Department of Electrical Engineering,
Syed Babar Ali School of Science and Engineering,
Lahore University of Management Sciences, Lahore, Pakistan

CALL CENTER SIMULATION IN NS-3 AND CALL CENTER APPLICATION IN NODE.JS

By

Ammarah Azmat Bajwa **Roll No: 18100243**

Hira Tariq **Roll No: 18100215**

Zarmeen Khan **Roll No: 18100273**

Muhammad Shamaas **Roll No: 18100217**

A report submitted in partial fulfillment of the requirements for the degree of

BS Electrical Engineering

Syed Babar Ali School of Science and Engineering,

Lahore University of Management Sciences

Under the supervision of

Dr. Tariq Mahmood Jadoon

Designation: Associate Professor

Email: jadoon@lums.edu.pk

Dr. Ijaz Haider Naqvi

Designation: Associate Professor

Email: ijaznaqvi@lums.edu.pk

Date: 22 May 2018

Advisor (Dr. Tariq Mahmood Jadoon)

_____

jadoon@lums.edu.pk

Co advisor (Dr. Ijaz Haider Naqvi)

_____

ijaznaqvi@lums.edu.pk

Contents

List of figures vi
List of tables viii
Acknowledgments ix
Chapter 1 5
Problem Statement 5
Chapter 2 6
Back ground and related work 6
2.1 Background 6
2.1.1 Continuous Time Markov Chain 7
2.1.2 M/M/1 Queue 9
2.1.3 M/M/∞ Queue 10
2.1.4 M/M/1/K Queue 11
2.2: Related Work 12
2.2.1 Erlang C 13
2.2.2 Erlang X 14
Chapter 3 15
Design Methodology And Tools 15
3.1: System level design 15
3.2: Design Methodology 16
3.3: Tools 17
3.3.1: NS-3 17
3.3.1.1: Initial System Model: Erlang C 17
3.3.1.2: Improved System Model: Erlang X 22
3.3.2: Node.js 27
3.3.2.1: Hypertext Preprocessor/ Active Server Pages protocols 29
3.3.2.2: Secured Network 30
3.3.2.2.1: Secure Hyper Text Transfer Protocol 31
3.3.2.2.2: Transport Layer Security 32
3.3.2.2.3: Digital Certificates 34
3.3.2.2.4: The Public/ Private Key 35
3.3.2.2.5: Perfect Forward Secrecy 36
3.3.2.3: Javascript Session Establishment Protocol 37
3.3.2.4: WebRTC 38
3.3.2.4.1: Standards and protocols 38
3.3.2.4.2: WebRTC Support Summary 38
3.3.2.4.3: WebRTC Applications 39
3.3.2.4.4: Media Stream 40
3.3.2.4.5: RTC Peer Connection 40
3.3.2.4.6: Interactive Connectivity Establishment 41
3.3.2.4.7: RTC Data Channel 42
3.3.2.4.8: Signaling 42

3.3.2.5: Global Functionality of Call Center 43
3.3.2.6: Server Code 44
3.3.2.7: Call Center Home Web Page 45
3.3.2.8: Caller Web Page 46
3.3.2.9: Agent Web Page 47
3.3.2.10: Additional Features 48
Chapter 4 49
Design, simulation and implementation results 49
4.1: Finalized System Model: Real Call Center 49
4.2: Factors Influencing the Choice of Final Design 50
4.3: Final System Diagram 53
4.4.1: Caller Node Application 58
4.4.2: Distributor Node Application 59
4.4.3: Agent Node Application 60
4.5: Design Results 61
4.5.1: Erlang C Block Diagram 61
4.5.2: Erlang X Block Diagram 61
4.5.3: Real Call Center Block Diagram 62
4.6: Simulation Results 63
4.6.1: Erlang C 63
4.6.2: Erlang X 64
4.6.3: Real Call Center 65
4.6.4: Data Analysis 71
4.6.5: Various Routing Schemes 73
4.7: Implementation Results 74
4.7.1: Node.js Call Center 74
4.8: Discussion 76
Chapter 5 77
Cost Analysis 77
Chapter 6 78
Societal Relevance 78
Chapter 7 79
Environmental impact and sustainability 79
Chapter 8 80
Conclusion and future recommendations 80
References 81
Appendix 83
NS-3 Simulation 83
Online Erlang C Calculator [2] 84
Online Erlang X Calculator [3] 84
NS-3 Code 86
Distributor Queue 87
Caller Application 88

Receiver Application 89
Distributor Application 90
Agent Application 91
Main 92
Node.js Code 94
App.js 94
Caller.html 96
Agent.html 98
CallCenter.html 100

LIST OF FIGURES

Number Page
Figure 1: Continuous Time Markov Chain 7
Figure 2: M/M/1 Queue Markov Chain 9
Figure 3: M/M/∞ Queue Markov Chain 10
Figure 4: M/M/1/K Queue Markov Chain 11
Figure 5: Erlang C Formula for Average Queue Waiting Time versus Number of Agents [33] 13
Figure 6: Real Call Center Model 15
Figure 7: Erlang C Caller Node Application 19
Figure 8: Erlang C Distributor Node Application 20
Figure 9: Erlang C Agent Node Application 21
Figure 10: Erlang X Caller Node Application 24
Figure 11: Erlang X Distributor Node Application 25
Figure 12: Erlang X Agent Node Application 26
Figure 13: Node.js Call Center Block Diagram 27
Figure 14: Node.js Event Loop [36] 29
Figure 15: TLS Handhsake betwwen Server and Client [35] 33
Figure 16: Privacy Enhanced Mail Certificates, Public Key Cryptography Standard #12 file, Personal Information Exchange Certificate and Private File for HTTPS Encryption 34
Figure 17: TLS Key Exchanges [35] 35
Figure 18: Perfect Forward Secrecy [35] 36
Figure 19: JSEP Architecture [34] 37
Figure 20: WebRTC Architecture [34] 39
Figure 21: Framework for finding ICE Candidates [34] 41
Figure 22: Call Initiation Chain [34] 43
Figure 23: Call Center Home Web Page 45
Figure 24: Caller Web Page 46
Figure 25: Agent Web Page 47
Figure 26: Call Center Records 49
Figure 27: Distribution of Call Durations 51
Figure 28: Distribution of Queue Waiting Times 51
Figure 29: Distribution of VRU Durations 52
Figure 30: Call Center Block Diagram 53
Figure 31: Virtual Response Unit Linked List 54
Figure 32: Callers Queue Linked List 54
Figure 33: Caller Node Application 58

Figure 34: Distributor Node Application 59
Figure 35: Agent Node Application 60
Figure 36: Erlang C Queue 61
Figure 37: Erlang X Queue with Abandonments 61
Figure 38: Real Call Center Block Diagram 62
Figure 39: Virtual Response Unit Linked List 62
Figure 40: Real Call Center Queue 62
Figure 41: Erlang C Simulation and Calculator Graph 63
Figure 42: Erlang X Simulation and Calculator Graphs 64
Figure 43: Number of Arivals per Hour 65
Figure 44: Average Queue Waiting Time: Actual Data vs Simulation Results 65
Figure 45: Average Queue Waiting Time: Actual Data vs Simulated Results 66
Figure 46: Average Wait Time: Actual Data vs Simulated Results 66
Figure 47: Real Call Center: Abandonment vs Agents Available and Number of Calls Arriving per Hour for 1 year 67
Figure 48: Real Call Center Simulation with Skill Based Routing: Abandonment vs Agents Available and Number of Calls Arriving per Hour for 1 year 68
Figure 49: Correlation between Real Call Center and NS-3 Simulation 69
Figure 50: Comparison of Routing Schemes: Fastest Server First (Upper Left), Longest Idle First (Upper Right), Lowest Cost Routing (Lower Left), Skill Based Routing (Lower Right) 70
Figure 51: Comparison of Routing Schemes 73
Figure 52: Degradation of Network Performance with increase in Traffic 74
Figure 53: NetAnim Simulation 83
Figure 54: Erlang C Calculator 84
Figure 55: Erlang X Calculator 84
Figure 56: NS-3 Call Center Log File 85
Figure 57: Node.js Call Center Log File 85

LIST OF TABLES

Number Page
Table 1: Profiles of Agents 52
Table 2: Erlang C Simulation and Calculator Results 63
Table 3: Erlang X Simulation and Calculator Results 64
Table 4: Real Call Center Simulation: Calculated Performance Metrics 72
Table 5: Comparison of Routing Schemes 73
Table 6: Comparison of Node.js Call Center Performance with NS-3 Simulation 75

ACKNOWLEDGMENTS

The authors wish to express sincere appreciation to Dr. Tariq Mahmood Jadoon and Dr. Ijaz Haider Naqvi for their supervision during the course of this senior project and in the preparation of this manuscript.

CHAPTER 1

PROBLEM STATEMENT

This project attempts to realistically simulate a Packet Switched Call Center in NS-3 by incorporating call abandonment, heavy call traffic, call drops and skill based routing. In order to find the deficiencies of our Call Center Simulation, we developed a Call Center Web Application in Node.js to study the effects of network congestion, variable propagation delays and transmission speeds on Quality of Service.

CHAPTER 2

BACK GROUND AND RELATED WORK

2.1 BACKGROUND

Call center modeling is a growing research field. They are an example of queuing systems. Queuing models help estimate ways to optimize staffing and maximize performance in a call center. We used NS-3, which is a discrete event network simulator, for our modeling. Calls arrive, wait in a virtual queue and then they are serviced via a routing mechanism. The most common model used at large by both practitioners and academics is the Erlang C model. The model however works under many assumptions mainly: calls arrival follows a Poisson distribution with a known average rate, callers wait infinitely without abandoning the call. Erlang X: an extension of Erlang C is more complex and realistic as it incorporates call abandonment, heavy call traffic; call drops and skill based routing. We attempted to improve Erlang X with additive variables, which makes it close to a realistic call center. We verified it by comparing simulation results against the performance records of a real call center.

In order to model our Call Center, we will need to understand the “Standard Queuing Model” which is based on two assumptions. First, all the agents are assumed to be "homogenous" in their services i.e. having same set of skills and speed to answer the call. Secondly, calls routing is done in such a way that the call is forwarded to the agent who is idle for the longest interval of time. This oversimplification affects the performance of the call center by increasing the average waiting time of the queue. In his paper [13], the author Thomas R. Robbins has highlighted the issues in the oversimplified queuing model and has suggested another routing scheme for the call center environments. Queuing system is implemented in all the call centers. If the agent is busy then the call is placed in the queue and waits there until the agent gets free. Mostly call centers are observed as having a "standard queuing model" which is an oversimplification of the practical call center [12], [5]. However, the performance can be enhanced by introducing some changes in the routing scheme. Agent requires particular skills in handling a certain call efficiently, which depend on their experience as well. More the experience of the agent in handling calls lesser will be the time he will take to service the call, thus the average waiting time in the queue will decrease [13]. Robbins has suggested "experience-based routing" during the peak hours that is when the call arrivals are high then the calls should be delivered to the experienced agents so that they can efficiently manage the load by servicing the calls in lesser time. However, when the arrival rate of calls is small then, according to Robbins, calls should be routed to the non-experienced agents so that they can learn how to service the calls and could gain some experience. This routing scheme not only improves the

"system performance" and the "customer service" but also enhance the learning of the newly hired agents [4].

Call Center is analytically modeled using continuous time Markov Chain that we will discuss briefly now.

2.1.1 CONTINUOUS TIME MARKOV CHAIN

A Call Center can be modeled using a Markov Chain as shown in Figure 1 where the number of the state represents the number of callers waiting in the Queue. It has the unique property of memorylessness where we can make predictions about future of the process based on the present state only. The call arrivals and service are assumed to have a Poisson distribution that expresses the probability of a given number of events occurring in a fixed interval of time if these events occur with a known constant rate and independently of the time since the last event [8].

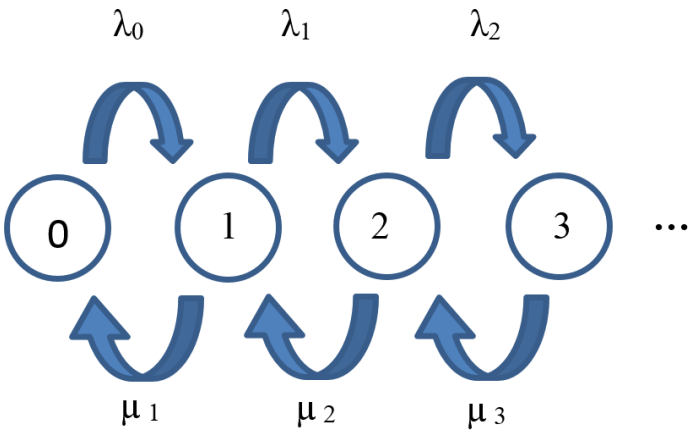


Figure 1: Continuous Time Markov Chain

The Probability of being in state j at time n = π_j (n).

Transition Probability P (X_{n+1}=j | X_n=i) = $p_{i,j}$ (n)

$q_{i,j}(n) = \lim_{\triangle t \to 0} \left(\frac{p_{i,j}(\triangle t)}{\triangle t}\right)$

Arrival rate of Calls in state j = $\lambda_j = q_{j,j+1}(n)$

Service Rate of Calls in state j = $\mu_j = q_{j,j-1}(n)$

$q_{j,j} = -(\lambda_j + \mu_j)$

Time Evolution of $\pi_j(t) = \frac{d\,\pi_j(t)}{dt} = q_{j,j}(t)\,\pi_j(t) + \sum_{k \neq j} q_{k,j}(t)\,\pi_k(t)$

For Stationary State $\pi_j(t) = \pi^*_j$, $\frac{d\,\pi_j(t)}{dt} = 0$ and $\sum_j \pi_j = 1$

The solution of these j differential equations gives the following probability distribution:

Stationary State Probability of state $j = \pi^*_j = \pi^*_0 \prod_{i=0}^{j-1} \frac{\lambda_i}{\mu_{i+1}}$

If $\lambda_j = \lambda, \mu_j = 0$ for all $j, \pi_j(t) = \frac{(\lambda t)^j}{j!} e^{-\lambda t}$ which is the Poisson Distribution Formula

Now we will discuss three different Markov Chains; M/M/1 Queue, M/M/∞ Queue and the M/M/1/K Queue.

2.1.2 M/M/1 QUEUE

An M/M/1 Markov Chain has Markovian Arrivals and Service with single Queue Server as shown in Figure 2. The Call Arrival Rate = λ and the Call Service Rate = μ. Queue can grow infinitely hence; infinite states are possible [8].

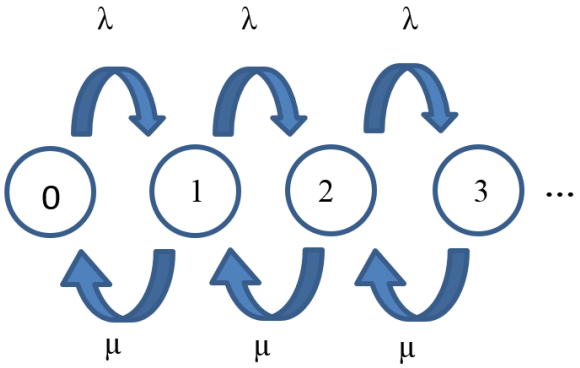


Figure 2: M/M/1 Queue Markov Chain

Stationary State Probability of state j = $\pi^*_j = \left(1 - \left(\frac{\lambda}{\mu}\right)\right)\left(\frac{\lambda}{\mu}\right)^j$

2.1.3 M/M/∞ QUEUE

An M/M/∞ Queue has Markovian Arrivals and Service with separate Queue Server for each Caller as shown in Figure 3. The Call Arrival Rate = λ and the Call Service Rate = nμ where n represents the number of the state. Queue can grow infinitely hence; infinite states are possible [8].

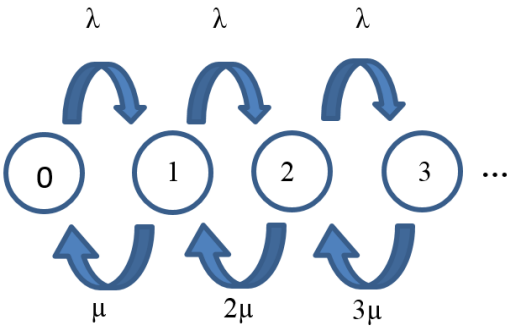


Figure 3: M/M/∞ Queue Markov Chain

Stationary State Probability of state j = $\pi^*_j = \frac{e^{-\frac{\lambda}{\mu}}(\frac{\lambda}{\mu})^j}{j!}$

2.1.4 M/M/1/K QUEUE

An M/M/1/K Markov Queue has Markovian Arrivals and Service with single Queue Server as shown in Figure 4. The Maximum size of Queue = k-1, Call Arrival Rate = λ and Call Service Rate = μ.

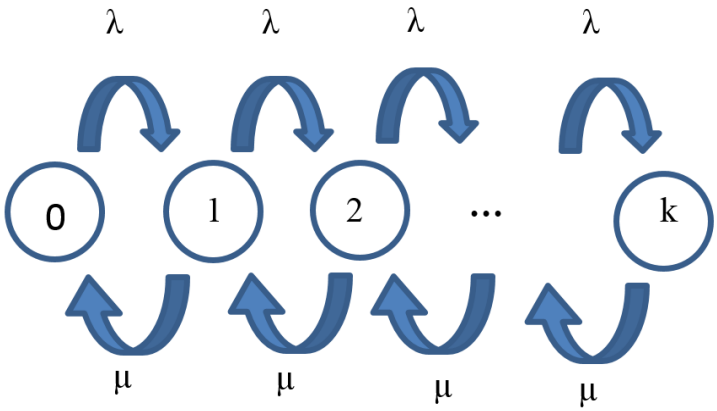


Figure 4: M/M/1/K Queue Markov Chain

Stationary State Probability of state j = $\pi^*_j = \frac{1-\left(\frac{\lambda}{\mu}\right)}{1-\left(\frac{\lambda}{\mu}\right)^{k+1}} \left(\frac{\lambda}{\mu}\right)^j$ where $j \leq k$

2.2: RELATED WORK

Thomas R. Robbins et al. in their paper [13] have compared the queuing models that are widely used in the call center analysis. First is the Erlang C model, many assumptions in the model simplifies it. This model assumes that the agents are "statistically identical" all have the same speed and ability to answer the call. In addition, the calls, which arrive according to the Poisson distribution, are having a known average rate of arrival [8]. Moreover, it assumes that the size of the queue is infinite and calls can wait for any large amount of time inside the queue without abandoning it. It means that the average waiting time could be large enough which is quite unrealistic and can have a drastic impact on the customer service and call center operations. However, in Erlang A model, the extension of the Erlang C model, abandonments are considered. The callers have some finite patience to wait in the queue and if it turns out the caller abandons the queue [7]. "Quality-driven and the Quality and efficiency-driven (QED) regimes", where the call centers are capable enough to handle all the calls and there is no abandonment, are considered in this paper and the performance metrics for both Erlang A and Erlang C in QED regimes are being compared. Erlang A model though give better results but it does not depict the accurate staffing requirement in the cases when the arrival rates have significant uncertainties [4]. In such cases, Erlang C is used in making the staffing decisions. However, in case of the "Efficiency-driven regime" when the arrival rate is large and all agents are occupied in servicing the calls here the Erlang C model simply is collapsed because there are significant abandonments. Erlang A model, which allows the abandonments, is used to predict the performance metrics in this kind of regime [5].

2.2.1 ERLANG C

The Erlang C Formula [7] is given below and the plot is shown in Figure 5 [33]:
λ = Number of calls per unit time
b = Average Call Duration
a = λ * b = Load in Erlang units
s = Number of Agents
For s > a,

Average Waiting Time in Queue = $\frac{C(s,\ a) * b}{(s-a)}$

Where Probability of Enqueue = $C(s,\ a) = \frac{\frac{s \cdot a^s}{s! \cdot (s-a)}}{\sum_{i=0}^{s-1} \frac{a^i}{i!} + \frac{s \cdot a^s}{s! \cdot (s-a)}}$

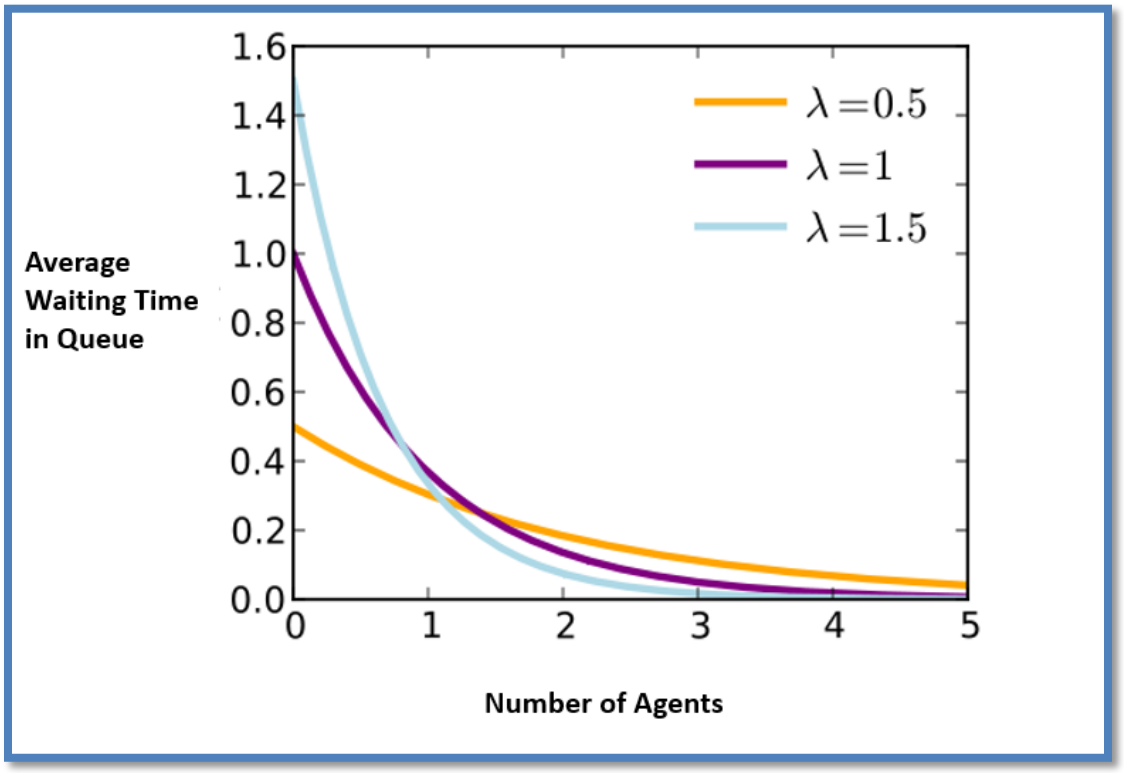


Figure 5: Erlang C Formula for Average Queue Waiting Time versus Number of Agents [33]

2.2.2 ERLANG X

The Erlang X or Erlang A model builds on the Erlang C Model by including abandonments of callers i.e. Callers can abandon Queue after waiting for some time. It is modeled using M/M/M+N Markov Chain [13]. Packets can be dropped if queue is full which models the limited number of telephone lines available. Calls can require multiple skills and Agents can have multiple skills. Specialists have only one skill whereas Generalists can have two or more skills. All Calls have same Call Duration distribution.

The Analytical Erlang X formula [14] is presented below.

Mean Patience of Callers = θ^{-1}, Arrival Rate = λ, Service rate = $n\mu$

Gamma Function $\gamma(x, y) = \int_0^y t^{x-1}e^{-t}dt$, $x \geq 0$, $y \geq 0$

$J = \frac{e^{\lambda/\theta}}{\theta}\left(\frac{\theta}{\lambda}\right)^{\frac{n\mu}{\theta}} \gamma\left(\frac{n\mu}{\theta}, \frac{\lambda}{\theta}\right)$, $\varepsilon = \frac{\sum_{j=0}^{n-1}\left(\frac{1}{j!}\right)\left(\frac{\lambda}{\mu}\right)^j}{\left(\frac{1}{(n-1)!}\right)\left(\frac{\lambda}{\mu}\right)^{n-1}}$, P (Wait > 0) = $\frac{\lambda J}{\varepsilon+\lambda J}(1-\theta)$

CHAPTER 3

DESIGN METHODOLOGY AND TOOLS

3.1: SYSTEM LEVEL DESIGN

The Real Call Center has Abandonments, Call Drops, Dynamic Agents Scheduling, Time varying call arrival rates, heterogeneous agents, Heavy Traffic and Call Retrials as shown in Figure 6, which require much more detailed modeling and calculations.

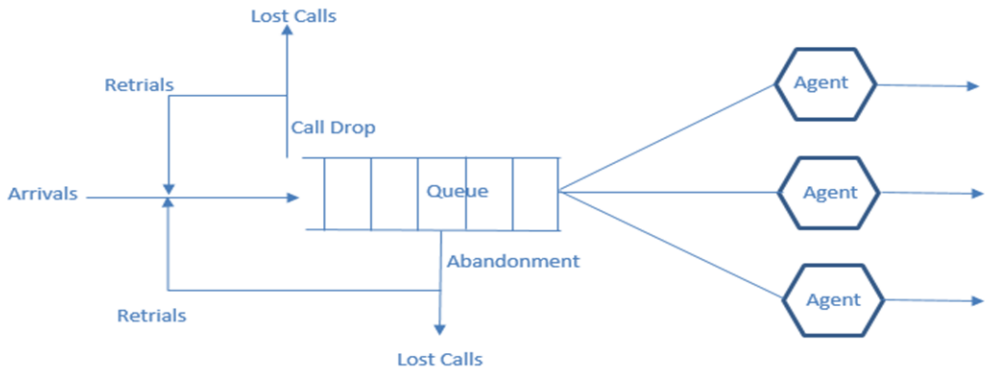


Figure 6: Real Call Center Model

We discussed the various models used for the analytical analysis of the Call Center Queuing Model. Now we will discuss the Simulation Results of our Proposed Model whose foundation is the M/M/N+M/K Queue and which was built to improve the existing Call Center Models. Our goals were:

1. Merge M/M/N and M/M/1/K models to include call drops and multiple Queue Servers.
2. Include Agent Skills and Caller Patience for call abandonments.
3. Incorporate randomness for propagation delays.
4. Perform experiments with simulator and compare results with the industrial models of Erlang C and Erlang X.
5. Iteratively perform simulation optimization to realize actual call center model.

3.2: DESIGN METHODOLOGY

The Call Centre structure comprises of three fundamental entities; the Caller Node, the Agent Node and the Distributor Node. Having built a structure for the Agents, we have further differentiated them based on their skills, no of calls serviced and operator number in our model. The callers have been differentiated based on call times, patience tag, waiting time in the queue and the skill type they require. While a common distributor node (buffer) has also been made which Enqueue arriving packets and Dequeue packets to the agent node.

Erlang C is a simple multi-server queuing system; calls arrive according to a Poisson process at an average rate. We verified Erlang C Formula with experimental results for mean waiting time of callers in Queue using the Ger Koole calculator [10], [11]. Our goal is to verify Erlang X with additive variables, which makes it close to a realistic call center through empirical analysis involving comparison of performance with real call center records [9].
Generally, abandonment rates cannot be estimated directly using the Erlang C model because the model assumes no abandonment occurs. Our model will assume that each caller has a finite willingness to wait, and will abandon the queue or hang up if his or her wait time exceeds his or her patience. We plan to achieve this by assigning each caller a patience counter, which is decremented after each second. Erlang X is modeled with limited available number of telephone lines thus calls will be dropped if the waiting queue is full. In addition to finite queue, we route our distributor in such a way that it can pick a blocking packet from anywhere in the queue and send it for service to prevent queue blockage [10], [11]. Furthermore, the agents in our model will also have multiple skills. There are two categories: Specialists, having only one skill and Generalists, having expertise in more than one skill [7]. We conclude by verifying experimentally that even with complex variables, we can achieve optimization in Erlang X just as that in Erlang C.

3.3: TOOLS

3.3.1: NS-3

We are implementing a Simulation of an Optimized Call Centre Model on NS3 Software, which is a discrete event simulator [6]. We have implemented packet switching in our simulation. This has been done in the software, by allocating a caller node, distributor and Agent Node. As the following is a representation of a more realistic model, so we have implemented Erlang X. It in cooperates queue abandonment of callers, unpredictable call traffic, call drops and skill based routing. Fastest Server First, Longest Idle Server First and Experience Based routings each in different circumstances have been used to service calls in our model [14].

A Caller Node keeps generating IP packets and sends them to the distributor node using a Point-to-Point Link. The Packet length follows a fixed distribution and has a fixed mean value. The Call Skill is denoted by a random variable and is stored in the TOS Tag of the IP Packet. The Caller Patience random variable is stored in the TTL Tag of the IP Packet. This is implemented in NS3 by a function that generates a random variable for Call length, Call Skill and Caller Patience, makes an IP packet of that size, adds the IP Tags and then sends it using a socket. The function then makes a self-call by scheduling the next call after a time interval given by a random number following a negative exponential distribution.
The Distributor Node receives the IP packets and keeps adding them to the tail of the Queue if the Queue is not completely full in which case call is dropped and lost. This is done in NS3 by calling an interrupt to Enqueue function whenever a packet arrives at the socket. It also checks every second if any Agents are idle and distributes the Packets from the Queue to them. It traverses the entire Queue and checks if the idle Agents have the skill required by the Packet. In NS3, it schedules a call to SendPacket function every second, which does the routing of calls using Point to Point Links to Agents with required skill. The Agent Node receive the IP packet from the Distributor Node and becomes busy for the duration of the call. Hence, it changes the Boolean idle flag to zero. While it is busy, it cannot receive any more packets. It schedules a call to ServiceComplete function after a period equal to the length of the call after which it changes its status to idle again.

