ 0.129 499
edgeRequestCore:vector	82	41.5	~ 23.815 3
edgeHaveData:vector	118	59.5	~ 34.207 7
edgeHitMissRatio:vector	282	~ 0.472 279	~ 0.167 494

Figure 4.16: Improved Cache Hit/Miss Ratio; 200 simulation seconds

For calculating our Hit/Miss Ratio, we use the number of times the Edge Node has the requested data, and the number of times it has to request it from the Core Node. We do not use the third metric (edgeHitMissRatio) due to the way OMNeT++ handles division and the aggregation of values while displaying the metrics. More accurate data can be obtained by converting this data into a JSON format, as this contains the metrics in much greater detail.

With this method, after 50 simulation seconds, the Cache Hit/Miss Ratio is 42.63%, 52.45% after 100 simulation seconds, and 57.08% after 200 simulation seconds, showing a clear improvement from the random access scenario and showing improvement over time.

5
Contributions

Both team members contributed equally, right from the inception of the project idea to the final system. Both of us contributed to:

- Using and setting up OMNeT++, and all the required files.
- Coming up with the replication algorithm for the replication manager, creating and coding the required data items and C++ files for the execution of the project.
- Creating the general architecture of the system for simulation.
- Running simulations for different configurations and collecting metrics over those configurations.
- Extensive testing and bug fixing.
- Finishing the presentations and the final project report.

6
Conclusions

The implementation of this Latency-Aware Data Replication System helped us understand the complexities associated with simulating a distributed system. One major lesson learnt was the significance of optimizing replication across multiple nodes to reduce latency as much as possible. Implementing the Edge Nodes, the system showed a clear reduction in the latency associated with servicing user data requests.

The replication mechanism successfully minimized latency for users by a significant amount by placing popular data at the Edge Nodes at the site of popular demand. The division of labor among the Edge Nodes, Core Nodes, and the Replication Manager worked well in distributing data efficiently.