3.3.1.1: INITIAL SYSTEM MODEL: ERLANG C

Erlang C is a single Queue server model that is modeled using M/M/N Queue [13] which was discussed earlier. Markovian processes model the Calls arrivals and service. There are N Queue servers or Agents, each with a service rate of μ hence the effective service rate is Nμ. All Agents are

homogenous and perfectly efficient at all time. The calls arrive at a fixed rate of λ. The Calls have a mean length and Callers do not hang up. The Callers can wait infinitely for service in the queue i.e. they never abandon the queue. Hence, the queue can build up infinitely and no calls are lost before service.

Simulation Methodology

We built the Queue using a chain of Link Structures in NS3. Each Link has a front and back pointer that point to proceeding and preceding Link. The Link is a container for the Call Packet, which is stored as a Packet Pointer. The length of the packet denotes the length of the call hence the Link also stores the Packet Size as a Data element.

```
struct Link
{
        Link *      Back;
        Link *      Front;
        Uint32_t    Data;
        Ptr<Packet> pkt;
}
```

A Caller Node keeps generating IP packets and sends them to the distributor node using a Point-to-Point Link. The Packet length follows a fixed distribution and has a fixed mean value. This is implemented in NS3 by a function that generates a random variable, makes an IP packet of that size and then sends it using a socket. The function then makes a self-call by scheduling the next call after a time interval given by a random number following a negative exponential distribution.
The Distributor Node receives the IP packets and keeps adding them to the tail of the Queue. This is done in NS3 by calling an interrupt to Enqueue function whenever a packet arrives at the socket. It also checks every second if any Agents are idle and distributes the Packets from the Queue to them. In NS3, it schedules a call to SendPacket function every second, which does the routing of calls using Point to Point Links to Agents.
The Agent Node receive the IP packet from the Distributor Node and becomes busy for the duration of the call. Hence, it changes the Boolean idle flag to zero. While it is busy, it cannot receive any more packets. It schedules a call to ServiceComplete function after a period equal to the length of the call after which it changes its status to idle again.

Next, we discuss the Caller, Distributor and Agent Node Applications designed for this Erlang C Model framework.

Caller Node Application

The Erlang C Caller Node Application is shown in Figure 7.

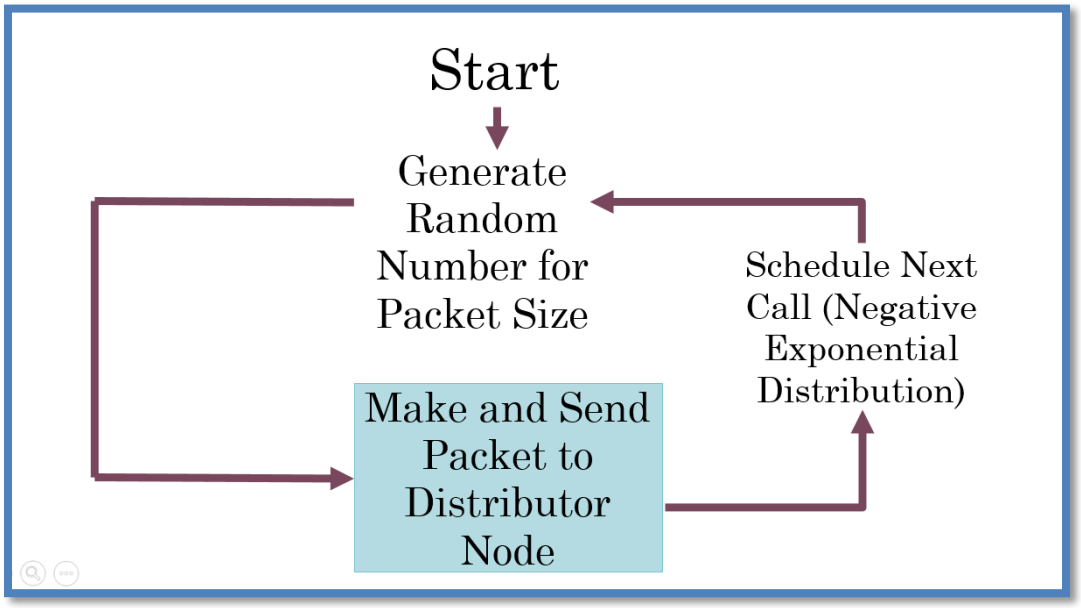


Figure 7: Erlang C Caller Node Application

```
void SendPacket ()
{…

        Size=PacketSizeGenerator->GetInteger ();                    //Generate Random Size
        Ptr<Packet> packet=Create<Packet> (size);                          //Make Packet

        Socket->Send (packet);                                                  //Send Packet

        Time= InterArrivalTimeGenerator->GetInteger ();
        Time tnext (Seconds (Time));                                    //Schedule self-call
        SendEvent=Simulator: Schedule (tnext, &CallerApp: SendPacket, this);

}
```

Distributor Node Application

The Erlang C Distributor Node Application is shown in Figure 8.

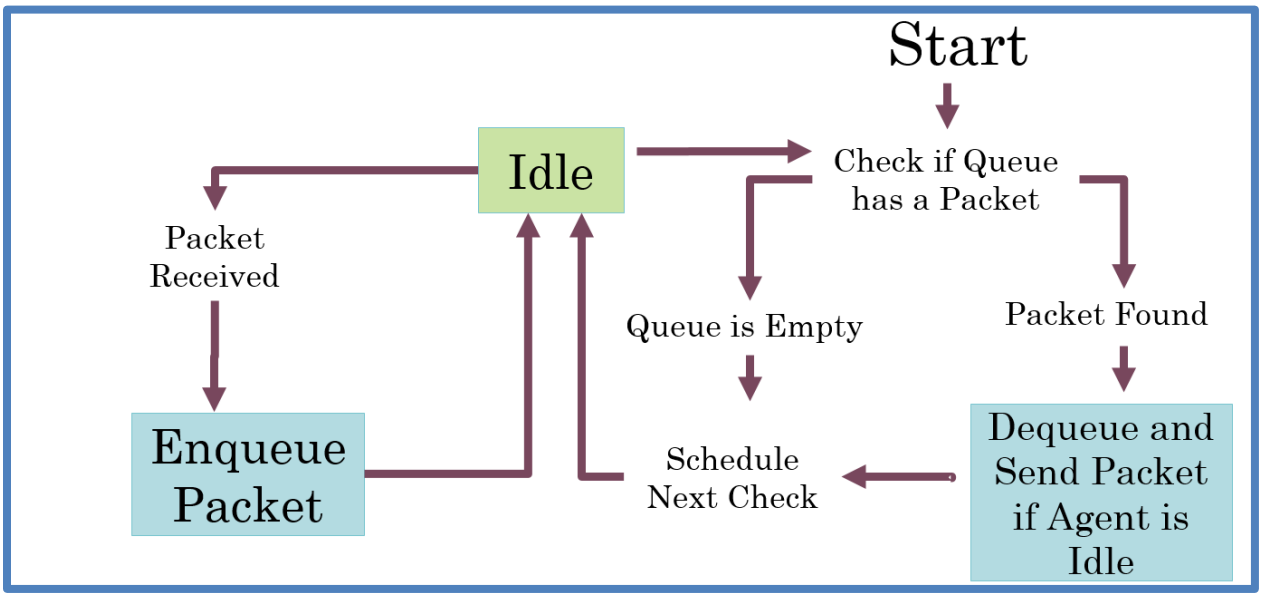


Figure 8: Erlang C Distributor Node Application

```
void SocketRecv (Ptr<Socket> socket)
//Receive Call Back Function
{…
 Ptr<Packet> packet=MySocket->Recv ();
Queue->Enqueue (packet);                                    //Enqueue Packet
 MySocket->SetRecvCallback (MakeCallback (&DistributorApp: SocketRecv, this));
}
void SendPacket ()
{…
 if (idleArray [PeerNumber] && (queue->NumberOfPacketsInQueue>0))
//Check if Agent is idle and Queue has a Packet
{
MyPacket=queue->Dequeue ();                                 //Dequeue Packet
MySocket->Send (MyPacket);                                  //Send Packet to Agent
Time tnext (Seconds (1);                                    //Schedule self-call
SendEvent=Simulator: Schedule (tnext, &DistributorApp: SendPacket, this);
}
}
```

Agent Node Application

The Erlang C Agent Node Application is shown in Figure 9.

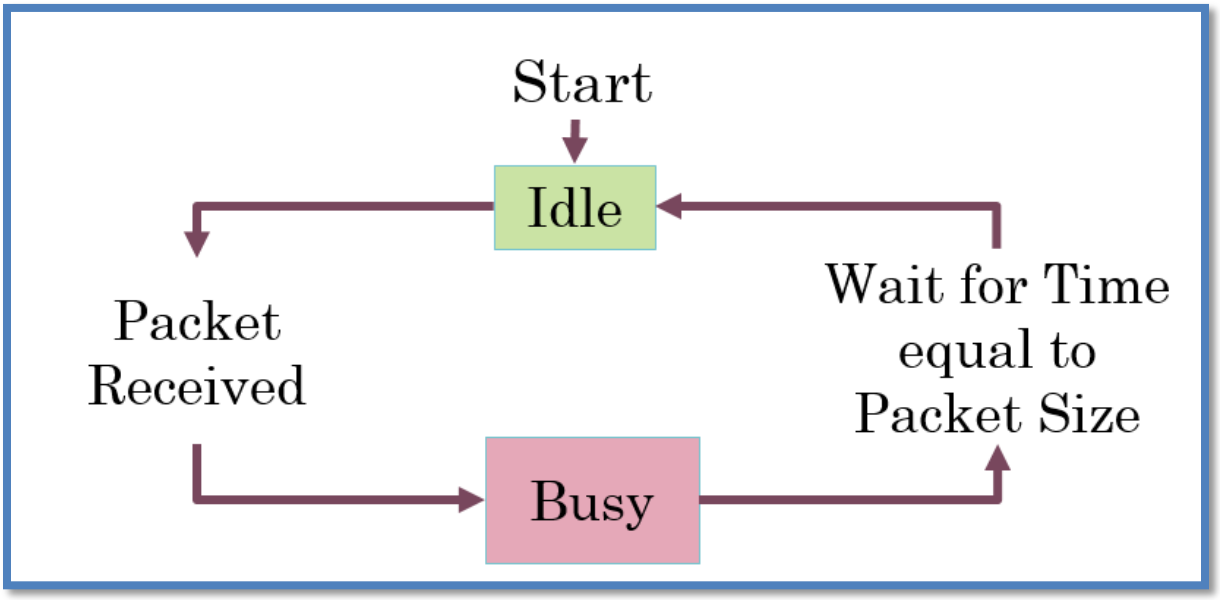


Figure 9: Erlang C Agent Node Application

```
void SocketRecv (Ptr<Socket> socket)
{…
        MyPacket=MySocket->Recv ();                                    //Receive Packet
        PacketSize=MyPacket->GetSize ();
        IdleArray [OperatorNumber] = 0;                                                //Busy
        Time tnext (Seconds (PacketSize));          //Call ServiceComplete after time = PacketSize
        SendEvent=Simulator: Schedule (tnext, OperatorApp::ServiceComplete, this);
}
void ServiceComplete ()
 {…
        IdleArray [OperatorNumber] =1;                                                  //Idle
        MySocket->SetRecvCallback (MakeCallback (&OperatorApp: SocketRecv, this));
                                                          //Call SocketRecv when Packet Comes
}
```

After Verifying the Erlang C Model, we built the popular Erlang X model, which incorporates call abandonments, call drops, and skill based routing.

3.3.1.2: IMPROVED SYSTEM MODEL: ERLANG X

The Erlang X or Erlang A model builds on the Erlang C Model by including abandonments of callers i.e. Callers can abandon Queue after waiting for some time. It is modeled using M/M/M+N Markov Chain [13]. Packets can be dropped if queue is full which models the limited number of telephone lines available. Calls can require multiple skills and Agents can have multiple skills. Specialists have only one skill whereas Generalists can have two or more skills. All Calls have same Call Duration distribution.

Next, we discuss the implementation details and Simulation Methodology of Erlang X model.

Simulation Methodology

Building on the Erlang C model, we modified the Queue Link by keeping a Skill section in each Link to aid distribution of packets to Agents of desired skill. The QueueWaitingTime Element denotes the Maximum time the caller will wait in the Queue before abandoning the Queue.

```
Struct Link
{
        Link *      Back;
        Link *      Front;
        Uint32_t    Data;
        Uint32_t    Skill;
        Ptr<Packet> pkt;
        Time        EnqueueTime;
        Uint32_t    QueueWaitingTime;
}
```

The Patience Counter of each Link decrements every second. If Timer of a Link hits zero, we remove that Link from Queue. Hence, the caller has abandoned the Queue before being serviced.

```
void CountDown ()
{…
 If (NumberOfPacketsInQueue>0)
 {
  Link* a          =Head;
 While (a! =Tail)                                    //Traverse Entire Queue
 {
   a              =a->Back;
```

```
   a              ->QueueWaitingTime--;                                    //Decrement counter
   If (a->QueueWaitingTime==0) {a-Back->Front = a->Front ;}                 // Abandonment
 }
 }
}
```

Before Enqueue, we check if queue is full. If queue is full, packet is dropped.

```
void Enqueue (Ptr<Packet> Packet)
{…
If (NumberOfPacketsInQueue<MaximumQueueLimit)                    //Check Queue Limit
{
 Packet->Front = Tail-> Front;
 Tail->Front = packet;                                                    //Enqueue
}
 Else
 {
  Myfile<<"Packet Dropped, MaximumQueueLimit Exceeded"<<endl;             //Drop
  }
}
```

To prevent one packet from blocking entire queue, we allow distributors to pick packet from any link of queue.

```
If (idle [PeerNumber] && (queue->NumberOfPacketsInQueue>0))
{…
 DistributorQueue::Link* a=queue->Head->Back;
 DistributorQueue::Link* b=0;
 While (a! =queue->Tail)                                          //Traverse Entire Queue
 {
  If ((SkillArray [PeerNumber] & a->Skill)>0)                     //Skill Matching
  {
 Socket->Send (a->Packet)                                         //Send Packet
 }
        }
}
```

Now we discuss the Caller, Distributor and Agent Node Applications designed for this Erlang X Model framework.

Caller Node Application

The Erlang X Caller Node Application is shown in Figure 10.

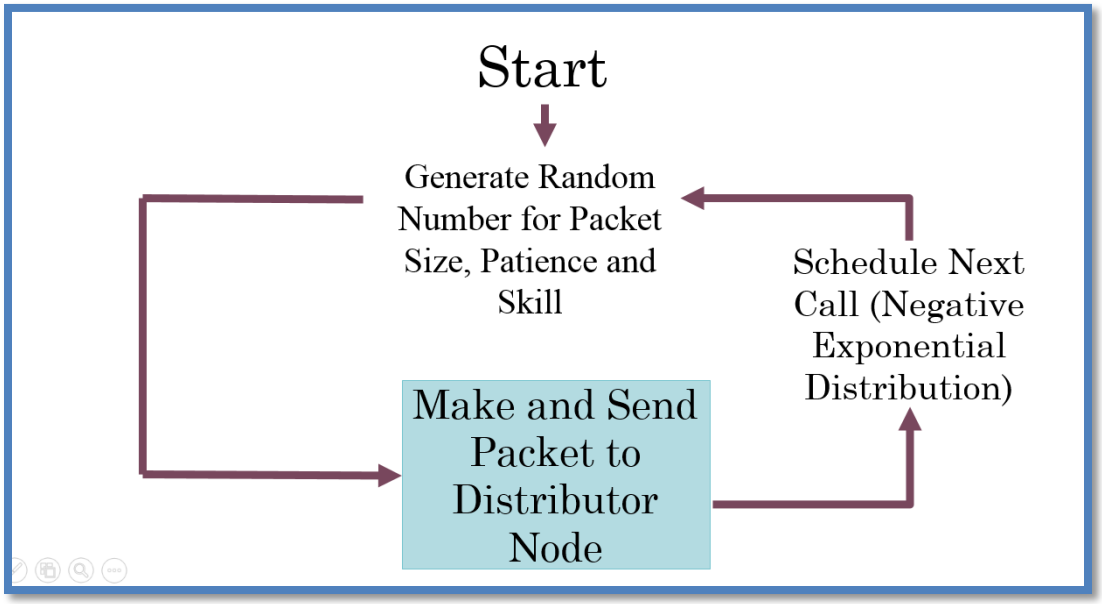


Figure 10: Erlang X Caller Node Application

```
void SendPacket ()
{…
 Size             =PacketSizeGenerator         ->GetInteger ();
Waitingtime =WaitingTimeGenerator       ->GetInteger ();
Skill             =SkillTypeGenerator           ->GetInteger ();
Time            =InterArrivalTimeGenerator->GetInteger ();   //Generate Random Numbers
Socket->SetIpTos (skill);                                                    //Set Packet Skill
 Socket->SetIpTtl (waitingtime);                                    //Set Packet Waiting Time
 Ptr<Packet> packet=Create<Packet> (size);                                 //Make Packet
 Socket->Send (packet);                                                      //Send Packet
Time tnext (Seconds (static_cast<double> (time)));                  //Schedule self-call
SendEvent=Simulator: Schedule (tnext, &CallerApp: SendPacket, this);
}
```

Distributor Node Application

The Erlang X Distributor Node Application is shown in Figure 11.

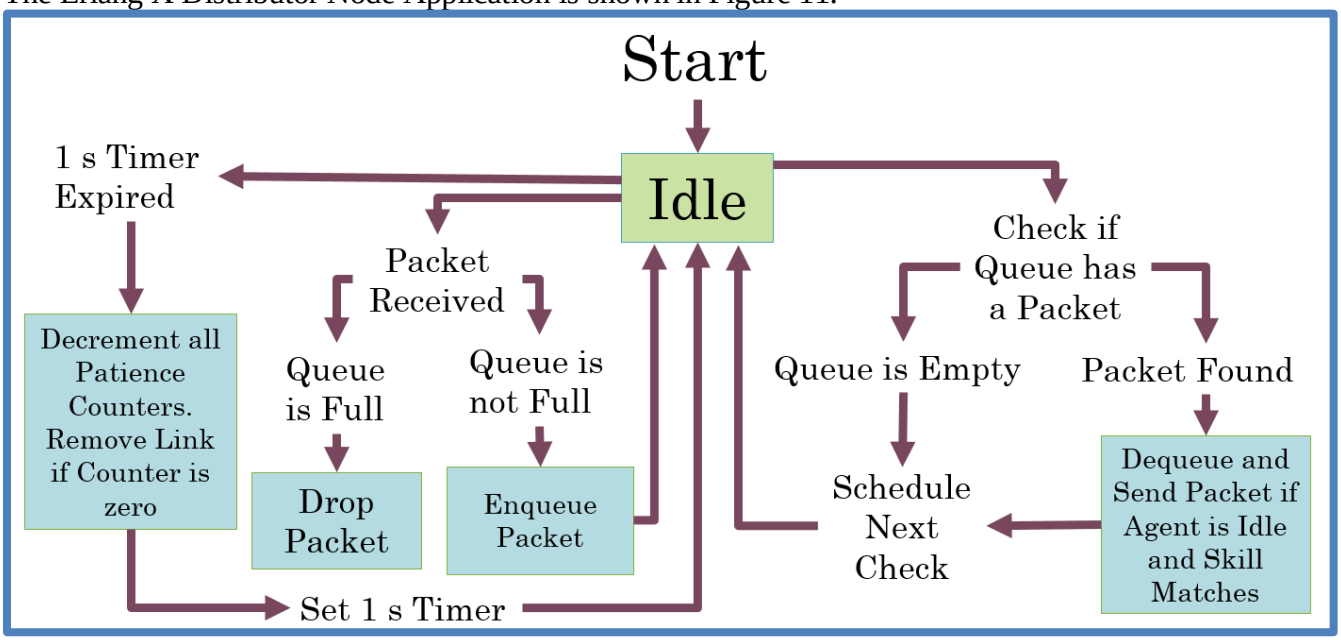


Figure 11: Erlang X Distributor Node Application

```
void SkillBasedRouting ()
{…
If (idle [PeerNumber] && (queue->NumberOfPacketsInQueue>0))                    //Checks
{
  DistributorQueue::Link* a=queue->Head->Back;
  If ((SkillArray [PeerNumber] & a->Skill)>0)                                  //Match Skill
  MySocket                  ->Send (a->Packet);                                //Send Packet
  Time                    tnext (Seconds (1));                               //Schedule self-call
  SendEvent =Simulator::Schedule (tnext, &DistributorApp: SkillBasedRouting, this);
 }
}
void SocketRecv (Ptr<Socket> socket)
{…
Address from;
          Ptr<Packet> packet=MySocket->RecvFrom (from);                        //Receive Packet
          Queue->Enqueue (packet);                                             //Enqueue Packet
           MySocket->SetRecvCallback (MakeCallback (&ReceiverApp: SocketRecv, this));
}
```

Agent Node Application

The Erlang X Agent Node Application is shown in Figure 12.

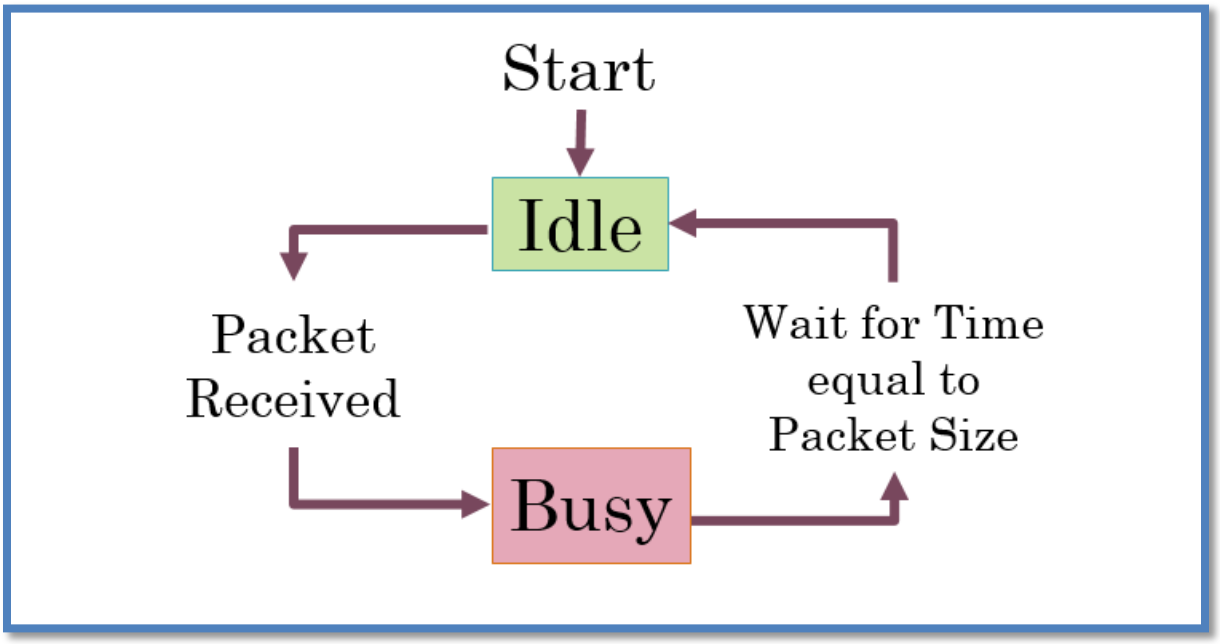


Figure 12: Erlang X Agent Node Application

```
void SocketRecv (Ptr<Socket> socket)
{…
 MyPacket      =MySocket->Recv ();                                    //Receive Packet
 PacketSize    =MyPacket->GetSize ();
 Time tnext (Seconds (PacketSize);              //call ServiceComplete after time = Packet Size
 SendEvent     =Simulator::Schedule (tnext, &AgentApp: ServiceComplete, this);
IdleArray [AgentNumber] =0;                                          //Become Busy
 }

void ServiceComplete ()
{…
 IdleArray [AgentNumber] =1;                                          //Become Idle
MySocket ->SetRecvCallback (MakeCallback (&AgentApp: SocketRecv, this));
                                          //Call SocketRecv Function when call arrives
}
```

We built our final Call Centre model by adding retrials, Virtual Response Unit, heterogeneous agents, time varying arrival rates and Agent Schedules.

3.3.2: NODE.JS

We made an internet based call center application in Node.js [18] which allows users to connect their internet devices to our HTTPS Server using an IP Address and Port, even through firewalls and NATs. The user can choose to become an Agent or a Caller on Home Page. As shown in Figure 13, Signaling for Javascript Session Establishment is done using Socket.io Library [26]. Caller and Agent stream live audio and Files using WebRTC Point to Point Connection [29].

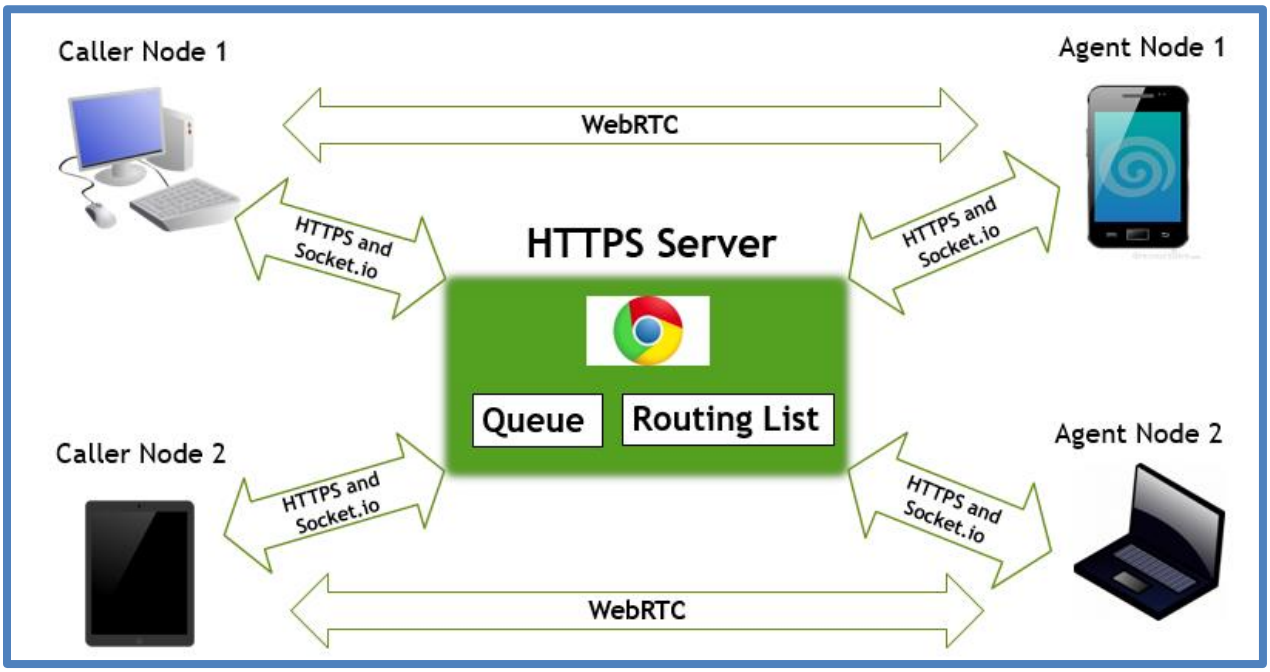


Figure 13: Node.js Call Center Block Diagram

The Node.js application was based on the Call Center Simulation in NS-3. It incorporated all the features of the simulation hence we could replicate IP calls by bidirectional audio calls and exchange of Text messages or Files between Caller and Agent. By collecting data in real time, we could record all the important performance metrics and compare them with our NS-3 Simulator results.

Node.js is single-threaded, non-blocking and asynchronous [18], which is very memory efficient. It works asynchronously to service multiple requests at same time.

We have used Node.js due to the following reasons:

- Node.js is an open source server framework
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Node.js uses JavaScript on the server

In comparison with the synchronous programming the asynchronous programming is time saving, as in case of the synchronous programming, we cannot go to the next line of command until and unless the first line of command is executed completely and properly. While in case of the asynchronous programming multiple lines of commands can be executed at the same time.

Asynchronous programming will avoid freezing of browsers, improved speed of the website applications, improve performance when multiple clients are using web services at the same time. It is effective in case of Fault tolerance. We can get rid of the DOM updates, CSS animation and user interactions (right clicking, text selection etc) getting blocked or frozen.

3.3.2.1: HYPERTEXT PREPROCESSOR/ ACTIVE SERVER PAGES PROTOCOLS

PHP is a widely-used, open source scripting language. Scripts are executed on the server. PHP can be used for generating dynamic page content and for performing the file operations like create, open, read, write, delete, and close files on the server. With PHP you are not limited to output HTML. You can output images, PDF files, and even flash movies. You can also output any text, such as XHTML and XML. Active Server Pages enable computer code to be executed by an Internet server [36].

PHP or ASP both are the synchronous protocols and they handle the file request in the following manner:
1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

However, Node.js which uses asynchronous programming handles the file request in the following manner and saves a lot more time.
1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

The Node.js Event Loop is shown in Figure 14 [36].

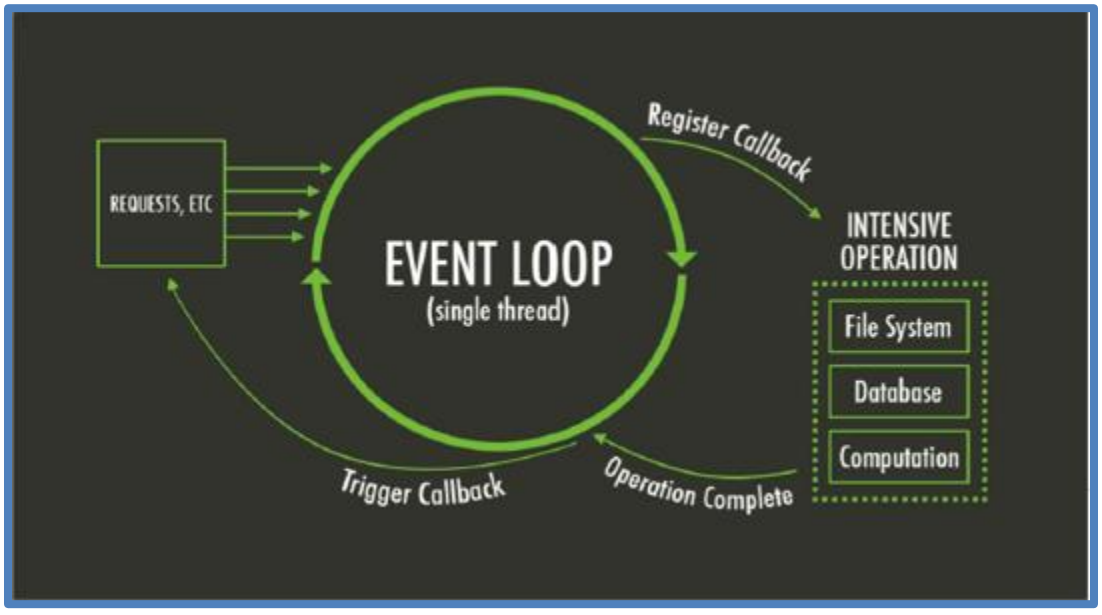


Figure 14: Node.js Event Loop [36]

3.3.2.2: SECURED NETWORK

SSL (Secure Sockets Layer) and TLS (Transport Layer Security) these are the protocols that ensures privacy between communicating applications and their users on the Internet. TLS is an updated and more secured version of SSL. Both SSL and TLS are used to encrypt confidential data sent over an insecure network such as the Internet. They are the cryptographic protocols that provide communications security over a computer network. These standard security technologies establish an encrypted link between a web server and a browser. This link ensures that all data passed between the web server and browsers remain private and integral. The connection is private (or secure) because symmetric cryptography is used to encrypt the data transmitted. The keys for this symmetric encryption are generated uniquely for each connection and are based on a shared secret negotiated at the start of the session [35].

3.3.2.2.1: SECURE HYPER TEXT TRANSFER PROTOCOL

HTTP Secure (HTTPS) is an extension of the Hypertext Transfer Protocol (HTTP) for secure communication over a computer network, and is widely used on the Internet. In HTTPS, the communication protocol is encrypted by Transport Layer Security (TLS), or formerly, its predecessor, Secure Sockets Layer (SSL). The protocol is therefore also often referred to as HTTP over TLS, or HTTP over SSL. The principal motivation for HTTPS is authentication of the accessed website and protection of the privacy and integrity of the exchanged data while in transit. It protects against man-in-the-middle attacks. The bidirectional encryption of communications between a client and server protects against eavesdropping and tampering of the communication. In practice, this provides a reasonable assurance that one is communicating without interference by attackers with the website that one intended to communicate with, as opposed to an impostor. HTTPS creates a secure channel over an insecure network. Over insecure networks (such as public Wi-Fi access points), as anyone on the same local network can packet-sniff and discover sensitive information not protected by HTTP. This ensures reasonable protection from eavesdroppers and man-in-the-middle attacks, provided that adequate cipher suites are used and that the server certificate is verified and trusted. The security of HTTPS is that of the underlying TLS, which typically uses long-term public and private keys to generate a short-term session key, which is then used to encrypt the data flow between client and server. Certificates are used for the server's authentication. As a consequence, certificate authorities and public key certificates are necessary to verify the relation between the certificate and its owner, as well as to generate, sign, and administer the validity of certificates [35].
Difference from HTTP:

- HTTPS URLs begin with "https://" and use port 443 by default, whereas HTTP URLs begin with "http://" and use port 80 by default.
- HTTP is not encrypted and is vulnerable to man-in-the-middle and eavesdropping attacks, which can let attackers gain access to website accounts and sensitive information and modify webpages to inject malware or advertisements. HTTPS is designed to withstand such attacks and is considered secure against them (with the exception of older, deprecated versions of SSL).

HTTP operates at the highest layer of the TCP/IP model, the Application layer; as does the TLS security protocol (operating as a lower sublayer of the same layer), which encrypts an HTTP message prior to transmission and decrypts a message upon arrival. Strictly speaking, HTTPS is not a separate protocol, but refers to use of ordinary HTTP over an encrypted SSL/TLS connection [35].

3.3.2.2.2: TRANSPORT LAYER SECURITY

A TLS Handshake between a HTTPS Server and Client, as shown in Figure 15 [35], has the following steps:

- The connection between the client and the server establishes in the following way through TLS protocol:
- A client sends a “Client Hello” message specifying a list of suggested cipher suites (set of algorithms that help to secure the network connection that uses TLS/SSL) and suggested compression methods.
- The server responds with a “Server Hello” message, containing the suggested Cipher Suite and compression method from the choices offered by the client.
- The server sends its “Certificate message”.
- The server sends its “Server Key” Exchange message.
- The server sends a Server Hello Done message, indicating it is done with handshake negotiation.
- The client responds with a “Client Key Exchange” message, which contain a “Pre-Master Secret”, public key. This Pre-Master Secret is encrypted using the public key of the server certificate.
- The client and server then use the random numbers and Pre-Master Secret to compute a common secret, called the "master secret". All other key data for this connection is derived from this master secret.
- The client now sends a “Change CipherSpec” record, essentially telling the server, "Everything I tell you from now on will be authenticated (and encrypted if encryption parameters were present in the server certificate).
- Finally, the client sends an authenticated and encrypted finished message, containing a hash and MAC over the previous handshake messages.
- The server will attempt to decrypt the client's finished message and verify the hash and MAC. If the decryption or verification fails, the handshake is considered to have failed and the connection should be torn down.
- Finally, the server sends a Change CipherSpec, telling the client, "Everything I tell you from now on will be authenticated (and encrypted, if encryption was negotiated).
- The server sends its authenticated and encrypted finished message.
- The client performs the same decryption and verification procedure as the server did in the previous step.
- Application phase: at this point, the "handshake" is complete and the application protocol is enabled. Application messages exchanged between client and server will also be authenticated and optionally encrypted exactly like in their finished message.

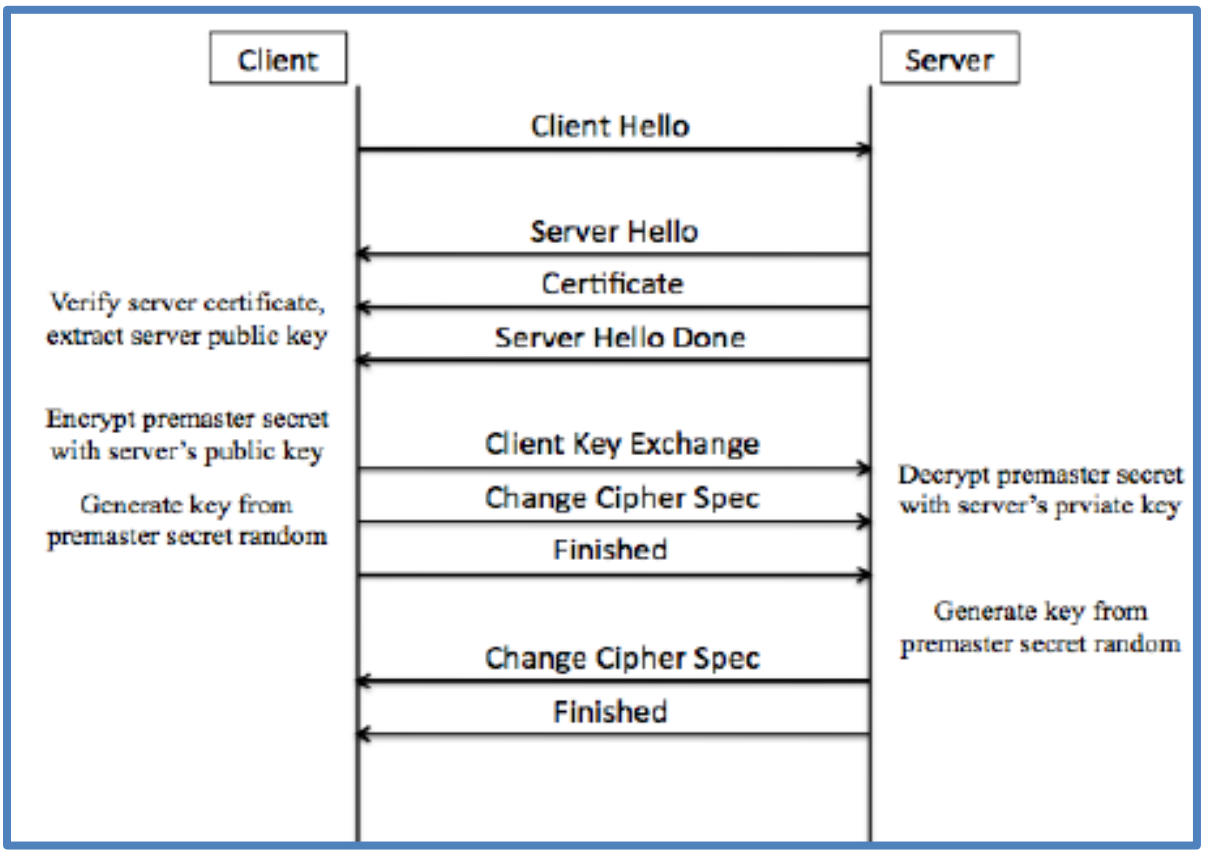


Figure 15: TLS Handhsake betwwen Server and Client
[35]

3.3.2.2.3: DIGITAL CERTIFICATES

Certificates are public keys that are digitally signed either by a Certificate Authority or by the owner of the private key. Those certificates that are signed by the owner of the private key are referred to as "self-signed ". In turns these certificates correspond to a private key.
Steps to get the Certificate [18]:

- The first step to obtaining a certificate is to create a Certificate Signing Request (CSR) file. The OpenSSL command-line interface can be used to generate a CSR for a private key
- Once the CSR file is generated, it can either be sent to a Certificate Authority for signing or used to generate a self-signed certificate.
- Once the certificate is generated, it can be used to generate a .pfx or .p12 file.

The different files that we generated are shown in Figure 16 below.

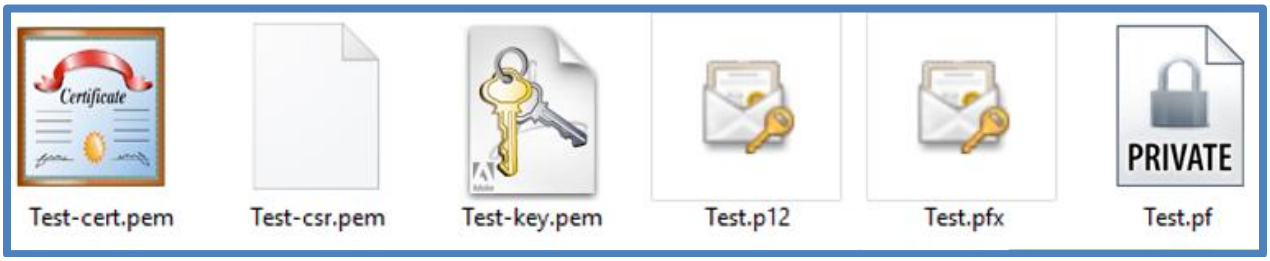


Figure 16: Privacy Enhanced Mail Certificates, Public Key Cryptography Standard #12 file, Personal Information Exchange Certificate and Private File for HTTPS Encryption

3.3.2.2.4: THE PUBLIC/ PRIVATE KEY

This sections explains a server-client Handshake. As shown in Figure 17 [35], when a client communicates with the server, the client sends the client "Hello' message which is the beginning of the communication, then the server sends back the certificate which includes the public key that the server has. The public key is already shared to everyone the client uses it to identify the server. Afterwards, the client computes the premaster secret after computing the premaster key it encrypts it using the public key of the server and sends it to the server. The server can then use its private key to then decrypt that encrypted premaster key. Now the server has the premaster secret. Now from the premaster secret the client and the server both generates the master secret and the session key which then leads to bulk encryption. But what happens if the private key is ever compromised either by the hackers or the security agencies. If the hacker can ever get the private key and then what he can do is that he can decrypt the premaster key sent by the client and then he can generate from that premaster secret the actual secret key that is being used in the bulk encryption and can decrypt any information from the secret key. In order to avoid this situation, the perfect forward secrecy comes into play [35].

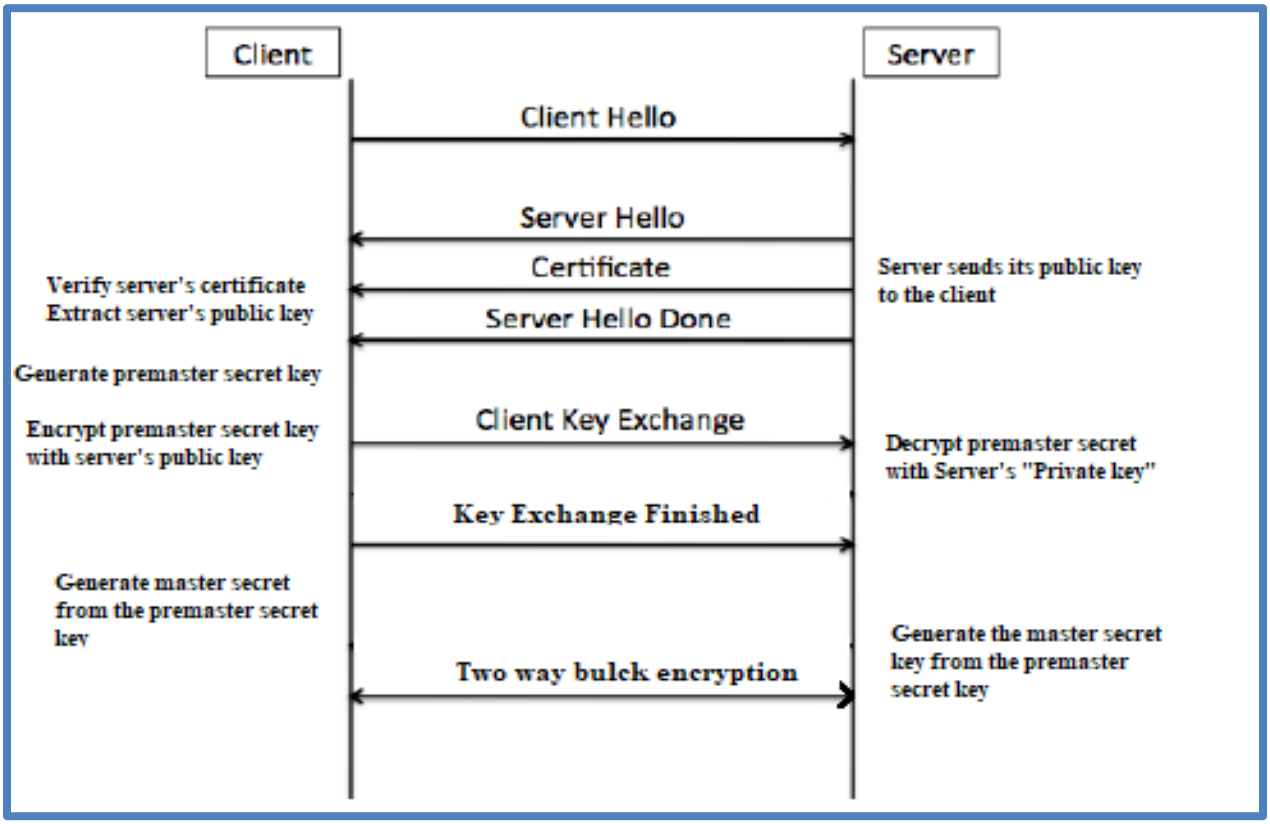


Figure 17: TLS Key Exchanges [35]

3.3.2.2.5: PERFECT FORWARD SECRECY

This section explains Perfect Forward Secrecy using Figure 17 [35] below. The steps are the following:

- Client sends the hello message as normal and then what the server's going to do is that the server then generates a prime number and also the modulo and sends back to the client but also a sit sends back to the client. It never relies on the server or the client using their public or private key in order to exchange the premaster secrets and ultimately the keys.
- Server picks the known prime number and the modulo and then it picks the random integer from this it calculates the value: the value A
- The client will pick its own random integer and using the same prime number and modulo, it calculates its value. It sends that value B back to the server.
- Through the values of A and B they both can generate the premaster secret. So, they both have the pre-master secret and they've got it without using any of their private key. And using this premaster secret, both generate the secret key and then starts communicating. Also, these randomly chosen integers are ephemeral (short lived).
- Brand new random integers are chosen all the time… ephemeral key exchange. And even if someone grab hold of the random integer. Then the communication of that one channel will be compromised but the next time it's completely different key.

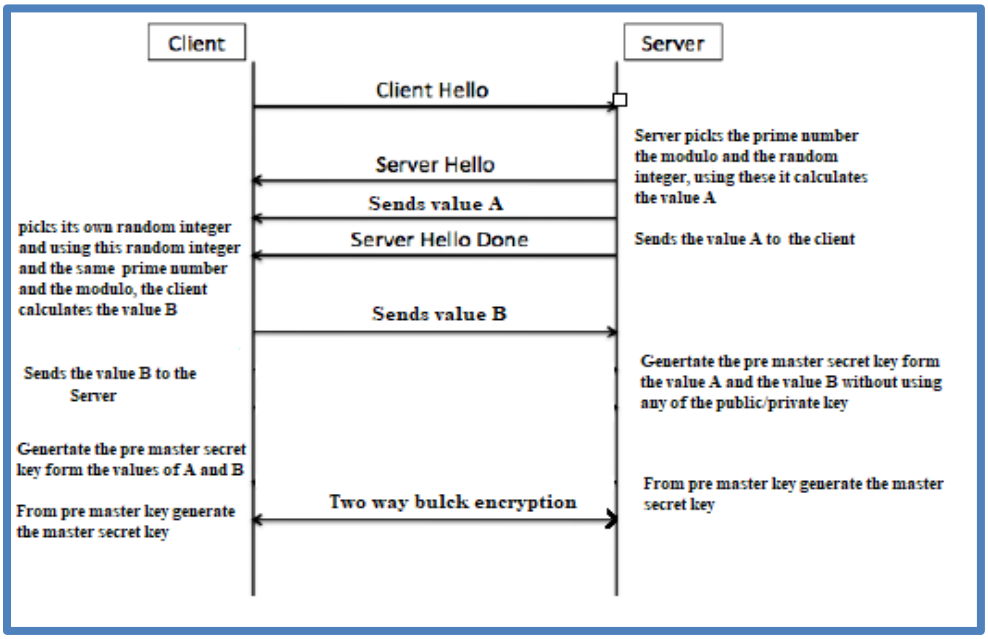


Figure 18: Perfect Forward Secrecy [35]

3.3.2.3: JAVASCRIPT SESSION ESTABLISHMENT PROTOCOL

The Javascript Session Establishment Protocol Architecture [16] is shown in Figure 19 [34].

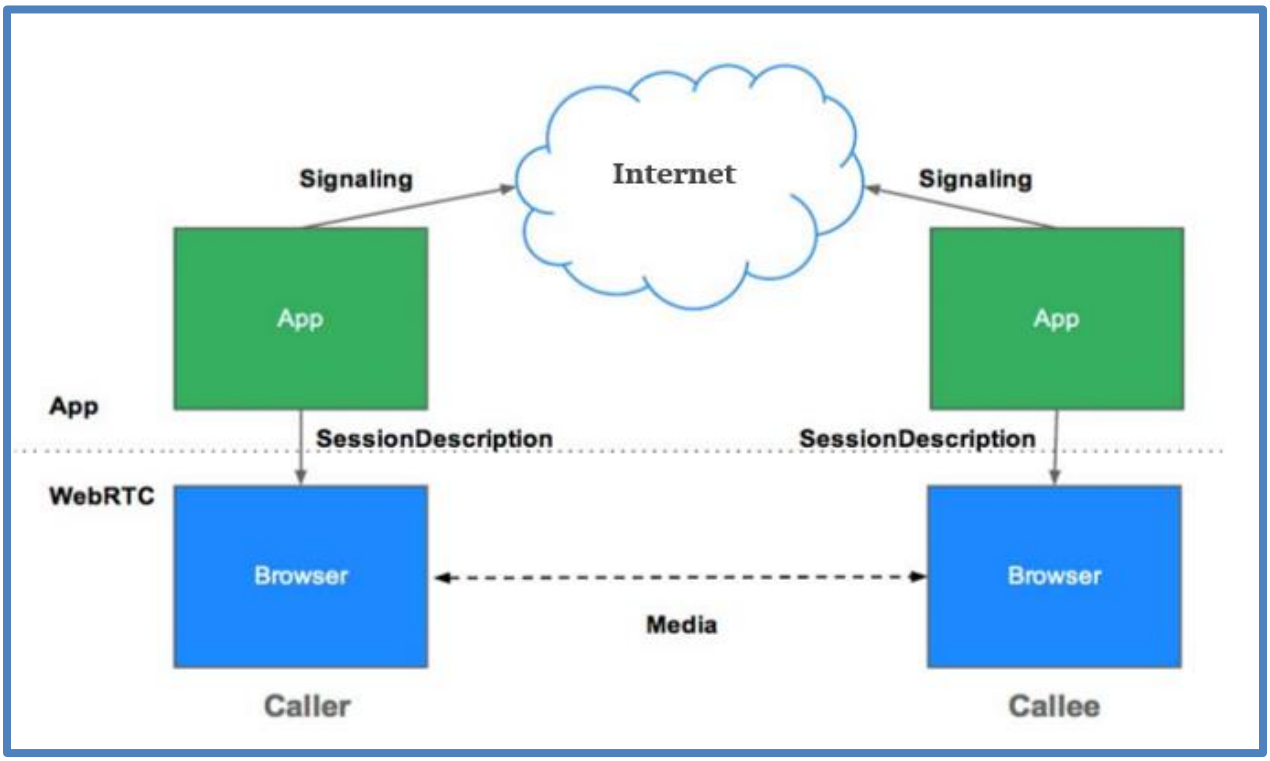


Figure 19: JSEP Architecture [34]

3.3.2.4: WEBRTC

WebRTC enables real time communication without any plugins. Provided the fact that live communication has been either very complex or expensive to implement, WebRTC provides a more cost effective and simpler mode of RTC (real time communication). Its first implementation was built by Ericsson in May 2011. It now enables video, data and audio communication. The guiding principle of the WebRTC project are its APIs, which by being open source, free, standardized and can be built into browsers and are far more efficient than the existing technologies [34].

3.3.2.4.1: STANDARDS AND PROTOCOLS

- The WebRTC W3C Editor's Draft
- W3C Editor's Draft: Media Capture and Streams (aka getUserMedia)
- IETF Working Group Charter
- IETF WebRTC Data Channel Protocol Draft
- IETF JSEP Draft
- IETF proposed standard for ICE
- IETF RTCWEB Working Group Internet-Draft: Web Real-Time Communication

WebRTC is used in various apps such as Facebook, Whatsapp, IOS Browser, appear.in etc.
WebRTC constitutes three APIs; Media Stream, RTC peer connection and RTC Data Channel [34].

3.3.2.4.2: WEBRTC SUPPORT SUMMARY

MediaStream and getUserMedia are supported by Chrome, Opera, Firefox and Microsoft Edge.
RTCPeerConnection is supported by Chrome, Opera and Firefox.
RTCDataChannel is supported by Chrome, Opera and Firefox [22].

3.3.2.4.3: WEBRTC APPLICATIONS

WebRTC not only enables the live streaming of audio, video and other data but also enables access to the network information such as IP addresses and port numbers. It is the exchange of this information between with other WebRTC clients is necessary to enable peer connections with them, even through NATs and firewalls. The signaled communication supported by WebRTC enables errors to be reported. Information about media and client capability sucha s codecs and resolution can also be exchanged in WebRTC [29].

The WebRTC Architecture [31] is shown in Figure 20 [34].

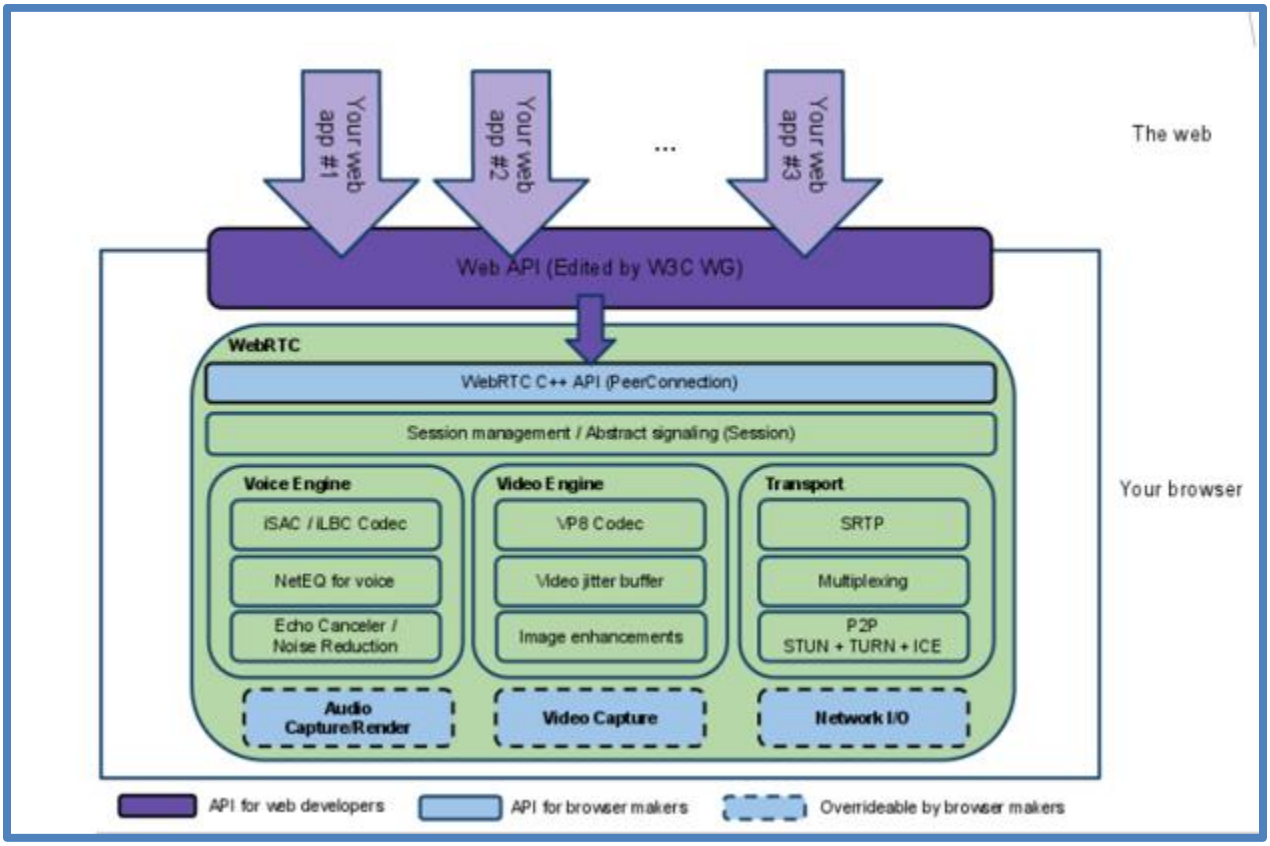


Figure 20: WebRTC Architecture [34]

3.3.2.4.4: MEDIA STREAM

The API for Media Stream supports synchronized media streams. The input for each Media Stream is generated by navigator.getUserMedia ( ). The output is passed to a video element or an RTC Peer Connection. The getUserMedia ( ) takes three parameters; a constraint object, a success callback and a failure callback. If a success callback is called, it is passed a Media Stream while if a failure callback is called an error object is passed in response. Each Media Stream has a label and these media streams cab retrieved in an array of Media Stream Tracks by using functions getAudioTracks() and getVideoTracks() [17].

3.3.2.4.5: RTC PEER CONNECTION

The RTCPeerConnection objects can exchange data and messages directly without having to use any intermediary signaling mechanisms. The caller create a new RTCPeerConnection and adds the stream from getUserMedia (). It then create an offer and sets it as the local description for the pc on which it is operating and as the remote description for the other pc of the callee. In the process it does not use any signaling communication. While the Callee when receives the stream from the caller it displays it as a video element [22].

WebRTC require four types of server-side functionality. These are signaling, NAT/firewall traversal, user discovery and communication and Relay servers. NAT traversal in RTC Peer Connection is achieved through implementing STUN and TURN protocols [28].

3.3.2.4.6: INTERACTIVE CONNECTIVITY ESTABLISHMENT

ICE candidates enable direct communication between the peers through either STUN (Session Traversal Utilities for NAT) or TURN (Traversal Using Relay NAT) Servers. As these enable access to the network information of peers behind NAT or firewalls. STUN servers are used initiallt fir RTC Peer connection if these fail then Relay servers are used instead [28].

The Interactive Connectivity Establishment framework is shown in Figure 21 [34].

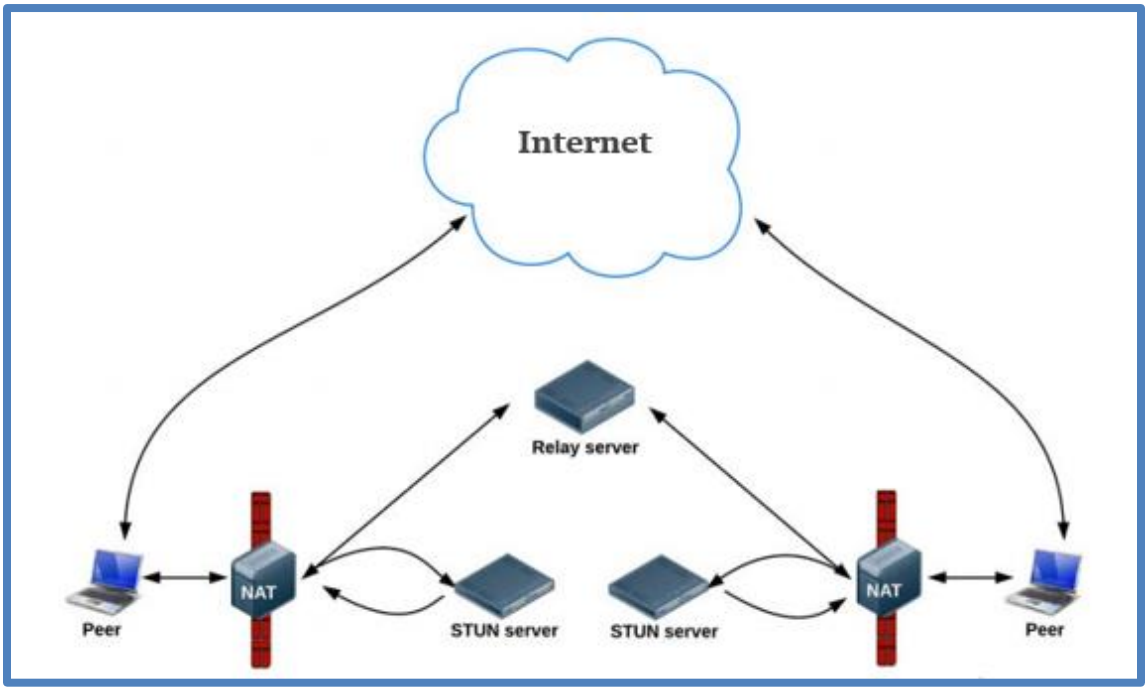


Figure 21: Framework for finding ICE Candidates [34]

3.3.2.4.7: RTC DATA CHANNEL

The RTCDataChannel API enables peer-to-peer exchange of arbitrary data, with low latency and high throughput. There are many potential use cases for the API, including: gaming, remote desktop application, Real-time text chat, file transfer and decentralized networks. The API has the following features: leveraging of RTCPeerConnection session setup, multiple simultaneous channels with prioritization, Reliable and unreliable delivery semantics, Built-in security (DTLS) and congestion control and the Ability to use with or without audio or video. Communication occurs directly between browsers, so RTCDataChannel can be much faster than WebSocket [22].

3.3.2.4.8: SIGNALING

WebRTC uses RTCPeerConnection to communicate streaming data between browsers, but also needs a mechanism to coordinate communication and to send control messages, a process known as signaling.Signaling is not part of the RTCPeerConnection. Signaling is used to exchange three types of information. Session control messages for initializing or closing communication and reporting error. Network configuration that is the IP address and port .Media capabilities, these include codecs and resolutions [29].

3.3.2.5: GLOBAL FUNCTIONALITY OF CALL CENTER

The Call Initiation Chain is shown in Figure 22 [34]. Initially upon request the users receive their respective pages. The caller then calls the getUserMedia () function [20] too get its stream which it saves as its local and starts a new peer connection object using this stream. The caller presses the call button and calls the createOffer () function of the peer connection. The call back of this function is set as local description [24] and is emitted to the agent who, similarly, on the other end has saved its local stream by calling the getUserMedia () [20]. The agent recieves the offer, saves it as its remote description and using this offer creates an answer and sends it over to the caller. The caller recieves this answer and saves it as its remote description [24]. The caller selects an ice candidate [27] from the arrived list of available candidates to its PC and sends it over the socket channel using socket.on () [26]. The agent sets the candidate and sends over its ice candidate using socket.emit () [26]. The remote streams are exchanged when a RTC PeerConnection communication channel is established [22].

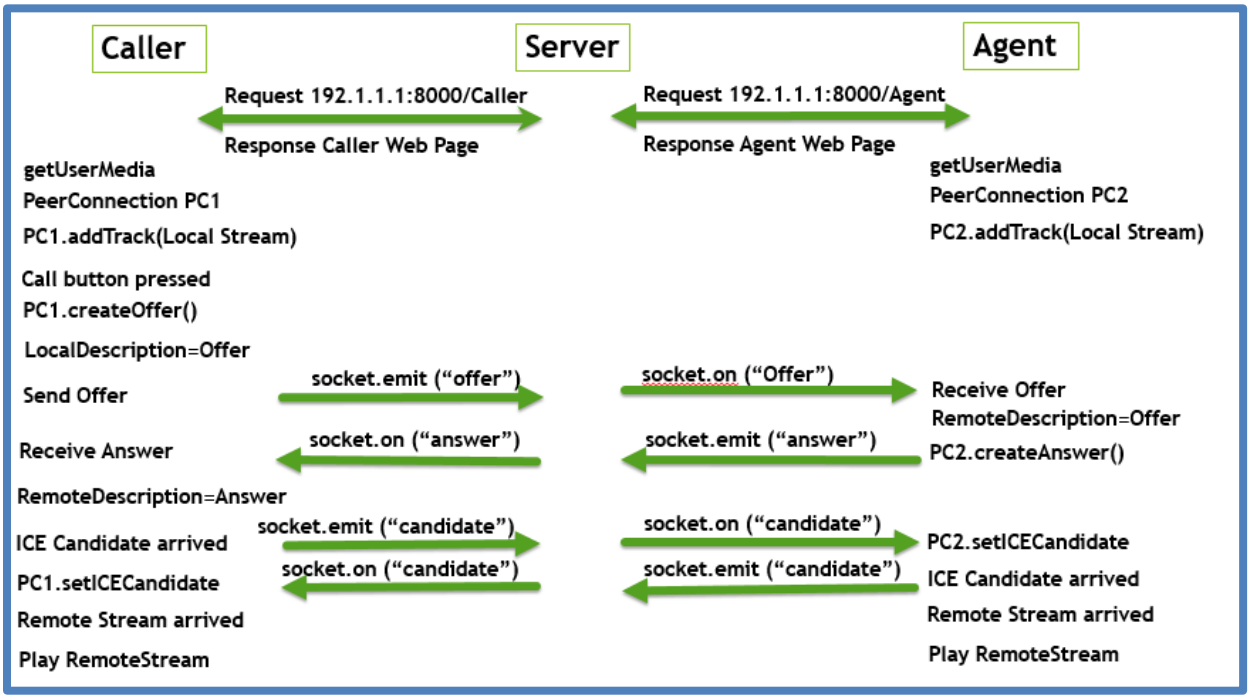


Figure 22: Call Initiation Chain [34]

3.3.2.6: SERVER CODE

We made a mediator server using Nodejs which connected the caller-agent peer and handled their respective communication.

We required various modules to implement our server [19]. We used namely:

- fs: for handling file system transfer between CallCentre, Caller and agent html page.
- http: to make Node.js act as HTTP server, similarly https for secured connectivity.
- socket.io: this module was used for real time event-based communication (request, response between caller and agent) [26].
- os: this provides several operating system-related utility methods.

We handled number of callers and agents by having separate socket arrays so that upon availability and skill matching, a pair can be made of the caller and the agent candidate. As we are using socket library for real-time communication, we used its following events

- Type: user defines if it wants to be agent/caller.
- ICECandidate: sends ICE candidate [27].
- Offer: sends Session Description Protocol SDP [24].
- TextMessage: sends and receives text messages.
- Answer: answer SDP to caller offer.
- FileInformation: sends file information from caller and sends to agent.
- Disconnect: fired when client (caller/agent) disconnects, count added to Drop variable

3.3.2.7: CALL CENTER HOME WEB PAGE

This Call center Home web page is shown in Figure 23 which can be accessed using https://10.103.76.173:8000/

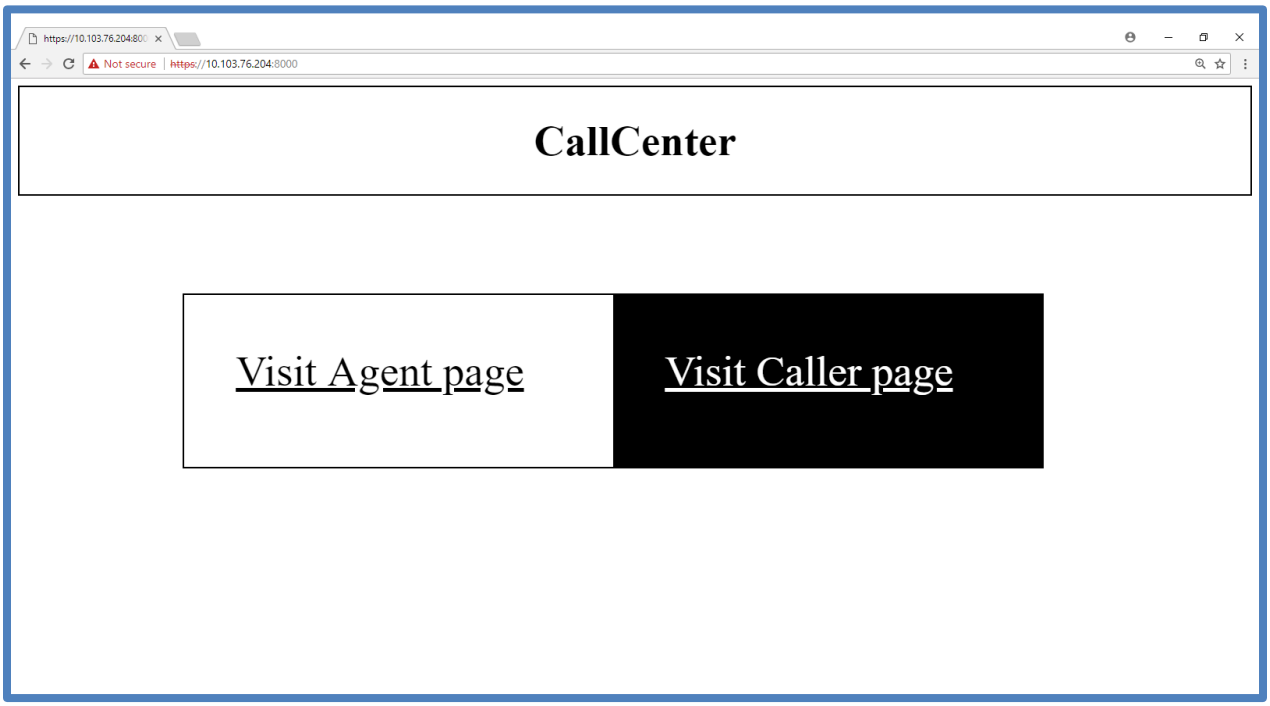


Figure 23: Call Center Home Web Page

The user then opts to become a caller, or an agent and they are then forwarded to their respective pages as below :

3.3.2.8: CALLER WEB PAGE

The Caller Web Page is shown in Figure 24. It has dedicated blocks showing local and remote audio streams, and a Fourier visualizer. Caller can send and receive text and files, choose the skill, bandwidth, codec, call and hang-up.

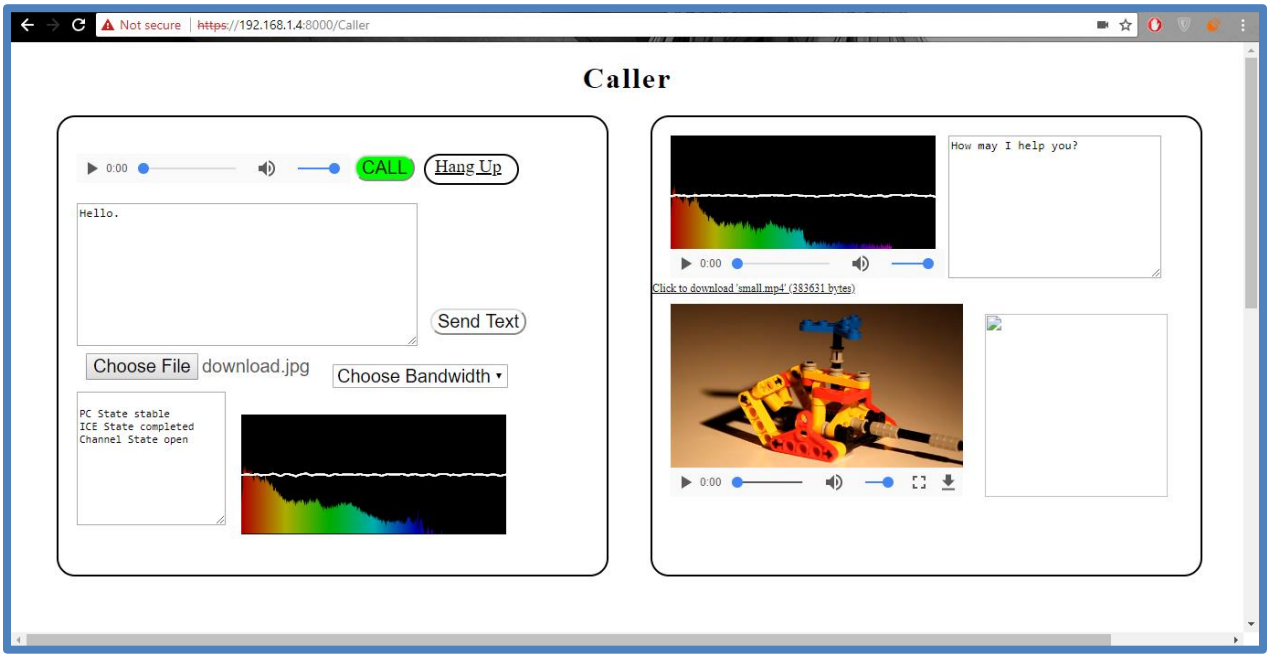


Figure 24: Caller Web Page

3.3.2.9: AGENT WEB PAGE

The Agent Web Page is shown in Figure 25. It has dedicated blocks showing local and remote audio streams, and a Fourier visualizer and available functions area. Agent can send, receive text and files, choose the skill, call and hang-up.

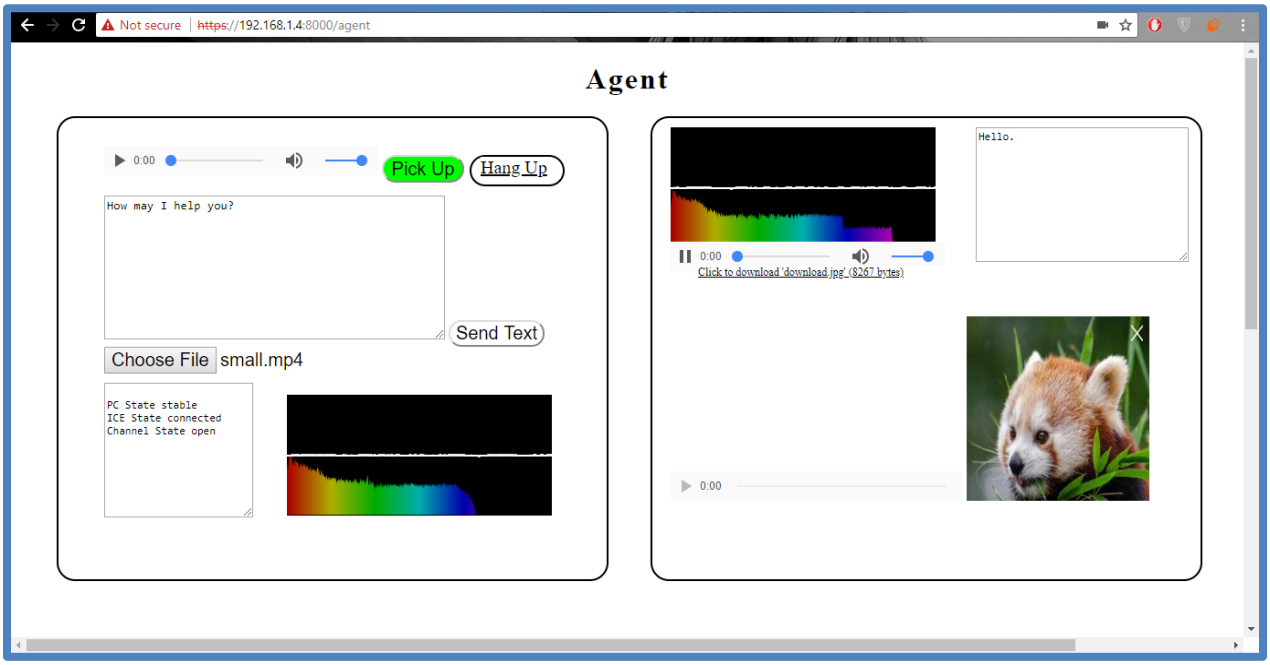


Figure 25: Agent Web Page

3.3.2.10: ADDITIONAL FEATURES

We added several additional features to our system, namely : a log file for call records, calculating performance metrics at server, the functionality to change bandwidth on the fly and the funcionality that the caller can specify upon which bandwidth it can communicate (125bps, 250bps, 500bps, 1000bps or Unlimited Bandwidth.).

Another flexibility is in choosing Audio Codec from a given the list : (iLBC (Default), iSAC 16K, Opus, G722, PCMU). Both pages have a stream visualizer showing Fourier Transform of local and remote Audio Signal. We also enabled group calls through making and managing multiple RTC PeerConnections at the same time.

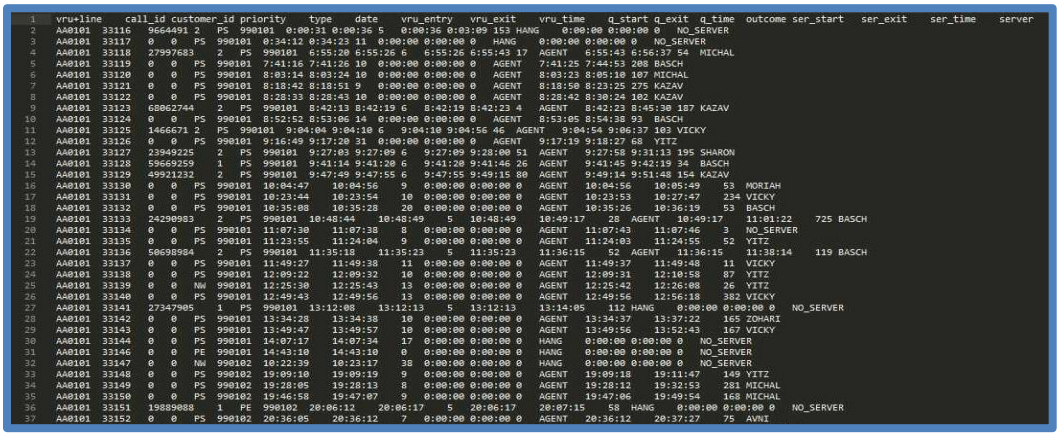
CHAPTER 4

DESIGN, SIMULATION AND IMPLEMENTATION RESULTS

In this chapter, we will be presenting the results obtained so far during the course of this senior project both from simulation and implementation perspectives as the final report (thesis) should be self-contained.

4.1: FINALIZED SYSTEM MODEL: REAL CALL CENTER

In order to better understand the real call center, we received the data from materials of a course entitled Service Engineering, taught by Professor Avishai Mandelbaum at the Technion, Israel [9]. The records are shown in Figure 26. It emphasized, theoretically, on Queuing Theory, and, practically, on Telephone Call Centers and Service Networks. We extracted a one-day data, sorted its respected fields, and based our simulations upon them. This worked as both a limitation, in terms of more close approximation, and an extension, to match theoretical results for our proposed model to that of theoretical models.



vru+line	call_id	customer_id	priority	type	date	vru_entry	vru_exit	vru_time	q_start	q_exit	q_time	outcome	ser_start	ser_exit	ser_time	server
AA0101	33116	9664491	2	PS	990101	0:00:31	0:00:36	5	0:00:36	0:03:09	153	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33117	0	0	PS	990101	0:34:12	0:34:23	11	0:00:00	0:00:00	0	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33118	27997683	2	PS	990101	6:55:20	6:55:26	6	6:55:26	6:55:43	17	AGENT	6:55:43	6:56:37	54	MICHAL
AA0101	33119	0	0	PS	990101	7:41:16	7:41:26	10	0:00:00	0:00:00	0	AGENT	7:41:25	7:44:53	208	BASCH
AA0101	33120	0	0	PS	990101	8:03:14	8:03:24	10	0:00:00	0:00:00	0	AGENT	8:03:23	8:05:10	107	MICHAL
AA0101	33121	0	0	PS	990101	8:18:42	8:18:51	9	0:00:00	0:00:00	0	AGENT	8:18:50	8:23:25	275	KAZAV
AA0101	33122	0	0	PS	990101	8:28:33	8:28:43	10	0:00:00	0:00:00	0	AGENT	8:28:42	8:30:24	102	KAZAV
AA0101	33123	68062744	2	PS	990101	8:42:13	8:42:19	6	8:42:19	8:42:23	4	AGENT	8:42:23	8:45:30	187	KAZAV
AA0101	33124	0	0	PS	990101	8:52:52	8:53:06	14	0:00:00	0:00:00	0	AGENT	8:53:05	8:54:38	93	BASCH
AA0101	33125	1466671	2	PS	990101	9:04:04	9:04:10	6	9:04:10	9:04:56	46	AGENT	9:04:54	9:06:37	103	VICKY
AA0101	33126	0	0	PS	990101	9:16:49	9:17:20	31	0:00:00	0:00:00	0	AGENT	9:17:19	9:18:27	68	YITZ
AA0101	33127	23949225	2	PS	990101	9:27:03	9:27:09	6	9:27:09	9:28:00	51	AGENT	9:27:58	9:31:13	195	SHARON
AA0101	33128	59669259	1	PS	990101	9:41:14	9:41:20	6	9:41:20	9:41:46	26	AGENT	9:41:45	9:42:19	34	BASCH
AA0101	33129	49921232	2	PS	990101	9:47:49	9:47:55	6	9:47:55	9:49:15	80	AGENT	9:49:14	9:51:48	154	KAZAV
AA0101	33130	0	0	PS	990101	10:04:47	10:04:56	9	0:00:00	0:00:00	0	AGENT	10:04:56	10:05:49	53	MORIAH
AA0101	33131	0	0	PS	990101	10:23:44	10:23:54	10	0:00:00	0:00:00	0	AGENT	10:23:53	10:27:47	234	VICKY
AA0101	33132	0	0	PS	990101	10:35:08	10:35:28	20	0:00:00	0:00:00	0	AGENT	10:35:26	10:36:19	53	BASCH
AA0101	33133	24290983	2	PS	990101	10:48:44	10:48:49	5	10:48:49	10:49:17	28	AGENT	10:49:17	11:01:22	725	BASCH
AA0101	33134	0	0	PS	990101	11:07:30	11:07:38	8	0:00:00	0:00:00	0	AGENT	11:07:43	11:07:46	3	NO_SERVER
AA0101	33135	0	0	PS	990101	11:23:55	11:24:04	9	0:00:00	0:00:00	0	AGENT	11:24:03	11:24:55	52	YITZ
AA0101	33136	50698984	2	PS	990101	11:35:18	11:35:23	5	11:35:23	11:36:15	52	AGENT	11:36:15	11:38:14	119	BASCH
AA0101	33137	0	0	PS	990101	11:49:27	11:49:38	11	0:00:00	0:00:00	0	AGENT	11:49:37	11:49:48	11	VICKY
AA0101	33138	0	0	PS	990101	12:09:22	12:09:32	10	0:00:00	0:00:00	0	AGENT	12:09:31	12:10:58	87	YITZ
AA0101	33139	0	0	NW	990101	12:25:30	12:25:43	13	0:00:00	0:00:00	0	AGENT	12:25:42	12:26:08	26	YITZ
AA0101	33140	0	0	PS	990101	12:49:43	12:49:56	13	0:00:00	0:00:00	0	AGENT	12:49:56	12:56:18	382	VICKY
AA0101	33141	27347905	1	PS	990101	13:12:08	13:12:13	5	13:12:13	13:14:05	112	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33142	0	0	PS	990101	13:34:28	13:34:38	10	0:00:00	0:00:00	0	AGENT	13:34:37	13:37:22	165	ZOHARI
AA0101	33143	0	0	PS	990101	13:49:47	13:49:57	10	0:00:00	0:00:00	0	AGENT	13:49:56	13:52:43	167	VICKY
AA0101	33144	0	0	PS	990101	14:07:17	14:07:34	17	0:00:00	0:00:00	0	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33146	0	0	PE	990101	14:43:10	14:43:10	0	0:00:00	0:00:00	0	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33147	0	0	NW	990102	10:22:39	10:23:17	38	0:00:00	0:00:00	0	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33148	0	0	PS	990102	19:09:10	19:09:19	9	0:00:00	0:00:00	0	AGENT	19:09:18	19:11:47	149	YITZ
AA0101	33149	0	0	PS	990102	19:28:05	19:28:13	8	0:00:00	0:00:00	0	AGENT	19:28:12	19:32:53	281	MICHAL
AA0101	33150	0	0	PS	990102	19:46:58	19:47:07	9	0:00:00	0:00:00	0	AGENT	19:47:06	19:49:54	168	MICHAL
AA0101	33151	19889080	1	PE	990102	20:06:12	20:06:17	5	20:06:17	20:07:15	58	HANG	0:00:00	0:00:00	0	NO_SERVER
AA0101	33152	0	0	PS	990102	20:36:05	20:36:12	7	0:00:00	0:00:00	0	AGENT	20:36:12	20:37:27	75	AVNI

Figure 26: Call Center Records

Next, we present the strategy used to extract meaningful information from the obtained call center records in factors influencing the choice of final design section.

4.2: FACTORS INFLUENCING THE CHOICE OF FINAL DESIGN

Our initial model was closer to the ideal situation in which the calls arrival follows the Poisson distribution with the fixed arrival rate. However, in case of real call center, the arrival rate is not fixed but it varies with time and also follows a distribution. To make our model more realistic we also added the Virtual Response Unit (VRU) functionality. Also, the calls were routed according to the single routing scheme, which was skill based routing, irrespective of the calls arrival rate, server's experience, call serving speed etc. which resulted in the average performance. In order to increase the performance and to reduce the queue waiting time we devised four different routing schemes, which are: Fastest Server First Routing (FSF), Longest Idle First routing (LIF), Lowest cost first Routing (LCR) and the Skill based routing (SBR). We utilized these routing schemes according to the changing call arrival rate with respect to time.

Moreover, in the results of our initial model, the mean waiting time found from the simulation was deviating from that of the real call center for lesser number of agents, though the results were quite similar for the larger number of agents. Thus, in order to match the results with the real call center, we did the data analysis on the real call center data and found the distributions for each variable like average call time, probability of waiting in the queue, traffic intensity, agent's service level, agent's salary etc. and used these variables for implementing various routing schemes.
In the real call center Arrival, rates are different at different times of day. Hence, we must change the average arrival rate of calls each hour. Callers must specify their required skill using Virtual Response Unit VRU; Callers can abandon Queue after waiting for some time and hang up at any time. The calls are routed based on their required skill. The Agents have inhomogeneous service time distributions and up to six skills. There are 23 agents available in the call center. Their scheduled are dynamic i.e. more agents are called to handle calls when traffic is high. Each Agents has an average call service time of three hours and answers around 75 calls each day. As in Erlang X, Specialists have only one skill and Generalists can have two or more skills. Agents are scheduled depending on Traffic Intensity.
We used the call center data to extract information about agents and callers. First, we used all the calls to extract the Call Durations Distribution. The same distribution can be generated in NS3 using a Random Variable Generator. There were a lot of call drops and call abandonments with a call duration of zero seconds. They were used to estimate the average patience of caller in the queue. The data revealed that the Agents were scheduled dynamically based on the arriving Call traffic. The Queue Waiting Times Distribution revealed that Queue waiting time was higher when call traffic was higher. We also used the Virtual Response Unit Times Distribution to estimate how much time the average caller spends in the VRU. The Distributions were input into the simulator to simulate what happened in the real call center on a particular day.

The Extracted Distributions of Call Durations are given in Figure 27.

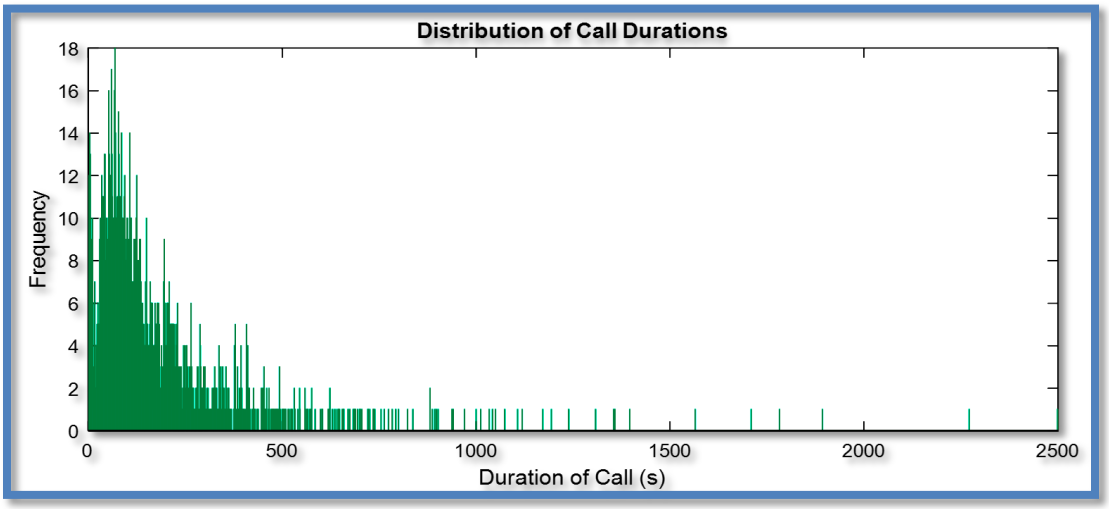


Figure 27: Distribution of Call Durations

The Extracted Distributions of Queue Waiting Times are given in Figure 28.

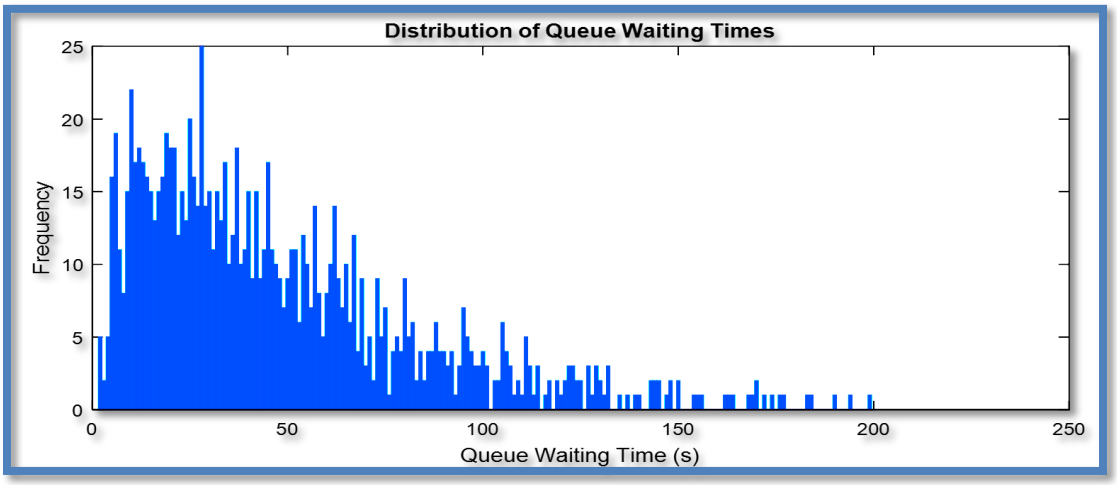


Figure 28: Distribution of Queue Waiting Times

The Extracted Distributions of VRU Durations are given in Figure 29.

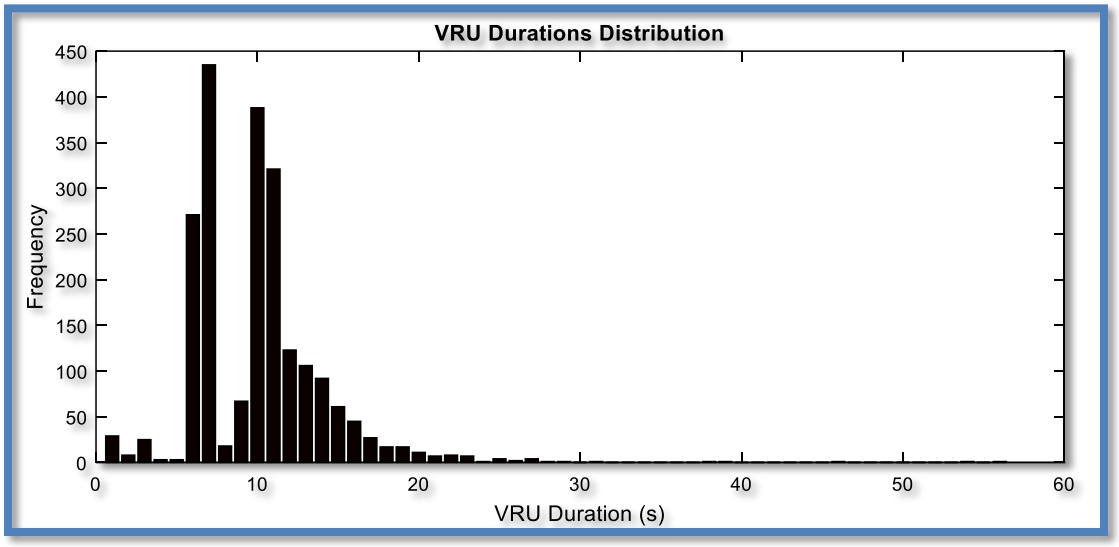


Figure 29: Distribution of VRU Durations

The Extracted Profiles of Agents are given in Table 1.

Agent	Total Answer Time (Hrs.)	Total Calls Answered	Average Service Time (s)	Skills (PE,IN,TT,NE,NW,PS)
GILI	4.032	111	130	001101
SHARON	5.734	154	134	001111
TALI	3.571	63	204	101011
YITZ	5.406	116	167	001011
ANVI	5.748	69	299	000111
YIFAT	4.356	98	160	110001
LORI	5.700	76	270	010011
TOVA	5.318	129	148	100111
KAZAV	5.646	133	152	001111
ELI	3.087	21	529	000100
MICHAL	4.278	91	169	100011
ZOHARI	2.302	25	331	000101
MIKI	4.614	113	146	101011
NAAMA	4.429	112	142	001101
ANAT	4.281	82	187	010111
IDIT	4.386	131	120	101011
DORIT	3.574	63	204	001001
SHLOMO	1.592	56	102	010011
BENSION	3.308	70	170	010011
GELBER	1.577	39	145	001011
MORIAH	4.601	89	186	100111
PINHAS	0.668	10	240	000011
Total	88.208	1491		

Table 1: Profiles of Agents

4.3: FINAL SYSTEM DIAGRAM

In our final design, the calls arrive according to the Poisson distribution with the variable arrival rate, which keeps, on changing with time. There is also the virtual response queue in which all the calls wait for some time specifying their requirements. The agent nodes are characterized based on their exhaustion level, efficiency level, training level, age, experience, leaves, breaks, schedules, speed, and learning, Calls Serviced, Mean Service Time, and Quitting Probability. Where the distribution of all such variables are drawn from the real call center data. In the final design, the basic routing was done based on the agent's skills and the caller's requirements. However, according to the variable arrival rate, dynamic routing was utilized. FSF was used when the calls arrival rate was higher so that more calls could be served and the calls drop rate would be small. In case of moderate arrival, rate lowest cost first routing was used to optimize the cost of running the call center. LIF was used when the arrival rate was small so that the non-experienced agents could gain experience by handling more calls. Upon Arrival, the Caller must enter the Virtual Response Unit Linked List where it is assigned a VRU Counter that decrements every second. When the VRU Counter hits zero, the Caller enters the Callers Queue if Agent with the required skill is busy. When the Agent becomes idle, the packet is routed to him. His job is to service the packet for a random period dictated by a random distribution with a mean value depending on the Agent's Experience. The Agents are paid on hourly basis and their scheduling is done based on the hourly traffic intensity. The Call Centre is shown in Figure 29, the Virtual Response Unit is shown in Figure 30 and the Queue is shown in Figure 31.

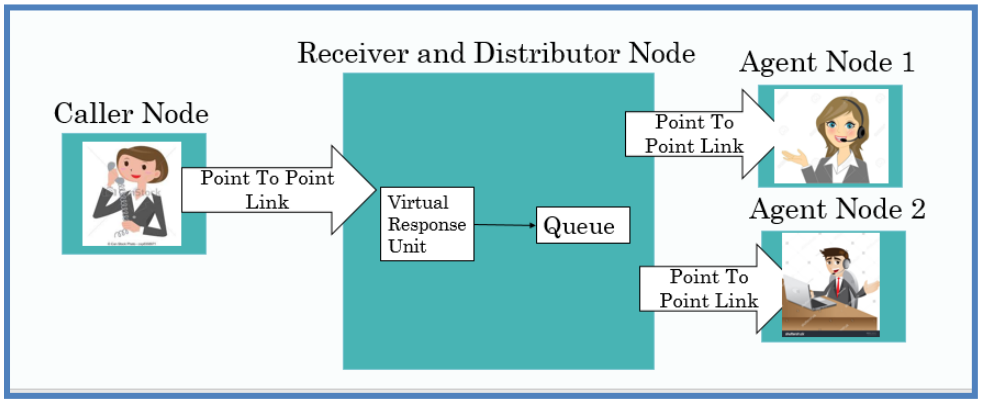


Figure 30: Call Center Block Diagram

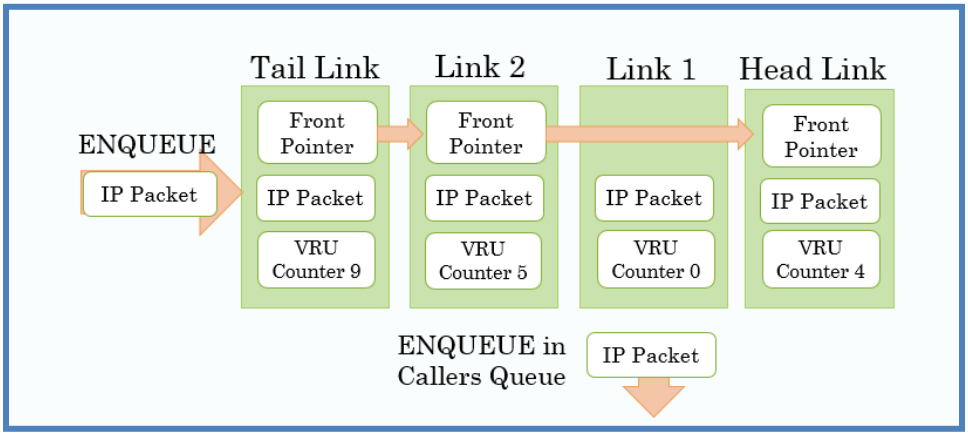


Figure 31: Virtual Response Unit Linked List

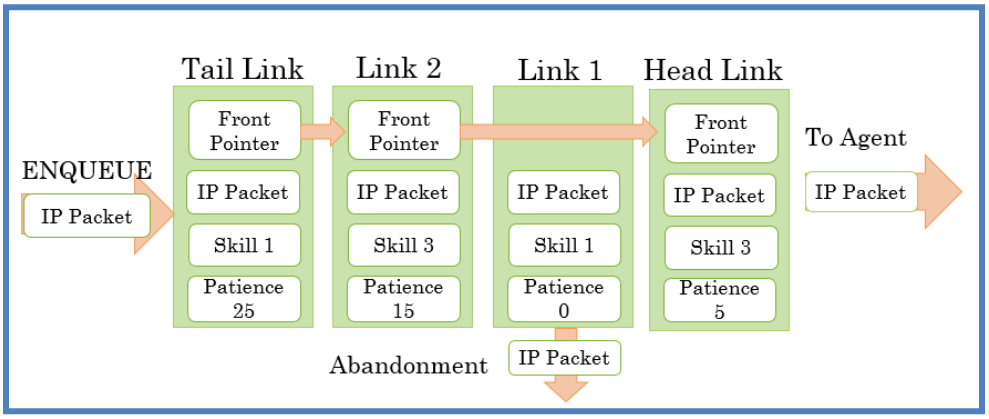


Figure 32: Callers Queue Linked List

4.4: Simulation Methodology

The technique that we used was that using the Agent profiles extracted earlier, we programmed the agents to have similar properties as the real agents. Then we generated traffic similar to the actual traffic and kept a log of all the calls in the simulation including: Packet Size, skill, VRU Time, Patience, Id, EnqueueTime, Dequeue Time, Drop Time, Abandonment Time, Agent Id, Agent Wage, Service Time and used this data to calculate important performance metrics of the call center and to compare results with real data.
We made a primary Virtual Response Unit Linked List that comes before Callers Queue. For that, we added a Virtual Response Unit counter in each Link. We decrement Virtual Response Unit Counter of each Link after every second and if Virtual Response Unit Timer hits zero, we remove the Link from Virtual Response Unit Queue and Enqueue in Callers Queue. Before Enqueue, we check if queue is full. If queue is full, packet is dropped.

```
//Whenever a packet is received at receiver node, we Enqueue it in VRU Queue. Each Link has a Packet Pointer, VRU Timer, EnqueueTime, Front and Back Link Pointers.
void VRUEnqueue (Ptr<Packet> packet, uint32_t VRUTime)
{…
 Link* p              =new Link;
 p->VRUtime       =VRUTime;
 p->pkt              =packet;
 p->EnqueueTime =Time (Seconds (Simulator: Now (). Get Seconds ()));
 Link * t             =Tail->Front;
 Tail->Front       =p;
 p->Front           =t;
 t->Back            =p;
 p->Back            =Tail;
 NumberOfPacketsInQueue ++;
 Total Packets     ++;
}
```

```
//This Function is called every second. It decrements the VRU counter of each Link in VRU Queue. If the timer of any Link is zero, the packet in that link is removed and we Enqueue it in Caller's Queue.
void VRUCountDown (Distributor Queue * queue)
{…
if (NumberOfPacketsInQueue>0)
 {
Link* a =Head;
while (a! =Tail)
{
a =a->Back;
a ->VRUtime= ((a->VRUtime)-1);
 if (a->VRUtime==0)
 {
Link* b          =a->Back;
Link* c          =a->Front;
b->Front         =c;
c->Back          =b;
NumberOfPacketsInQueue--;
queue->Enqueue (a->pkt);
}
}
}
}
```

Each Agent has his own distribution of service times and skills.

```
//SocketRecv is called whenever Agent Receives a Packet for service. This Function generates a random number for Operator Efficiency. Service time duration is average of Packet Size and Operator Efficiency. Agent remains busy for that duration and calls ServiceComplete function after that time to become idle again.
void SocketRecv (Ptr<Socket> socket)
{…
MyPacket      =MySocket->Recv ();
PacketSize    =MyPacket->GetSize ();
OP_Efficiency = EfficiencyGenerator->GetInteger ();
Servicing [AgentNumber] =1;
Time          tnext (Seconds (static_cast<double> ((PacketSize+OP_Efficiency)/2)));
SendEvent     =Simulator: Schedule (tnext, &AgentApp: ServiceComplete, this);
}
```

```
void ServiceComplete ()
{…
 idleArray [AgentNumber] =1;
 Servicing [AgentNumber] =0;
 Packets Served          ++;
 MySocket              ->SetRecvCallback (MakeCallback (&AgentApp: SocketRecv, this));
 Total Wages+=WagePerCall;
}
```

A Caller Node keeps generating IP packets and sends them to the distributor node using a Point-to-Point Link. The Packet length follows a fixed distribution and has a fixed mean value. The Call Skill is denoted by a random variable and is stored in the TOS Tag of the IP Packet. The Caller Patience random variable is stored in the TTL Tag of the IP Packet. This is implemented in NS3 by a function that generates a random variable for Call length, Call Skill and Caller Patience, makes an IP packet of that size, adds the IP Tags and then sends it using a socket. The function then makes a self-call by scheduling the next call after a time interval given by a random number following a negative exponential distribution.

The Distributor Node receives the IP packets and keeps adding them to the tail of the VRU Linked List along with a VRU Timer Number, which is decremented every second. Once VRU counter becomes zero, packet is Enqueued in Callers Queue if the Queue is not completely full. If Queue is full, call is dropped. Caller can either redial or else call is lost. It also checks every second if any Agents are idle and distributes the Packets from the Queue to them. It traverses the entire Queue and checks if the idle Agents have the skill required by the Packet. In NS3, it schedules a call to SendPacket function every second, which does the routing of calls using Point to Point Links to Agents with required skill.

The Agent Node receive the IP packet from the Distributor Node and becomes busy for the duration of the call. Hence, it changes the Boolean idle flag to zero. It schedules a call to ServiceComplete function after a period equal to the length of the call after which it changes its status to idle again. Agents are scheduled and leave service based on schedules.

Now we present the Caller, Distributor and Agent Node Applications designed for this Final Call Center Model framework.

4.4.1: CALLER NODE APPLICATION

The Caller Agent Node Application is shown in Figure 33.

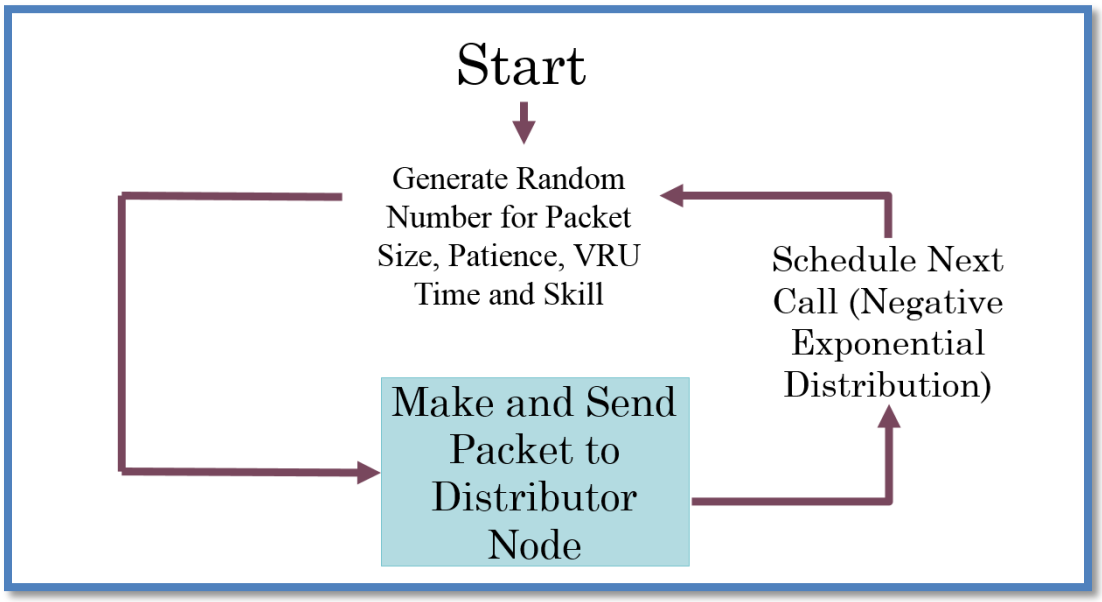


Figure 33: Caller Node Application

```
void SendPacket ()
{…
 Size            =PacketSizeGenerator         ->GetInteger ();
Waitingtime   =WaitingTimeGenerator      ->GetInteger ();
Skill             =SkillTypeGenerator          ->GetInteger ();
Time             =InterArrivalTimeGenerator  ->GetInteger ();          //Generate Random Numbers
Socket->SetIpTos (skill);                                                    //Set Packet Skill Tag
 Socket->SetIpTtl (waitingtime);                                      //Set Packet Waiting Time Tag
 Ptr<Packet> packet=Create<Packet> (size);                                //Make Packet
 Socket->Send (packet);                                                       //Send Packet
Time tnext (Seconds (static_cast<double> (time)));                  //Schedule self-call
SendEvent=Simulator: Schedule (tnext, &CallerApp: SendPacket, this);
}
```

4.4.2: DISTRIBUTOR NODE APPLICATION

The Distributor Node Application is shown in Figure 34.

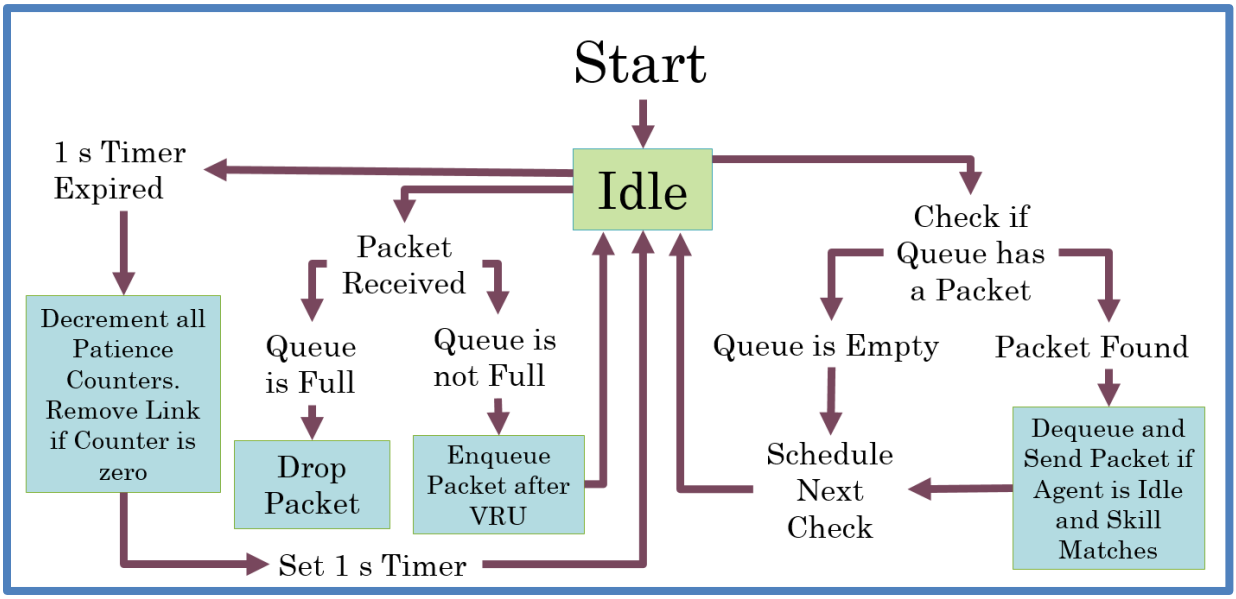


Figure 34: Distributor Node Application

```
void SkillBasedRouting ()
{…
If (idle [PeerNumber] && (queue->NumberOfPacketsInQueue>0))                    //Checks
 { DistributorQueue::Link* a=queue->Head->Back;
 If ((SkillArray [PeerNumber] & a->Skill)>0)                                   //Match Skill
 MySocket                ->Send (a->Packet);                                   //Send Packet
 Time                    tnext (Seconds (1));                                  //Schedule self-call
 SendEvent =Simulator::Schedule (tnext, &DistributorApp: SkillBasedRouting, this);
 }
}
void SocketRecv (Ptr<Socket> socket)
{…Address from;
 Ptr<Packet> packet=MySocket->RecvFrom (from);                                 //Receive Packet
 Queue->Enqueue (packet);                                                      //Enqueue Packet
 MySocket->SetRecvCallback (MakeCallback (&ReceiverApp: SocketRecv, this));
}
```

4.4.3: AGENT NODE APPLICATION

The Agent Node Application is shown in Figure 35.

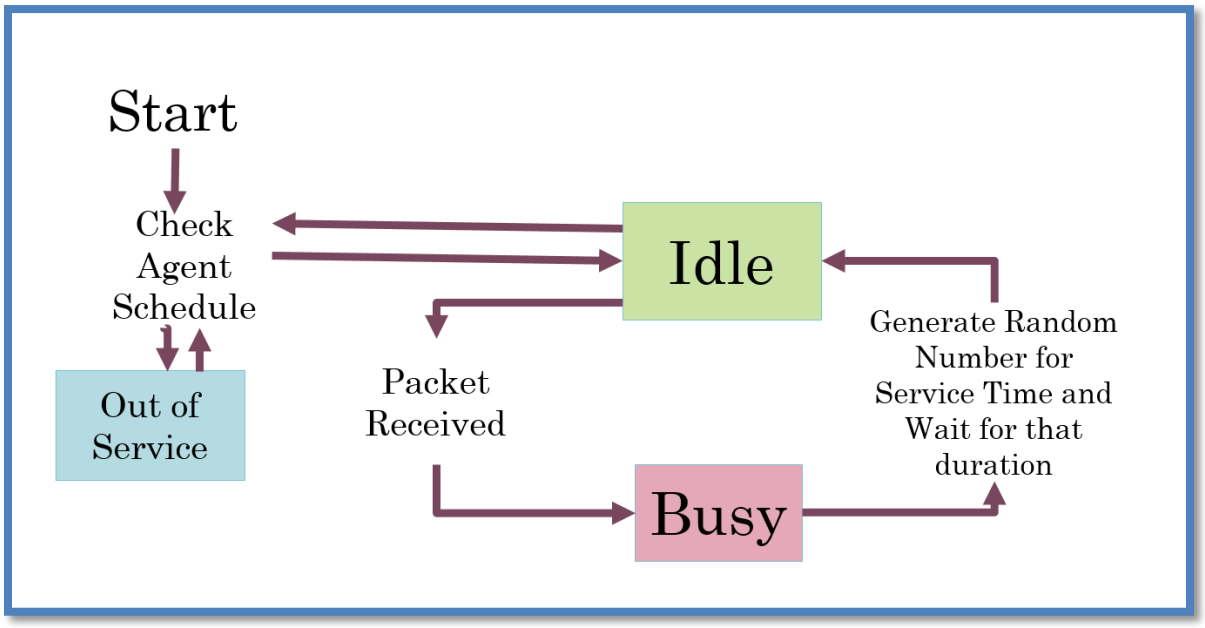


Figure 35: Agent Node Application

```
void SocketRecv (Ptr<Socket> socket)
{…
MyPacket      =MySocket->Recv ();                                    //Receive Packet
PacketSize     =MyPacket->GetSize ();
OP_Efficiency = EfficiencyGenerator->GetInteger ();   //Generate random number for service time
 Time          tnext (Seconds (static_cast<double> ((PacketSize+OP_Efficiency)/2)));
SendEvent      =Simulator: Schedule (tnext, &AgentApp: ServiceComplete, this);//Schedule call to
ServiceComplete Function after Service Time is over
 IdleArray [AgentNumber] =0;                                                    //Busy
 }
void ServiceComplete ()
{… IdleArray [AgentNumber] =1;                                                  //Idle
MySocket ->SetRecvCallback (MakeCallback (&AgentApp: SocketRecv, this));
                                        //Call SocketRecv Function when call arrives
}
```

4.5: DESIGN RESULTS

4.5.1: ERLANG C BLOCK DIAGRAM

The model for simple Erlang C Queue is shown in Figure 36.

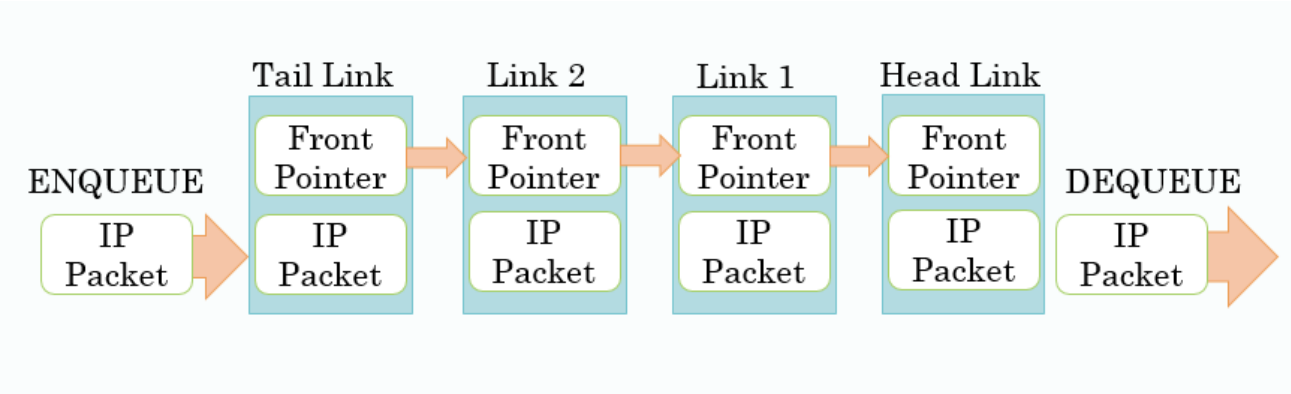


Figure 36: Erlang C Queue

4.5.2: ERLANG X BLOCK DIAGRAM

The model for improved Erlang X Queue is shown in Figure 37.

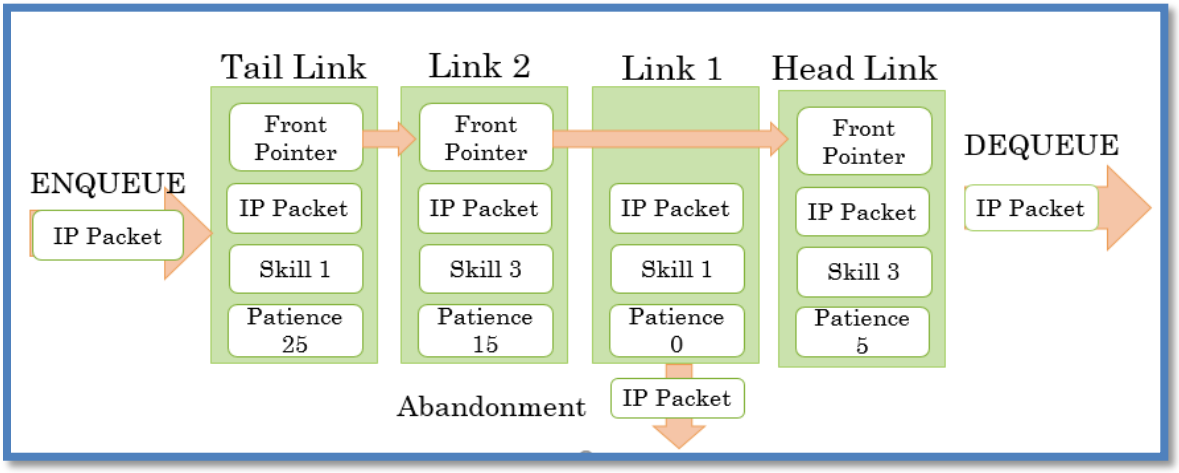


Figure 37: Erlang X Queue with Abandonments

4.5.3: REAL CALL CENTER BLOCK DIAGRAM

The model for Real Call Center VRU Queue and Callers Queue is shown in Figure 39 and 40.

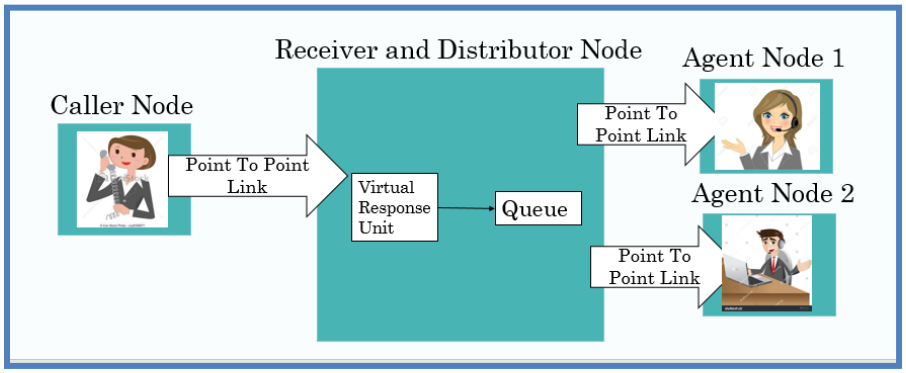


Figure 38: Real Call Center Block Diagram

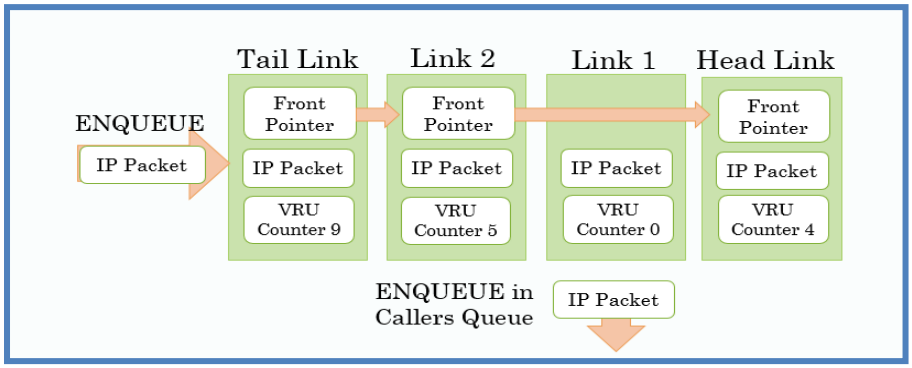


Figure 39: Virtual Response Unit Linked List

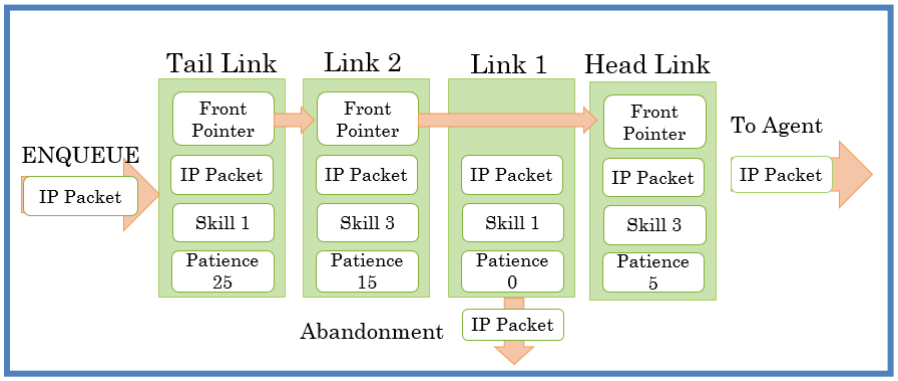


Figure 40: Real Call Center Queue

4.6: SIMULATION RESULTS

4.6.1: ERLANG C

After having discussed the Caller, Distributor and Agent Node Applications, now we will discuss the simulation results and compare them with Erlang C Calculator [2]. The Simulation technique was that we generated 10000 calls and calculated the mean queue waiting time for that run. Then we repeated these 10 times to obtain a 90% confidence interval and mean of distribution for a particular number of agents. The results are given in Table 2. We plotted the simulation results against Erlang C Calculator results [2] as shown in Figure 41.

Number of Agents	***Mean Call Duration***	***Mean Inter-Arrival Time***	***Average Waiting Time (Simulation)***	***Standard Deviation***	***90% Confidence Interval***	***Average Waiting Time (Erlang C Formula)***
4	50.978	15.511	44.272	5.7407	[41.1349, 47.4093]	45.450
5	50.322	15.668	7.8829	1.1426	[7.2585, 8.5073]	8.2108
6	50.583	15.477	2.4085	0.2988	[2.2452, 2.5718]	2.5495
7	50.105	15.539	0.6374	0.1835	[0.5371, 0.7377]	0.6990
8	50.469	15.628	0.1540	0.1097	[0.0941, 0.2139]	0.2056

Table 2: Erlang C Simulation and Calculator Results

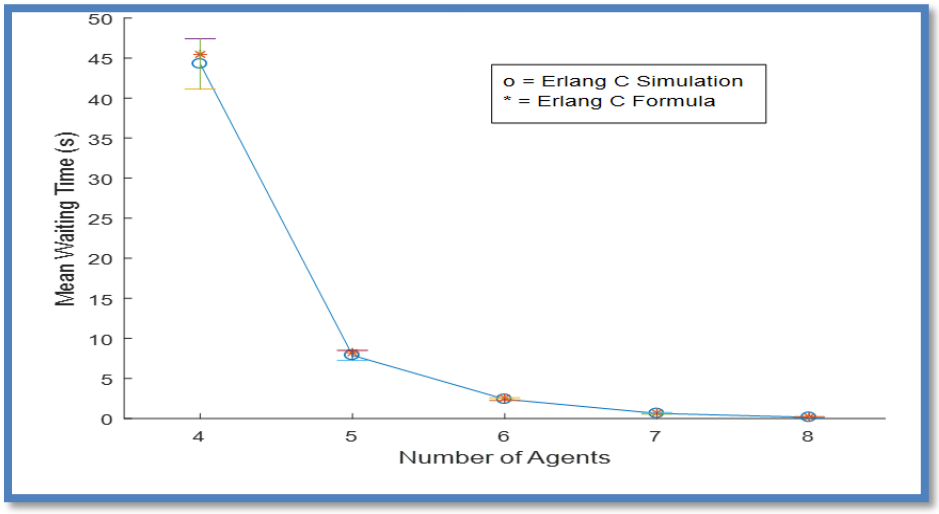


Figure 41: Erlang C Simulation and Calculator Graph

4.6.2: ERLANG X

Our Simulation technique was that we generated 10000 calls and calculated the mean queue waiting time for that run. Then we repeated 10 times to obtain a 90% confidence interval and mean of distribution for a particular number of agents. We plotted the simulation results against Erlang X Calculator results [3] as shown in Table 3 and Figure 42.

Number of Agents	***Arrivals per minute***	***Mean Call Length***	***Average Patience***	***Abandonments Percentage***	***Mean Waiting Time (Erlang X Simulation)***	***99% Confidence Interval***	***Mean Waiting Time (Erlang X Formula)***
7	10.844	50.364	200.37	27.506	38.454	[36.9980, 39.9094]	55.87
8	10.873	50.673	201.70	19.537	26.982	[26.3361, 27.6283]	36.57
9	10.849	50.704	202.81	14.845	19.822	[18.3754, 21.2692]	21.56
10	10.939	50.794	201.45	9.2483	12.649	[11.4348, 13.8612]	12.65
11	10.980	50.433	200.75	5.2236	7.1655	[5.9945, 8.3365]	6.68
12	10.793	50.784	200.50	2.8851	3.7695	[3.0709, 4.4681]	3.31
13	10.966	50.075	199.48	1.5449	2.1767	[1.4433, 2.9101]	1.68

Table 3: Erlang X Simulation and Calculator Results

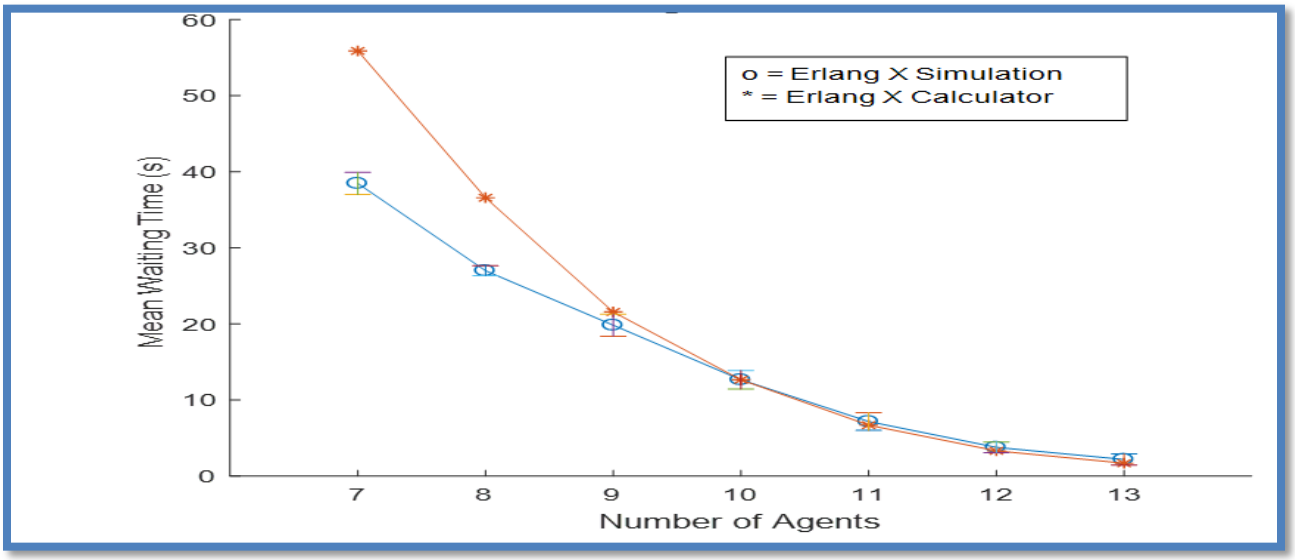


Figure 42: Erlang X Simulation and Calculator Graphs

4.6.3: REAL CALL CENTER

This section discusses the results of Real Call Center Simulation. The technique used was that we generated call traffic similar to real Call Center as studied using Call Center Records [9] and calculated the mean queue waiting time for each hour. Then we repeated these 10 times to obtain a 90% confidence interval and mean of distribution for a particular hour. We plotted the results against Real Call Center results. The Caller Traffic is shown in Figure 43. The Mean Queue Waiting Times for Simulation and Real Call Center are shown in Figure 44.

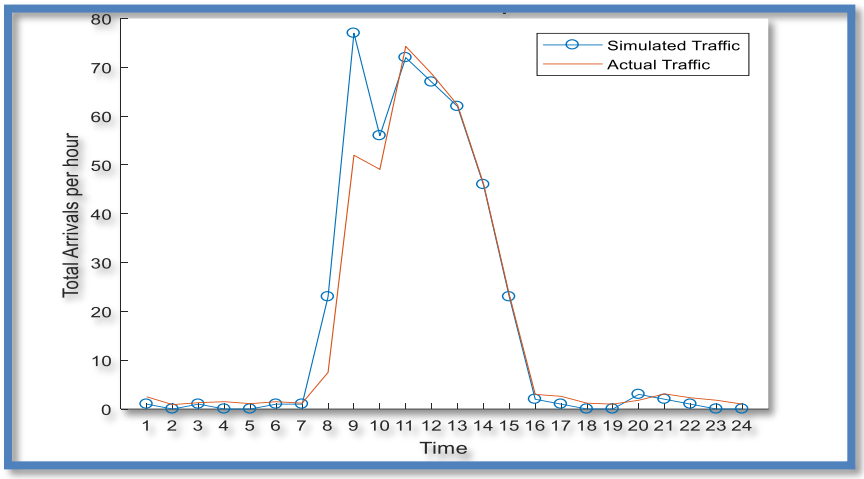


Figure 43: Number of Arivals per Hour

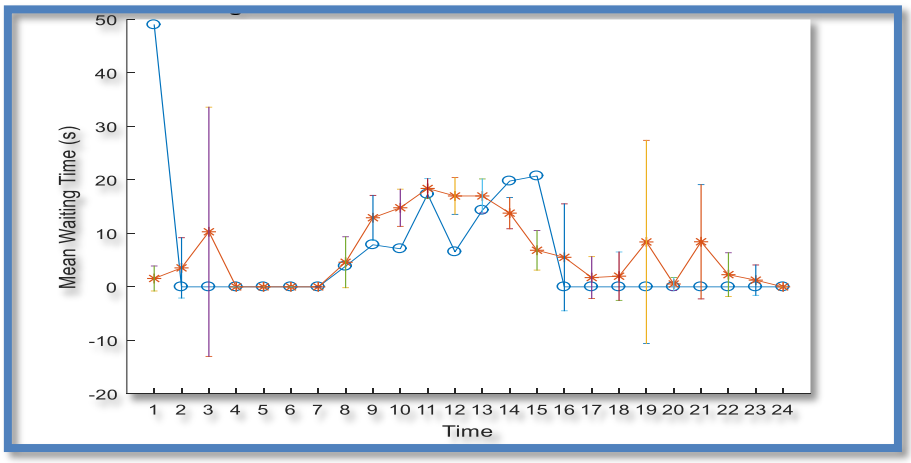


Figure 44: Average Queue Waiting Time: Actual Data vs Simulation Results

Two Sample Simulator Runs are shown in Figures 45 and 46.

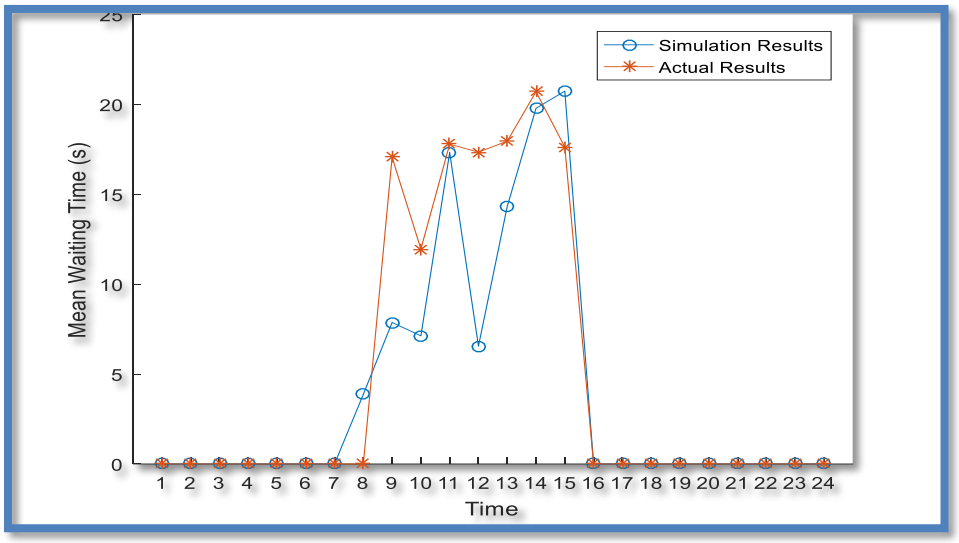


Figure 45: Average Queue Waiting Time: Actual Data vs Simulated Results

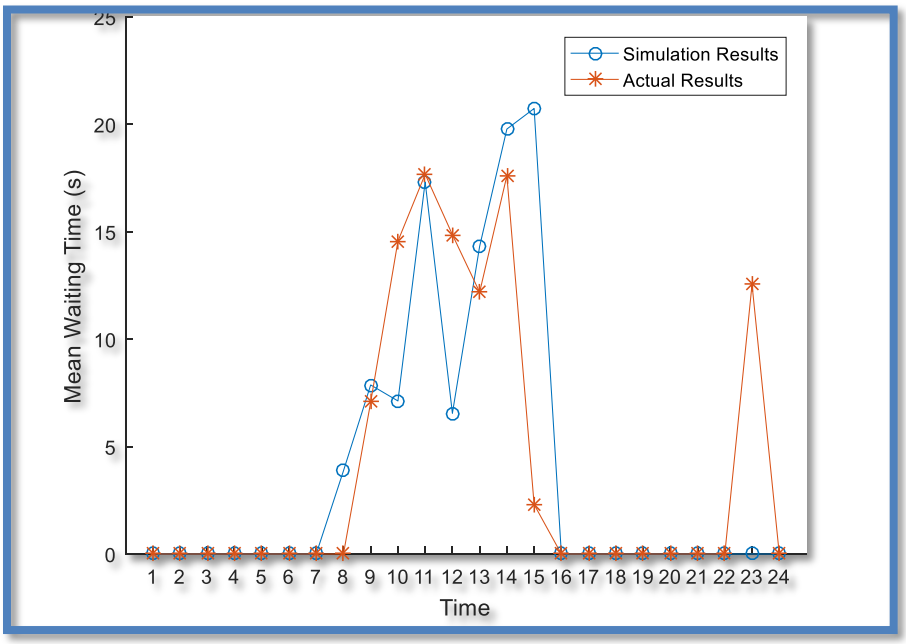


Figure 46: Average Wait Time: Actual Data vs Simulated Results

We studied the real call center records for 1 year which had almost 300000 calls in 354 days. By analyzing every day, we plotted the abandonments vs Number of Agents available in an hour and Number of Calls arriving in the hour as shown in Figure 47. As the number of Agent available increases, the abandonments decrease. When the Number of Calls arriving per hour increase, the abandonments increase.

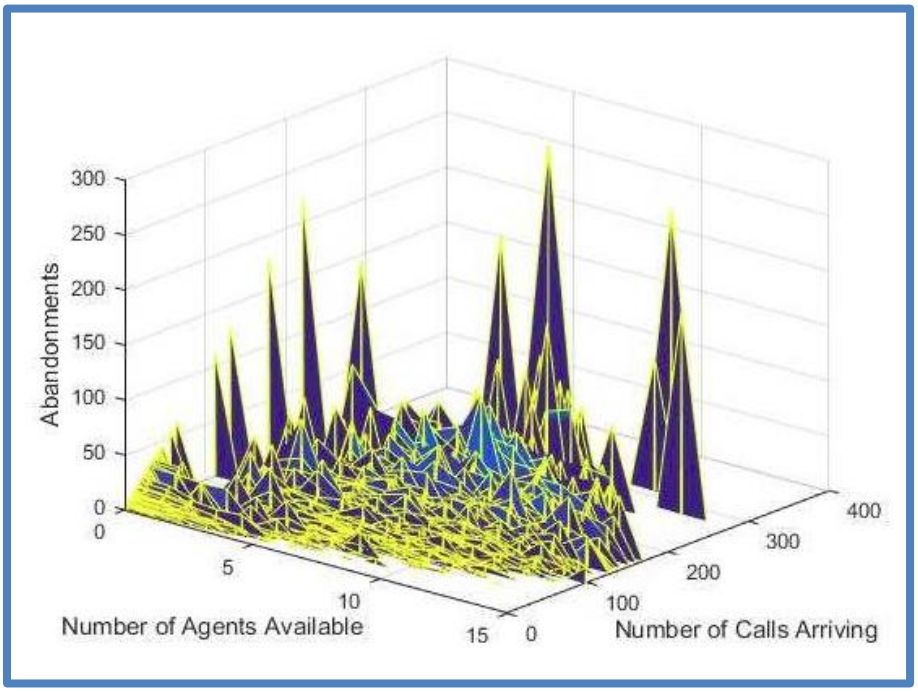


Figure 47: Real Call Center: Abandonment vs Agents Available and Number of Calls Arriving per Hour for 1 year

Then we simulated 1 year of calls in NS-3 with same conditions as in the real call center. The controlled variables included: Scheduling of Agents, Call Traffic, Call Duration Distribution, VRU Duration Distribution, Agent Profiles and Caller Patience Distribution. By analyzing every day, we plotted the abandonments vs Number of Agents available in an hour and Number of Calls arriving in the hour as shown in Figure 48.

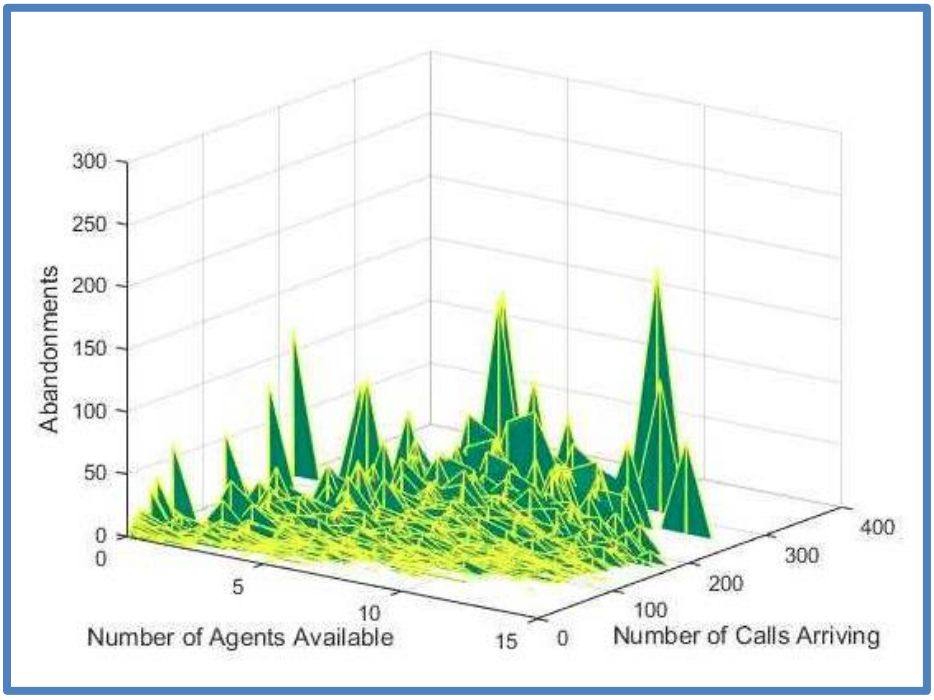


Figure 48: Real Call Center Simulation with Skill Based Routing: Abandonment vs Agents Available and Number of Calls Arriving per Hour for 1 year

In order to compare the performance of NS-3 Simulator with the real Call Center, we calculated the Pearson's Correlation Coefficient for the two distributions. Pearson's Correlation Coefficient is defined as the covariance between two random variables divided by the product of the standard deviations of the random variables. Hence it measures linear dependence between two variables. The Pearson product-moment correlation coefficient will always be between -1 and 1. The closer the value is to either -1 or 1 the more highly correlated the two variables. A correlation coefficient equal to either -1 or 1 indicates a perfect linear relationship between the two variables. A correlation coefficient close to 0 simply indicates that the two variables are not linearly related, however they still may be highly correlated in a nonlinear sense. Hence there is a great disadvantage of using Pearson's Correlation Coefficient.

Pearson's Correlation Coefficient = $\frac{N\sum XY-(\sum X)(\sum Y)}{\sqrt{[N\ \sum(X^2)-(\sum X)\text{^}2][N\ \sum(Y^2)-(\sum Y)\text{^}2]}}$ = 0.917

The Pearson Correlation Coefficient is very close to 1 hence the results are highly positively correlated as evident from Figure 49. Besides Abandonments and Mean Waiting Time, other important performance metrics were also calculated which will be presented in the next section.

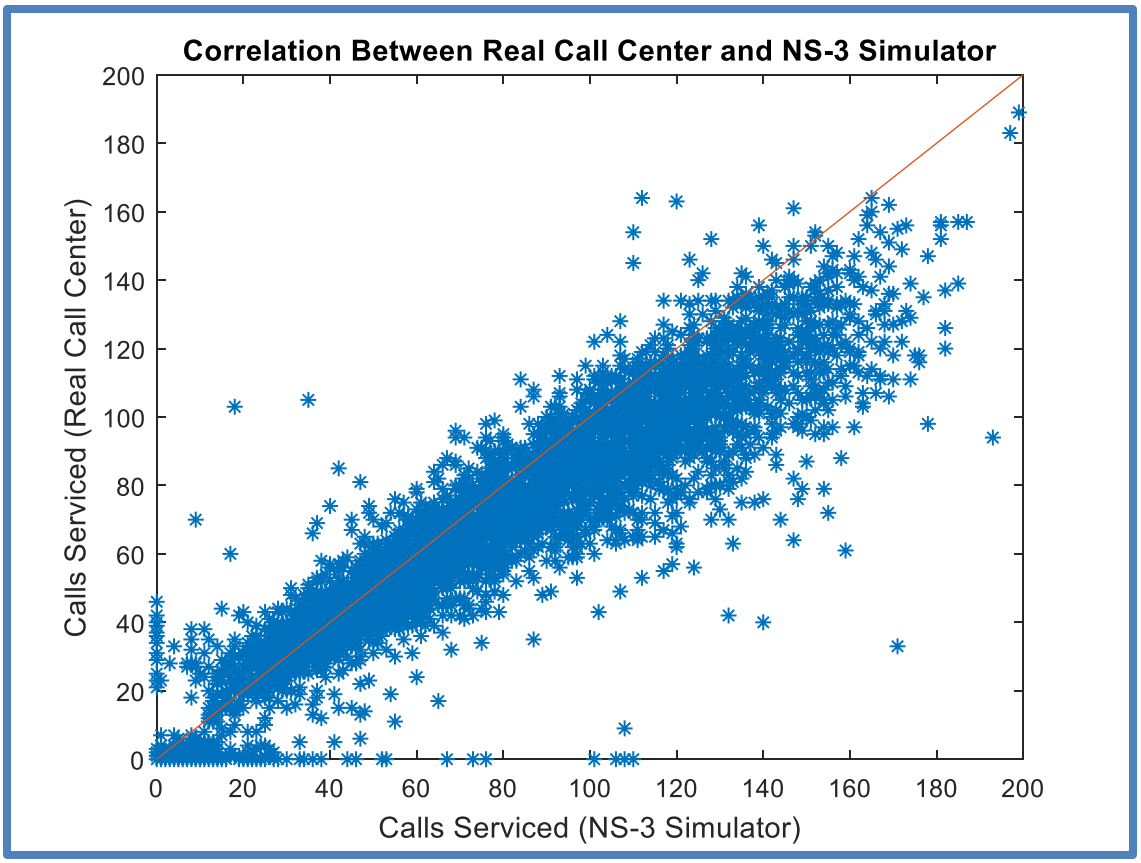


Figure 49: Correlation between Real Call Center and NS-3 Simulation

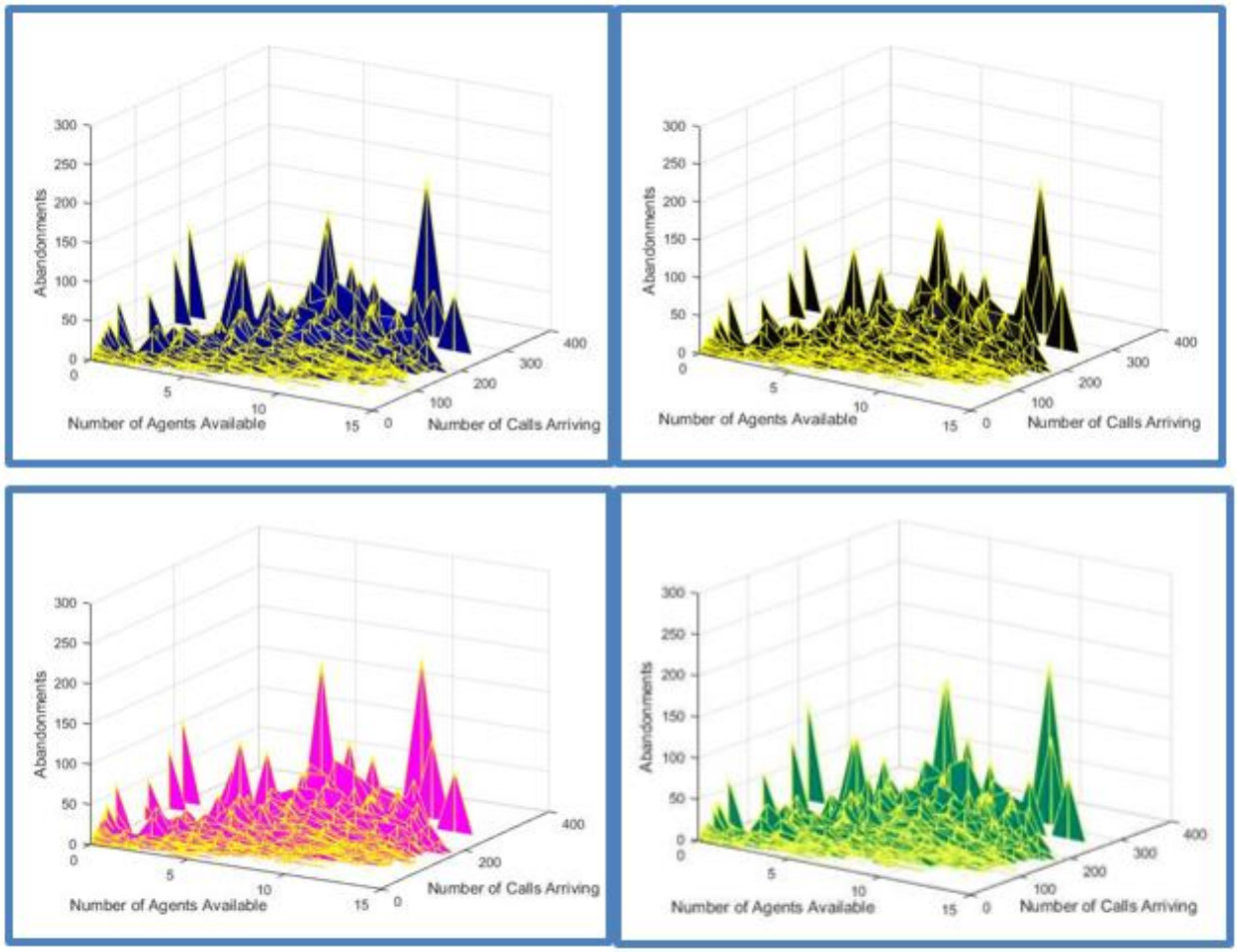


Figure 50: Comparison of Routing Schemes: Fastest Server First (Upper Left), Longest Idle First (Upper Right), Lowest Cost Routing (Lower Left), Skill Based Routing (Lower Right)

4.6.4: DATA ANALYSIS

In addition to the Observations taken above, we kept a log of all the calls in the simulation including: Packet Size, skill, VRU Time, Patience, Id, EnqueueTime, Dequeue Time, Drop Time, Abandonment Time, Agent Id, Agent Wage, Service Time and used this data to calculate important performance metrics of the call center:

1. **Service level of the call**: it is the percentage of calls answered in less than specific time limit.
2. **Total Problem Resolution**: This was found by calculating the number calls resolved initially to those of the incoming calls that is the percentage of calls resolved in the first attempt.
3. **Agent Salaries** (hourly basis) depending on their performance
4. **Net Promoter Score**: this was calculated in response to every call, so was used to determine the performance of our call center system overall. Under this system every call received resulted in an increment in the NPS by 10, if dropped contributed nothing to it, if abandoned resulted in an increment by 10 and if was answered the NPS was further incremented by 10.
5. **Quality Scores**: these were agent rating done by analyzing and calculating the speed of answer of each agent and also the frequency of calls a specific agent answers in a given period of time.
6. **Profits**: this was calculated by finding the ratio of the calls received in a given interval of time to the agent's hourly salary.
7. **Agents utilization** (percentage of time the agents are busy)
8. **Average Caller Time in System** : sum of the queuing delay and the call time for each call.
9. **Probability of wait** : this was calculated by considering the arrival of calls when queue is not empty.
10. **Traffic Intensity** (Arrival rate / Service Rate)

The Calculated Performance metrics for a sample run are shown in Table 4.

Performance Metric	Value
Arrivals Per Minute	1.038
Mean Packet Size	169.577s
Blocking Percentage	0%
Average Patience	785s
Abandonments Percentage	0.268276%
Mean Waiting Time	0.131809s
Call Resolution	1487/1489
Net Promoter Score	71.8521
Agents Utilization	15.343%
First Call Resolution	100%
Average Time in System	174s
Total Wages	Rs.37489
Quality Scores	1483
Waiting Probability	0.402414%
Service Level	99%
Traffic Intensity	2.9337

Table 4: Real Call Center Simulation: Calculated Performance Metrics

We verified the call center model and we could guage the different performance metrics, so we attempted to improve the performance by using four different routing schemes.

4.6.5: VARIOUS ROUTING SCHEMES

1. Fastest Server First (FSF): calls are routed to the idle agents having high speed.
2. Longest Idle First (LIF): calls are routed to the agents who are idle for longest time.
3. Lowest Cost Routing (LCR): calls are routed to the idle agents having lowest wage.
4. Skill Based Routing (SBR): calls are routed to the idle agents having required skills.

The schemes are compared in Table 5 and the results are plotted in Figure 51.

Routing Scheme	Abandonments Percentage	Net Promoter Score	Utilization of Agents	Quality Score	Probability of waiting	Calls Serviced	Total Wages
Longest Idle First Routing (Default)	68.4105%	1.59518	4.85981%	-549	18.7123%	471/1491	Rs.12183
Fastest Server First Routing	5.96915%	68.5756	14.4659%	1313	22.1328%	1402/1491	Rs.33910
Skill Based Routing	0.268276%	71.8521	15.3430%	1483	0.402414%	1487/1491	Rs.37489
Lowest Cost Routing	0.201207%	71.8891	15.3533%	1485	0.201207%	1488/1491	Rs.37148

Table 5: Comparison of Routing Schemes

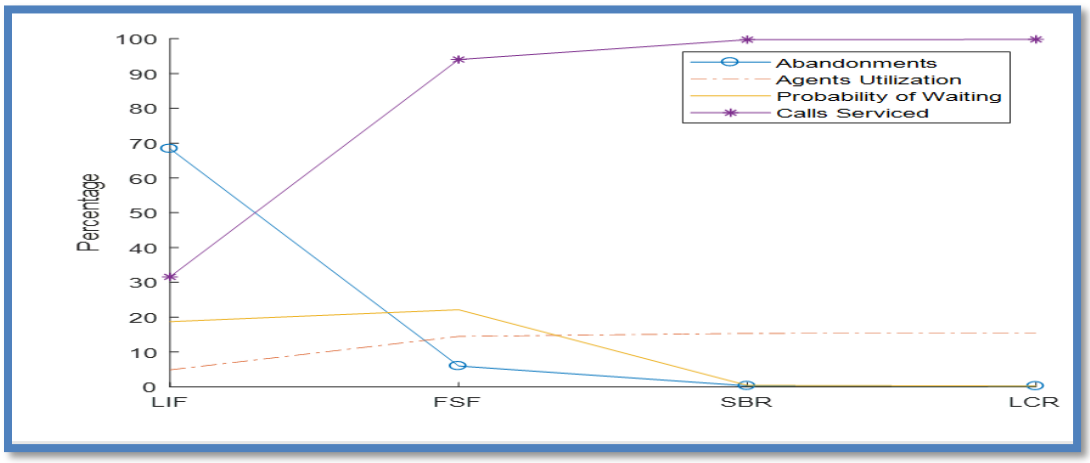


Figure 51: Comparison of Routing Schemes

4.7: IMPLEMENTATION RESULTS

4.7.1: NODE.JS CALL CENTER

The Node.js Call Center implementation differs from the NS-3 Call Center simulation because the network performance degrades in a non-linear manner as the number of Callers in the system increases but the Simulator does not model this effect. By varying the numbers of Active Callers in the System, we were able to determine the degradation in the Quality of Service of the network. Figure 52 shows that the time taken to send a 5 MB file increases greatly as the number of active Agent-Caller Peers increases from 2 (970 ms) to 8 (2890 ms). The internet speed is also variable and data transmission is sporadic.

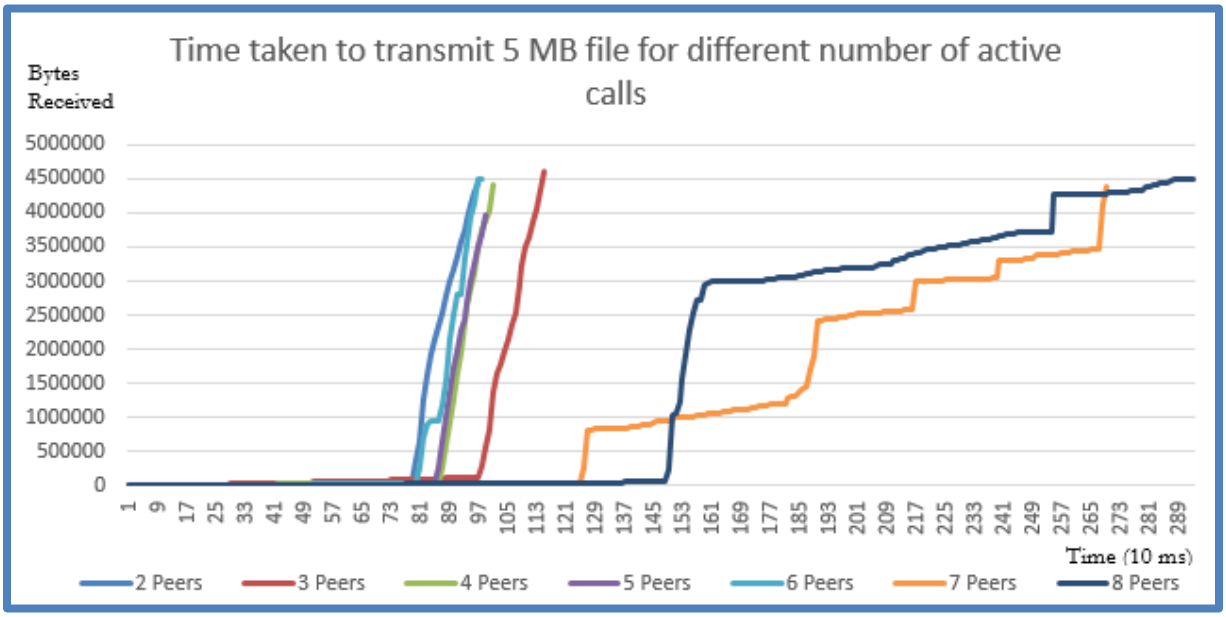


Figure 52: Degradation of Network Performance with increase in Traffic

Call Center Simulation in NS-3 and Call Center Application in Node.js

We simulated 115 calls of average duration 86 seconds in Node.js Call Center for 1 hour and studied the performance metrics. There were 3 Agents present. Then we carried out the simulation in NS-3 to compare the performance of the two Call Centers. The results are shown in Table 6 for comparison.

Performance Metric	Node.js Call Center	NS-3 Call Center
Simulation Time	3600s	3600s
Arrival Rate (Arrivals Per Minute)	1.917	2.0
Inter Arrival Time	31.304s	30.19s
Mean Packet Size	86.446s	88.275
Total Calls Serviced	96	90
Blocks	1	3
Blocking Percentage	2.128%	2.5%
Total Queue Time	3611.134s	3779s
Abandonments	8	26
Abandonments Percentage	8.1633%	21.667%
Mean Waiting Time	37.615s	31.4948s
Call Resolution	96/115	90/120
Net Promoter Score	67.555	56.48
Total Wages	Rs.67200	Rs.64800
Quality Scores	96	61
Service Rate (Calls Serviced per second)	0.0182/s	0.025/s
Traffic Intensity (Arrival Rate/ Service Rate)	1.1104	1.333

Table 6: Comparison of Node.js Call Center Performance with NS-3 Simulation

4.8: DISCUSSION

Relative Ranking of Design Options

After matching the results to that of the real call center, our main goal was to reduce the mean waiting time in the queue in order to increase the performance of the call center. For this, we used different routing schemes according to the changing calls arrival rates. The relative ranking of these schemes are as follows.

- Amongst four different routing schemes, the lowest Cost Routing (LCR) gave the best results. In this routing scheme, calls are routed to the idle agents having lowest wage. It is used when the calls arrival rate is highest. It reduces the cost of paying the agents thus the Cost of running the call center can be optimized by using this routing, since expenses are minimized. LCR not only reduces the abandonment percentage and probability of waiting but it increased the utilization of agents and the number of calls being served.
- The second-best Routing scheme is the skill based routing, which we were already using in our initial model. According to this routing method, calls are routed to the agent whose skills are matched to the caller requirements. It gave nicer results as abandonment percentage and probability of waiting was lower.
- Fastest Server First (FSF) is on the third number. According to this routing scheme, the calls are routed according to the agent's serving speed and it is used when the calls arrival rate is high so that more calls could be answered in lesser time. However, the abandonment percentage and probability of waiting was slightly higher than the other schemes.
- Longest Idle First (LIF) is the worst among all the routing schemes. It is utilized when the arrival rate is very low and calls are routed to the agent who has been idle, waiting for the call, for the longest time. It gave the worst results.

Now we will present a Cost Analysis of our Project.

CHAPTER 5

COST ANALYSIS

We did not incur any cost in this project since all our work was based on coding in NS-3 and Node.js which are open source softwares.

In the next section, we will discuss the societal relevance of our project.

CHAPTER 6

SOCIETAL RELEVANCE

The Call Center Simulation in NS-3 is highly relevant for the society because it can be used in several Queuing systems like Call Centers, Shopping malls, Train Stations, Banks, hospitals etc. to improve the different performance metrics. The simulation can be run very quickly hence it saves time and effort of running analytical experiments. The Node.js Call Center Application is very useful for education, marketing and online advertising, medical services etc. The Call Center can be used to reach any type of audience without incurring any cost since the online calls are free.

Now we will discuss the aspect of environmental impact and sustainability.

CHAPTER 7

ENVIRONMENTAL IMPACT AND SUSTAINABILITY

As the worldwide service industry increases, most countries have seen a significant rise in the number of call centers operating within their borders. While appearing to be a very clean environment to work in, and with less environmental impact than traditional manufacturing industries, call centers nonetheless can cover large floor-space areas, employ hundreds or even thousands of people, and consume resources accordingly. Therefore, it stands to reason that the environmental footprint may be heavier than first imagined, and indeed ISO 14001 compliance and accreditation are becoming more of a requirement for these facilities, especially given that in many cases the call centers themselves may be servicing calls regarding products or services on behalf of organizations that do have a large environmental impact, such as electronics, mobile telephones, or even utility supplies.

However, shifting the call center environment online, using Wi-Fi connectivity, the physical environmental load of utilities of hardware, electricity and energy can be limited. Cloud technology can also be used for scalability or as need per requirement. Some of the benefits of cloud computing is that it reduces reliance on hardware and infrastructure, reducing maintenance cost and lowering office carbon emissions, fewer machines means less power consumption. Cloud computing if combined with Software as a service (SAAS) payment model allows call centers to only pay for what they will be using, this is a huge advantage when call centers experience spikes in usage or traffic. Thus, no server must be 24/7 running, we can scale system up and down and only pay for use to add real cost reduction benefit. Cloud Computing is also allowing businesses to transform their working environments. In many cases staff can work remotely, can access shared resources and can build in flexibility to their day to day work. This has another potential positive green effect. With fewer staff commuting to one single place of work the overall carbon footprint of the business can be lowered significantly. Lastly, cloud computing has made expensive and large inaccessible resources available. Our call center model is an optimum idea for utilizing minimum resources. However, environmental friendly policies can add value to the overall spending and help preserve resources.

After discussing all these dimensions of our project, next chapter will conclude our report and provide future recommendations.

CHAPTER 8

CONCLUSION AND FUTURE RECOMMENDATIONS

We modeled a real-life call center in NS-3 using a packet switched network where duration of calls was equal to length of the packet. If no agents were idle, packets were stored in a Linked List Queue. By introducing different Routing schemes like Fastest Server First, Lowest Cost Routing and Longest Idle First Routing, we were able to improve the call center performance as evident by the significant improvement in observed performance metrics. Further work is required to tweak NS-3 Simulation to enable accurate simulation of different Call Centers and Queuing Models by including traffic dependent propagation delays to incorporate network congestion, costumer experience based routing, Quality of Service measurements, sporadic traffic and heavy traffic conditions.

In order to find the deficiencies of our Call Center Simulation, we made an internet-based call center application in Node.js which allows users to connect their internet devices to our HTTPS Server using an IP Address and Port, even though firewalls and NATs. Interestingly, the network performance degraded greatly when the number of active calls increased and the Quality of Service was greatly degraded under heavy traffic conditions. The Call Center Web Pages often take very long to render hence the call center performance is very unpredictable. Often the Agents are available but they cannot attend waiting calls because the server is not responding or the Internet speed is very low. The transmission rate was also sporadic unlike the NS-3 Model which assumed that the packets were transmitted at constant bit rate with fixed propagation delay. The variable bit rates in Node.js Application communication could be simulated in NS-3 Simulation to make a more accurate Call Center Model. In order to benefit the community, we could obtain Certification to expand the reach of Web Server so that it can be deployed for education, businesses, marketing, hospitals etc.

REFERENCES

[1] "Call Centre." *Wikipedia*, Wikimedia Foundation, 17 Sept. 2017, en.wikipedia.org/wiki/Call_centre. Accessed Sept. 2017.

[2] "Erlang C Calculator." *Call Center Optimization*, www.gerkoole.com/CCO/erlang-c.php. Accessed Sept. 2017.

[3] "Erlang X Calculator." *Call Center Optimization*, www.gerkoole.com/CCO/erlang-x.php. Accessed Sept. 2017.

[4] Gans, Noah, et al. *Telephone Call Centers: Tutorial, Review, and Research Prospects*. 2nd ed., vol. 5, Amsterdam, Netherlands, 2003.

[5] Iversen, Villy B. *Teletraffic Engineering*. Lyngby, Denmark, 2001.

[6] Jerry, et al. *Discrete-Event System Simulation*. 4th ed., Prentice Hall.

[7] Koole, Ger. *Call Center Mathematics*. Amsterdam, Netherlands, 2007.

[8] Keshav, Srinivasan. *Mathematical Foundations of Computer Networking*. 2012.

[9] Mandelbaum, A. Call Center Data. http://iew3.technion.ac.il/serveng/callcenterdata/index.html. 2002

[10] *Ns-3 Tutorial*. 2017, www.nsnam.org/docs/release/3.26/tutorial/ns-3-tutorial.pdf. Accessed Sept. 2017.

[11] *Ns-3 Documentation*, www.nsnam.org/docs/release/3.26/doxygen/index.html. Accessed Sept. 2017.

[12] Perros, Harry. *Computer Simulation Techniques: The definitive introduction!* Raleigh, North Carolina, 2009.

[13] Robbins, Thomas R. *Experience-Based Routing in Call Center Environments*. 2nd ed., vol. 7, Greenville, North Carolina, 2015.

[14] Robbins, Thomas R., et al. *Evaluating the Erlang C and Erlang A Models for Call Center Modeling Working Paper*. Greenville, North Carolina.

[15] "Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols." *IETF Tools*, tools.ietf.org/html/rfc5245.

[16] "Javascript Session Establishment Protocol." *IETF Tools*, tools.ietf.org/html/draft-uberti-rtcweb-jsep-02.

[17] "Navigator." *MDN Web Docs*, developer.mozilla.org/en-US/docs/Web/API/Navigator.

[18] “Node.js v8.11.1 Documentation.” *Index | Node.js v8.11.1 Documentation*, nodejs.org/dist/latest-v8.x/docs/api/.

[19] Npm Documentation, docs.npmjs.com/.

[20] “Privacy Indicator Requirements.” *Media Capture and Streams*, w3c.github.io/mediacapture-main/getusermedia.html.

[21] “RTCDataChannel.” *MDN Web Docs*, developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel.

[22] “RTCPeerConnection.” *MDN Web Docs*, developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection.

[23] “Rtcweb Status Pages.” *IETF*, tools.ietf.org/wg/rtcweb/charters.

[24] “Session Description Protocol.” *Wikipedia*, Wikimedia Foundation, 10 Apr. 2018, en.wikipedia.org/wiki/Session_Description_Protocol.

[25] “Session Traversal Utilities for NAT (STUN).” *IETF Tools*, tools.ietf.org/html/rfc5389.

[26] Socket.IO - Docs, socket.io/docs/.

[27] “TCP Candidates with Interactive Connectivity Establishment (ICE).” *IETF Tools*, tools.ietf.org/html/rfc6544.

[28] “Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN).” *IETF Tools*, tools.ietf.org/html/rfc5766.

[29] *WebRTC 1.0: Real-Time Communication Between Browsers*, w3c.github.io/webrtc-pc/

[30] “WebRTC Data Channel Protocol.” *IETF Tools*, tools.ietf.org/html/draft-jesup-rtcweb-data-protocol-01.

[31] “Web Real-Time Communication Use Cases and Requirements.” *IETF Tools*, tools.ietf.org/html/draft-ietf-rtcweb-use-cases-and-requirements-10.

[32] Webrtc. “Webrtc/Samples.” *GitHub*, github.com/webrtc/samples/.

[33] “Exponential Distribution.” *Wikipedia*, Wikimedia Foundation, 10 May 2018, en.wikipedia.org/wiki/Exponential_distribution.

[34] Dutton, Sam. “Getting Started with WebRTC - HTML5 Rocks.” *HTML5 Rocks - A Resource for Open Web HTML5 Developers*, www.html5rocks.com/en/tutorials/webrtc/basics/.

[35] “HTTPS.” *Wikipedia*, Wikimedia Foundation, 20 May 2018, en.wikipedia.org/wiki/HTTPS.

[36] “Node.js Tutorial.” *W3Schools Online Web Tutorials*, www.w3schools.com/nodejs/.

APPENDIX

NS-3 SIMULATION

Figure 53: NetAnim Simulation

ONLINE ERLANG C CALCULATOR [2]

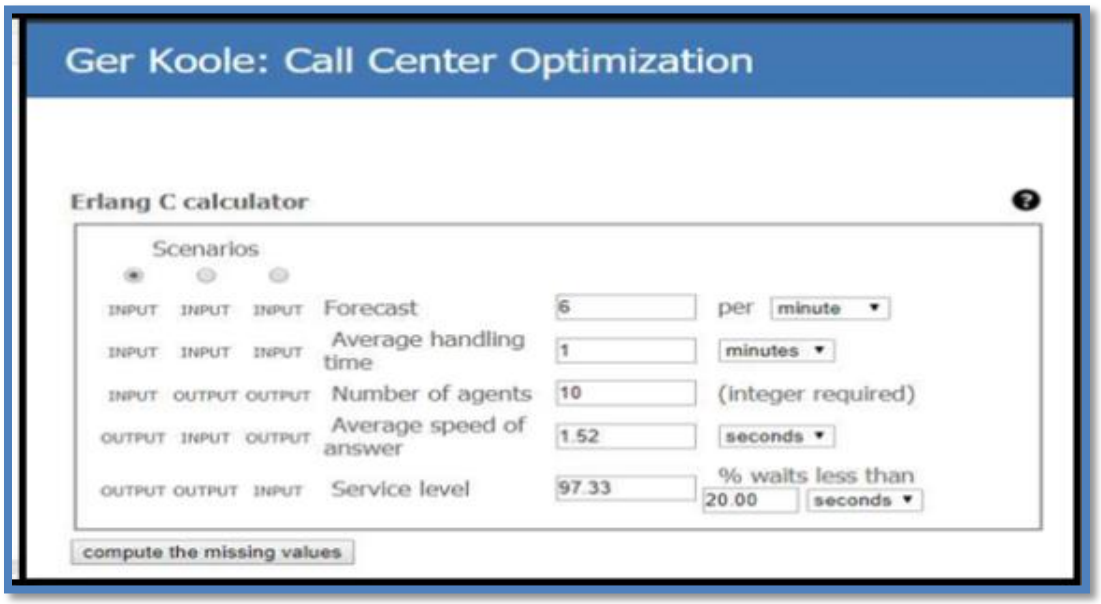


Figure 54: Erlang C Calculator

ONLINE ERLANG X CALCULATOR [3]

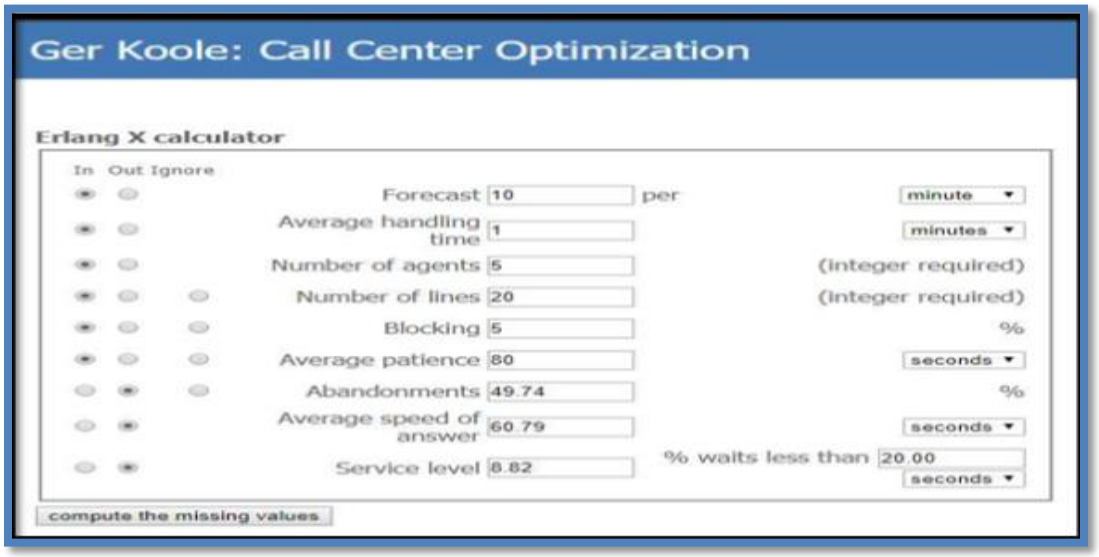
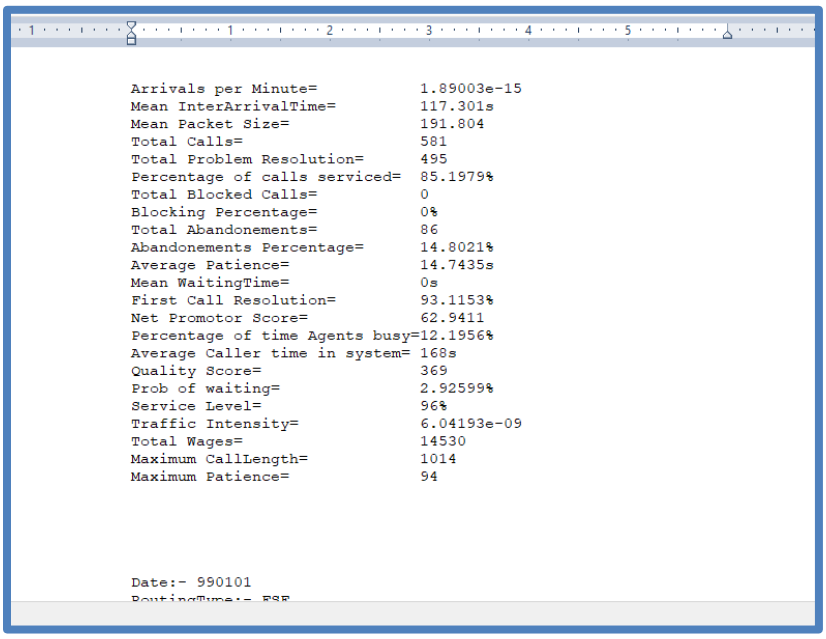


Figure 55: Erlang X Calculator



```
Arrivals per Minute=            1.89003e-15
Mean InterArrivalTime=          117.301s
Mean Packet Size=               191.804
Total Calls=                    581
Total Problem Resolution=       495
Percentage of calls serviced=   85.1979%
Total Blocked Calls=            0
Blocking Percentage=            0%
Total Abandonements=            86
Abandonements Percentage=       14.8021%
Average Patience=               14.7435s
Mean WaitingTime=               0s
First Call Resolution=          93.1153%
Net Promotor Score=             62.9411
Percentage of time Agents busy=12.1956%
Average Caller time in system=  168s
Quality Score=                  369
Prob of waiting=                2.92599%
Service Level=                  96%
Traffic Intensity=              6.04193e-09
Total Wages=                    14530
Maximum CallLength=             1014
Maximum Patience=               94



Date:- 990101
```

Figure 56: NS-3 Call Center Log File

log.txt - Notepad

File Edit Format View Help

```
callerLeft  1 undefined 1526039272061 1526039389548 false 1526039503355 113807 true
callerLeft  1 undefined 1526039400141 1526039421382 false 1526039527517 106135 true
callerLeft  1 undefined 1526039457807 1526039478712 false 1526039527558 48846 true
callerLeft  7 undefined 1526039508952 1526039510248 false 1526039529011 18763 true
0.000019331316380953444 51778.15555555555 10448165 124382.91666666667 84 1 2.127659574468085 3094861 36843.583333333336 5 6.756756756756757 0.9333333333333333 322 810 20 68.57638888888889 11722262 3052087813949
26036679.729125656 4191300 46570 58800 84 83 92.22222222222223 10000 28.888888888888886 26 923630 0.00001802060012030896 1.071727005793152
Agent Left  1526039478709 700 7 false 1526039478712 0 1526039588206 0
Agent Left  1526039492037 700 7 false 1526039510246 0 1526039588288 0
0.000019601614235544 51016.206521739135 10448165 124382.91666666667 84 1 2.127659574468085 3094861 36843.583333333336 5 6.756756756756757 0.9333333333333333 322 810 20 68.57638888888889 11928010 3052088001485
25587570.780750517 4191300 46570 58800 84 83 92.22222222222223 10000 28.888888888888886 26 923630 0.00001802060012030896 1.087733710569037
callerLeft  1 undefined 1526039542409 1526039596415 false 1526039596415 0 true
callerLeft  1 undefined 1526039464259 1526039485909 false 1526039598768 112859 true
Agent Left  1526039485904 700 7 false 1526039485905 0 1526039630210 0
0.000019955417496742016 50111.70526315789 10561024 125726.47619047618 87 1 2.127659574468085 3246165 37312.24137931035 6 6.451612903225806 0.9157894736842105 339.5 855 24 68.19860484201887 12072316 3052088145790
25281711.85868561 4266956 45881.24731182796 60900 88 86 90.52631578947368 10000 27.368421052631582 26 923630 0.00001817666717644905 7 1.0978589915866221
callerLeft  1 undefined 1526039590870 1526039625987 false 1526039665890 39903 true
callerLeft  1 undefined 1526039535034 1526039619585 false 1526039665968 46383 true
callerLeft  1 undefined 1526039603638 1526039635274 false 1526039667152 31878 true
Agent Left  1526039619582 700 7 false 1526039619584 0 1526039673963 0
Agent Left  1526039635270 700 7 false 1526039635273 0 1526039675775 0
Agent Left  1526039625985 700 7 false 1526039625986 0 1526039677707 0
0.000019955417496742016 50111.70526315789 10679188 122749.28735632185 88 1 2.127659574468085 3319034 37716.295454545456 6 6.451612903225806 0.9263157894736842 339.5 855 24 68.19860484201887 12218924 3052088292392
24978372.010432344 4418260 46508 61600 88 87 91.57894736842105 10000 27.368421052631582 26 923630 0.00001820634031663722 4 1.0960696740632965
Agent Left  1526039688063 700 7 false  0 1526039747896 0
0.000019955417496742016 50111.70526315789 10679188 122749.28735632185 88 1 2.127659574468085 3319034 37716.295454545456 6 6.451612903225806 0.9263157894736842 339.5 855 24 68.19860484201887 12278757 4578128040288
37284947.00471717 4418260 46508 61600 88 87 91.57894736842105 10000 27.368421052631582 26 923630 0.00001820634031663722 4 1.0960696740632965
0.000019710132335437492 50735.32653061224 10679188 122749.28735632185 89 1 2.127659574468085 3320097 37304.4606741573 6 6.451612903225806 0.9270833333333334 343 864 24 68.23720552396426 12278757 4578128040288
37284947.00471717 4418260 46508 62300 89 88 91.66666666666666 10000 28.125 27 923630 0.00001792877883259879 1.0993572133089051
callerLeft  1 undefined 1526039609530 1526039682399 false 1526039867758 185359 true
callerLeft  1 undefined 1526039816953 1526039867845 false 1526039867845 0 true
Agent Left  1526039682397 700 7 false 1526039682398 0 1526039867963 0
callerLeft  1 undefined 1526039820980 1526039870103 false 1526039870103 0 true
0.000019710132335437492 50735.32653061224 10864547 122073.5617977528 89 1 2.127659574468085 3320097 37304.4606741573 8 8.163265306122245 0.9270833333333334 343 864 24 68.23720552396426 12464323 4578128225853
36729858.700332135 4591144 46848.40816326653 62300 89 88 91.66666666666666 10000 28.125 27 923630 0.00001792877883259879 1.0993572133089051
Agent Left  1526039805583 700 7 false 1526039813003 0 1526039916256 0
0.000019977672478521035 50055.88118811881 10864547 122073.5617977528 92 1 2.127659574468085 3349132 36403.608695652176 8 8.163265306122245 0.9108910891089109 360.5 909 32 67.38378793699577 12574996 4578128329106
36406598.690814696 4591144 46848.40816326653 64400 92 91 90.09900990099001 10000 27.722772277227726 28 923630 0.00001813497546850329 1.101610118702904
callerLeft  1 undefined 1526039904562 1526039921988 false 1526039992527 70539 true
callerLeft  1 undefined 1526039811941 1526039913004 false 1526039992608 179604 true
Agent Left  1526039910748 700 7 false 1526039910750 0 1526039992619 0
Agent Left  1526039921985 700 7 false 1526039921987 0 1526039992719 0
callerLeft  1 undefined 1526039900231 1526039910751 false 1526039992769 82018 true
0.000019977320100602884 50067.087378640776 11196708 121703.34782608696 92 1 2.127659574468085 3349132 36403.608695652176 8 8.163265306122245 0.9108910891089109 360.5 909 32 67.38378793699577 12727601 4578128481707
35970081.727947004 4620153 45744.08910891089 64400 92 91 90.09900990099001 10000 27.722772277227726 28 923630 0.00001813497546850329 1.101363552474506
callerLeft  1 undefined 1526039895780 1526039896869 false 1526040031910 135041 true
0.000020200310122856363 49504.19047619047 11331749 123171.18478260869 94 1 2.127659574468085 3451697 36720.18085106383 8 8.163265306122245 0.8952380952380953 374.5 945 32 67.55456899741029 12727601 4578128481707
35970081.727947004 4621242 44866.42718446602 65800 96 93 88.57142857142857 10000 26.666666666666668 28 923630 0.00001805017757342456 1.1191197449824437
Agent Left  1526040056619 700 7 false 1526040056622 0 1526040112616 0
Agent Left  1526039878613 700 7 false 1526039896865 0 1526040112687 0
Agent Left  1526040052220 700 7 false 1526040052222 0 1526040112715 0
0.000020200310122856363 49504.19047619047 11331749 123171.18478260869 96 1 2.127659574468085 3611134 37615.979166666664 8 8.163265306122245 0.9142857142857143 374.5 945 32 67.55456899741029 13078167 4578128814016
35005890.45862467 4621242 44866.42718446602 67200 96 95 90.47619047619048 10000 26.666666666666668 28 923630 0.00001819185471075235 1.1104041035968097
```

Figure 57: Node.js Call Center Log File

NS-3 CODE

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/applications-module.h"
#include "ns3/netanim-module.h"
#include "ns3/animation-interface.h"
#include "ns3/ipv4-address-helper.h"
#include "ns3/internet-stack-helper.h"
#include "ns3/ipv4-global-routing-helper.h"
#include "ns3/ipv4-global-routing.h"
#include "ns3/callback.h"
#include "ns3/point-to-point-helper.h"
#include "ns3/simulator.h"
#include <string>
#include <cstring>
#include <sstream>
#include <iostream>
#include <fstream>
using namespace ns3;
using namespace std;
ofstream myfile;


int Mean_InterArrival_Time       =5;
int Mean_Packet_Size             =50;
int Mean_Patience                =60;
int number_of_Agent_nodes        =5;
int simtime                      =86500;
int randm                        =1;
int Maximum_Queue_Length         =20;
uint32_t WagePerCall             =100;
uint32_t AverageServiceTime      =45;
double precision                 =1;
uint32_t meanskill               =31;
uint32_t meanVRUTime             =10;
int RoutingType                  =4;//1-SBR, 2-FSF, 3-LIF, 4-LCR
int MaxCallsPerHour              =0;
uint32_t TotalWages              =0;
double MaxCallLength             =0;
int   **TABLE                    =new int  *  [25];
bool * idleArray                 =new bool   [number_of_Agent_nodes];
bool * Servicing                 =new bool   [number_of_Agent_nodes];
uint32_t * SkillArray            =new uint32_t[number_of_Agent_nodes];
uint32_t * WagesArray            =new uint32_t[number_of_Agent_nodes];
uint32_t * AvgServiceTimeArray  =new uint32_t[number_of_Agent_nodes];
uint32_t *TimeOfLastServiceArray=new uint32_t[number_of_Agent_nodes];
string **TEMP3                   =new string*[90000];
string **AVAIL_AGENT             =new string*[90000];
int   **AG_SER                   =new int  *[90000];
string *TEMP                     =new string [90000];
string *ENTRYTIME                =new string [90000];
int   *VRU_ENTRY                 =new int   [90000];
string *ordered                  =new string [90000];
Ptr<Node> DistributorN;
```

DISTRIBUTOR QUEUE

```
class DistributorQueue{
public:
DistributorQueue(int m,AnimationInterface* anim)
{…}


struct Link
{
 Link *     Back;
 Link *     Front;
 uint32_t   Data;
 uint32_t   Skill;
 Ptr<Packet> pkt;
 Time       EnqueueTime;
 uint32_t   QueueWaitingTime;
 uint32_t   VRUtime;
};


void Enqueue(Ptr<Packet> packet)
{…}


void CountDown()
{…}


void VRUEnqueue(Ptr<Packet> packet,uint32_t VRUTime)
{…}



void VRUCountDown(DistributorQueue * queue)
{…}


Ptr<Packet> Dequeue()
{…}


Link * Head;
Link * Tail;
Link * MyLink;
AnimationInterface* MyAnimator;
int TotalPackets,NumberOfPacketsInQueue,TotalPacketsDequeued,MaximumQueueLimit;

};
```

CALLER APPLICATION

```
class CallerApp:public Application{
public:
CallerApp()
{…}


void Setup(Ptr<Socket> mysocket, Address address,uint32_t meantime, uint32_t meansize, uint32_t meanwaitingtime, uint32_t minskill,
uint32_t boundskill)
{…}


void StartApplication()
{…}


void StopApplication ()
{…}


void SetHourlyPacketInterArrivalTime()
{…}


void SendPacket()
{…}

Address                  peer;
EventId                  SendEvent;
EventId                  SendEvent1;
Ptr<Socket>              socket;
int                   TotalPacketsSent;
Ptr<ExponentialRandomVariable> PacketSizeGenerator;
Ptr<ExponentialRandomVariable> SkillTypeGenerator;
Ptr<ExponentialRandomVariable> WaitingTimeGenerator;
Ptr<ExponentialRandomVariable> InterArrivalTimeGenerator;
uint32_t time,size,waitingtime,skill,myboundsize,myboundwaitingtime,myboundskill;

};
```

RECEIVER APPLICATION

```
class ReceiverApp: public Application{
public:
ReceiverApp():peer(),MySocket(0)
{
 VRUTimeGenerator    =CreateObject<ExponentialRandomVariable>();
}


void Setup(Ptr<Socket> socket,Address Agentaddress,DistributorQueue* myqueue,int mypeernumber,uint32_t
meanwaitingtime,DistributorQueue* VRUqueue)
{…}


void StartApplication()
{…}

 void StopApplication ()
 {…}

void SocketRecv(Ptr<Socket> socket)
{…}


void CountDown()
{…}


Address         peer;
int            PeerNumber;
EventId        SendEvent;
uint32_t        PacketSize;
Ptr<Packet>      MyPacket;
Ptr<Socket>      MySocket;
Ptr<ExponentialRandomVariable> VRUTimeGenerator;
DistributorQueue* queue;
DistributorQueue* VRUQueue;
uint32_t VRUtime;
};
```

DISTRIBUTOR APPLICATION

```
class DistributorApp: public Application{
public:
DistributorApp():peer(),dataRate(0),MySocket(0){}

void Setup(Ptr<Socket> socket, DataRate datarate,Address Agentaddress,uint32_t* mySkillArray,DistributorQueue* myqueue,bool* myidle,int
mypeernumber,AnimationInterface * anim)
{…}

void StartApplication()
{…}

void StopApplication ()
{…}

void SendPacket(int AgentNumber)
{…}

void swap(uint32_t* Arr,int a, int b)
{…}

uint32_t * SortMax(uint32_t Array[], int n)
{…}

//Fastest Server First Routing Scheme...
void FSF()
{…}

//Longest Idle First...
void LIF()
{…}

// Lowest Cost First routing Scheme..
void LCR()
{ …}

void SkillBasedRouting()
{…}

bool *          idle;
Address         peer;
int             PeerNumber;
DataRate        dataRate;
EventId         SendEvent;
uint32_t        PacketSize;
Ptr<Packet>     MyPacket;
uint32_t *      SkillArray;
Ptr<Socket>     MySocket;
DistributorQueue* queue;
AnimationInterface* MyAnimator;
};
```

AGENT APPLICATION

```
class AgentApp:public Application{
public:
AgentApp()
{…}

void Setup(Ptr<Socket> socket,Address address, uint32_t myAgentskill,int myAgentnumber,AnimationInterface* animator,uint32_t
pic1,uint32_t pic2, uint32_t wage, uint32_t AvgServiceTime)
{…}

void StartApplication()
{…}

 void StopApplication ()
 {…}

void SocketRecv(Ptr<Socket> socket)
{…}

void ServiceComplete()
{…}

Address          peer;
uint32_t CallsRecieved_InADay; //......to model the exhaustion level.
uint32_t CallsRecieved_SinceJoined; //......to model the experience level.
uint32_t AverageServiceTime;
uint32_t WagePerCall; //......to model the wages.
uint32_t TimeOfLastService; //..... For Longest Idle First Routing scheme.
uint32_t TimeOfLastArrival; //..... For Longest Idle First Routing scheme.
uint32_t breaks; //......to model the .
uint32_t         myp1;
uint32_t         myp2;
int              PacketsServed;
EventId          SendEvent;
int              AgentNumber;
uint32_t         PacketSize;
Ptr<Socket>      MySocket;
Ptr<Packet>      MyPacket;
uint32_t         AgentSkill;
AnimationInterface* mya;
//EFFICIENCY OF THE Agent.
uint32_t OP_Efficiency;
Ptr<ExponentialRandomVariable> EfficiencyGenerator;

};
```

MAIN

```
NS_LOG_COMPONENT_DEFINE ("CallCenterSimulation");
int
main (int argc, char *argv[])
{
ifstream INSTREAM;
INSTREAM.open("january.txt");
ofstream o;
o.open("results.txt");
…
LogComponentEnable ("CallCenterSimulation", LOG_LEVEL_INFO);

 NodeContainer caller_node_container;
 NodeContainer distributor_node_container;
 NodeContainer Agent_nodes_container;

 caller_node_container. Create(1);
 distributor_node_container. Create(1);
 Agent_nodes_container. Create(number_of_Agent_nodes);


 PointToPointHelper caller_distributor_p2p_Helper;
 caller_distributor_p2p_Helper.SetDeviceAttribute ("DataRate", StringValue ("64kbps"));
 caller_distributor_p2p_Helper.SetChannelAttribute("Delay"   , StringValue ("3s"    ));

 NetDeviceContainer caller_distributor_net_device_container;
 NetDeviceContainer distributor_Agents_net_device_container_Array[number_of_Agent_nodes];

 caller_distributor_net_device_container = caller_distributor_p2p_Helper.Install (caller_distributor_nodes_container);
 for(int i=0;i<number_of_Agent_nodes;i++){distributor_Agents_net_device_container_Array[i] = caller_distributor_p2p_Helper.Install
(distributor_Agents_node_container[i]);}

 InternetStackHelper stack;
 Ipv4AddressHelper address;
 stack  .Install (c);
 address.SetBase ("10.1.1.0", "255.255.255.0");
 Ipv4InterfaceContainer interface     =address.Assign (caller_distributor_net_device_container);


 Ptr<Socket> CallerSocket       = Socket::CreateSocket(c.Get(0)  ,TypeId::LookupByName ("ns3::Ipv4RawSocketFactory"));
 Ptr<Socket> DistributorSocket   = Socket::CreateSocket(c.Get(1)  ,TypeId::LookupByName ("ns3::Ipv4RawSocketFactory"));
 Ptr<Socket> AgentSocketArray     [number_of_Agent_nodes];
 Ptr<Socket> DistributorSocketArray[number_of_Agent_nodes];

 AnimationInterface* anim = new AnimationInterface ("anim1.xml");
 DistributorQueue q         =DistributorQueue(Maximum_Queue_Length,anim);
 DistributorQueue* queue     = &q;
 DistributorQueue Q         =DistributorQueue(1000,anim);
 DistributorQueue* VRUqueue  = &Q;

 Ptr<CallerApp>     capp = CreateObject<CallerApp>     ();
 Ptr<ReceiverApp>    rapp = CreateObject<ReceiverApp>    ();

 for (int i = 0; i < number_of_Agent_nodes; i++)
 {
 Ptr<AgentApp>    Oapp   =CreateObject<AgentApp>   ();
```

```
 c.Get (i+2)->AddApplication (Oapp);

Ptr<DistributorApp> Dapp=CreateObject<DistributorApp>();
 c.Get (1)  ->AddApplication (Dapp);
}


Ipv4GlobalRoutingHelper Router = Ipv4GlobalRoutingHelper();
 Router.PopulateRoutingTables();

 Simulator::Stop (Seconds (simtime));
 Simulator::Run ();
 NS_LOG_INFO("Ending Topology");
 Simulator::Destroy ();
 myfile.close();




…
 out<<"Arrivals per Minute="<<60/((double(sum2)/double(calls-1))/1e6)<<endl;
 out<<"InterArrivalTime="<<((double(sum7)/double(calls)))<<"s"<<endl;
 out<<"Mean Packet Size="<<(double(sum4)/double(calls)) <<endl;
 out<<"Blocking Percentage="<<((double(Drops)*100)/double(calls))<<"%"<<endl;
 out<<"Average Patience="<<(double(sum5)/double(calls)) <<"s"<<endl;
 out<<"Abandonements Percentage="<<((double(Abandonements)*100)/double(calls)) <<"%"<<endl;
 out<<"Mean WaitingTime="<<(double(mean)/1e6) <<"s"<<endl;
 out<<"Total Calls="<< calls<<endl;
 out<<"Total Problem Resolution="<< totalcalls<<endl;
 out<<"Percentage of calls serviced="<< ((double(calls)-double(Abandonements)-double(Drops))/double(calls))*100<<"%"<<endl;
 out<<"First Call Resolution="<< ((double(calls)-double(prob))/double(calls))*100<<"%"<<endl;
 out<<"Net Promotor Score="<<(((double(Promoter)-double(Detractor)))/(double(Promoter)+double(Passive)+double(Detractor)))*100<<endl;
 out<<"Utilization="<<((double(totalcalls)*(double(sum4)/double(calls)))/(double(simtime*number_of_Agent_nodes)))*100<<"%"<<endl;
 out<<"Average Caller time in system= "<<(double(avgCalTime)) <<"s"<<endl;
 out<<"Total Agents Salary="<<(double((var/3600)*HourlyWage)) <<"Rs"<<endl;
 out<<"Quality Score="<<(double(QS)) <<endl;
 out<<"Prob of waiting="<<(double(pep)/double(calls))*100<<"%"<<endl;
 out<<"Service Level="<<double(servicelevel) <<"%"<<endl;
 out<<"Traffic Intensity="<<(double(double(calls-1)*1000000)/((double(sum2))))*(double(sum4)/double(calls)) <<endl;
 out<<"Total Wages="<<TotalWages<<endl;
 out<<"MaxCallLength="<<MaxCallLength<<endl;
 return 0;
 o.close();
}
```

NODE.JS CODE

APP.JS

```
var fs = require('fs');
var os = require("os");
var sios = require('socket.io');
var http = require('http');
var https = require('https');

var callarray = [];
var agentarray = [];
var MaxQueueTime = 0;
var infoObj = {CallersNo:0, AgentsNo:0};
var StartTime = Date.now();
var ArriPerSec = 0;
var InterArriTime = 0;
var TotalServicTime = 0;
var MeanCalltime = 0;
var TotalCallServ = 0;
var Drops = 0;
var BlockPercen = 0;
var TotalQueueTime = 0;
var AvgQueTime = 0;
var Abandon = 0;
var AbandonPercen = 0;
var PercenCallServ = 0;
var Passive = 0;
var Promoter = 0;
var Detractor = 0;
var NetPromoScore = 0;
var AgentAvail = 0;
var AgentBusy = 0;
var Utilization = 0;
var TotCallerTime = 0;
var AvgCallerTime = 0;
var AgenSalary = 0;
var QualityScores = 0;
var Wait = 0;
var ProbWait = 0;
var SevicThresh = 10000;
var ServiceLevel = 0;
var CorreServiced = 0;
var MaxCallLen = 0;
var ServiceRate = 0;
var TrafficIntens = 0;
var logString="";

const options = {
  key: fs.readFileSync('./ryans-key.pem'),
  cert: fs.readFileSync('./ryans-cert.pem')
};

var app = https.createServer(options, (req, res) => {
   if (req.url === '/')
```

```
    {fs.readFile('CallCentre.html', function(error,data){res.end(data);})}

  if ((req.url === '/Caller')||(req.url === '/caller'))
    {fs.readFile('caller.html', function(error,data){res.end(data);})}

  if ((req.url === '/Agent')||(req.url === '/agent'))
    {fs.readFile('agent.html', function(error,data){res.end(data);})}

});

app.listen(8000,'192.168.1.4');

const io = sios.listen(app);

function Routing()
{…}

function SendToPeer(Socket,Tag, Message)
{…}

fs.readFile("log.txt", 'utf8', function (err,data)
{…});

function logfile()
{…}

setInterval(logfile,60000);

io.on('connection', function (socket) {
socket.on('disconnect', function(){…});
socket.on('offer',function(offer){… Routing(); });
socket.on('Type', function (data) {… Routing(); } );
socket.on('answer',function(answer){SendToPeer(socket,'answer',answer);});
socket.on('CompleteFileReceived',function(){SendToPeer(socket,"CompleteFileReceived","")});
socket.on('TextMessage', function(Message){SendToPeer(socket, 'TextMessage', Message);});
socket.on('iceCandidate', function(candidate){ SendToPeer(socket,"iceCandidate", candidate );});
socket.on('fileInformation',function(fileInformation) {SendToPeer(socket,'fileInformation',fileInformation)});
socket.on('StartSendingFile',function(StartSendingFile) {SendToPeer(socket,'StartSendingFile',StartSendingFile)});
});
```

CALLER.HTML

```
<!DOCTYPE HTML>
<html>
<head> …
<h1>Caller</h1>
<script src="/socket.io/socket.io.js"></script>
</head>
<body>

 <div>
 <audio id = "LocalVideo"></audio>
 <button id = "CallButton">CALL</button>
 <a>Hang Up</a>
 </div>

<div>
<textarea id = "sent"></textarea>
 <button>Send Text</button>
 <form><input type="file"></input></form>
 <select id="BandwidthList"></select>
 <canvas id="localcanvas”></canvas>
 <textarea id="ByteRecieved"></textarea>
 </div>

 <div>
 <canvas id="remotecanvas"></canvas>
 <audio id = "RemoteVideo"></audio>
 <textarea id = "received"></textarea>
 <a id = "download">noFile</a>
 <video> </video>
 <img></img>
 </div>

<div>
<p>Name:</p>
<textarea id="NameText"></textarea>
<p>Codec:</p>
<select id="CodecList">Choose Codec</option> </select>
<p>Skill:</p>
<select id="SkillSet"></select>
</div>

 <script>
  var socket = io();
  var ReceiveText;
  var pc1 = new RTCPeerConnection(servers);
  var localStream;
  var sendChannel;
  var offerOptions = {offerToRecieveAudio:1,offerToRecieveVideo:0,voiceActivityDetection:false};
  var fileInput = document.getElementById('fileInput');
  var receiveChannel;
  var receiveBuffer = [];
  var receivedSize = 0;
  var FileName;
  var sendChannel;
  var downloadAnchor = document.getElementById('download');
```

```
var recieveVideo = document.getElementById('recieveVideo');
var recieveImage = document.getElementById('recieveImage');
var SkillSet = document.getElementById("SkillSet");
var CallButton = document.getElementById("CallButton");
var TextButton=document.getElementById("TextButton");
var fileInput=document.getElementById("fileInput");
var bandwidthSelector=document.getElementById("BandwidthList");
var codecSelector=document.getElementById("CodecList");
var SignInForm=document.getElementById("SignInForm");
var localcanvas=document.getElementById("localcanvas");
var remotecanvas=document.getElementById("remotecanvas");
socket.connect();

socket.on('Information', function(data){…});
socket.on('TextMessage',function(data){…});
socket.on('answer',function(answer){…);
socket.on('iceCandidate',function(candidate){…});
socket.on('fileInformation',function(fileInformation){…));
socket.on('StartSendingFile',function(StartSendingFile){…});
socket.on("CompleteFileReceived",function(){…})

pc1.onicecandidate = function(e) {…};
pc1.ontrack = gotRemoteStream;
pc1.ondatachannel = receiveChannelCallback;

sendChannel = pc1.createDataChannel('sendDataChannel');

// functions.....................

function SkillChosen() {…}
function gotStream(Stream){…}
function displayStates() {…}
function myfunc(){…}
function CallButtonPressed(){…}
function gotDescription1(desc) {…}
function onIceCandidate(event){… }
function FileChosen() {…}
function onSendChannelStateChange() {…}
function gotRemoteStream(event){…}
function SendData() {…}
function receiveChannelCallback(event) {…}
function onReceiveMessageCallback(event) {…}
function BandwidthChosen(){…}
function updateBandwidthRestriction(sdp, bandwidth) {…}
function getCodecPayloadType(sdpLine) {…}
function setDefaultCodec(mLine, payload) {…}
function codecChosen(){…}
function StreamVisualizer(remoteStream, canvas) {…}
StreamVisualizer.prototype.start = function() {…};
StreamVisualizer.prototype.draw = function() {…};
StreamVisualizer.prototype.getFrequencyValue = function(freq) {…};

</script>
</body>
</html>
```

AGENT.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<h1>Agent</h1>
<script src="/socket.io/socket.io.js"></script>
 </head>
 <body>
 <div>
 <audio id = "LocalVideo"></audio>
 <button>Pick Up</button>

<div>
 <a id = "Hang Up">Hang Up</a>
</div>
 <textarea></textarea>
 <button>Send Text</button>
 <form onchange="FileChosen()"><input type="file"></form>
 <textarea id= "ByteRecieved"></textarea>
 <canvas id="localcanvas"></canvas>
 </div>

 <div>
 <canvas></canvas>
 <audio></audio>
 <textarea></textarea>
 <a></a>
 <video></video>
 <img></img>
 </div>

  <div>
  <p>Name:</p>
  <textarea></textarea>
 <p>Skill:</p>
 <select id="SkillSet"></select>
</div>

   <script>
   var socket = io();
   var InfoObj ={};
   var servers = null;
   var receiveChannel;
   var receiveBuffer = [];
    var receivedSize = 0;
    var Remotestream;
    var pc2 = new RTCPeerConnection(servers);
    var offerOptions = {offerToRecieveAudio:1,offerToRecieveVideo:0,voiceActivityDetection:false};
   var downloadAnchor = document.getElementById('download');
    var FileName;
    var FileSize=0;
    var sendChannel;
    var fileInput = document.getElementById('fileInput');
    var recieveVideo = document.getElementById('recieveVideo');
    var recieveImage = document.getElementById('recieveImage');
    var ByteRec = document.getElementById('ByteRecieved');
```

```
    var SkillSet=document.getElementById("SkillSet");
    var PickupCallButton=document.getElementById("PickupCallButton");
    var TextButton=document.getElementById("TextButton");
    var SignInForm=document.getElementById("SignInForm");
    var NameText=document.getElementById("NameText");
    var localcanvas=document.getElementById("localcanvas");
    var remotecanvas=document.getElementById("remotecanvas");

    socket.connect();
    socket.on('Information', function(data){…});
    socket.on('TextMessage',function(data){…});


    socket.on('offer',function(offer){…});
    socket.on('iceCandidate',function(candidate){…});
    socket.on('StartSendingFile',function(StartSendingFile){SendData();});
    socket.on("CompleteFileReceived",function(){fileInput.disabled=false;})


    pc2.onicecandidate = function(e) {console.log('Local IceCandidate received'+JSON.stringify(e));onIceCandidate(e);};
    pc2.ontrack = gotRemoteStream;
    pc2.ondatachannel = receiveChannelCallback;


    sendChannel = pc2.createDataChannel('sendDataChannel');
    sendChannel.binaryType = 'arraybuffer';
    sendChannel.onopen = onSendChannelStateChange;

    function PickupCall(){…}
    function SkillChosen() {…}
    function displayStates() {…};
    function gotDescription2(desc) {…}
    function gotStream(Stream){…}
    function receiveChannelCallback(event) {…}
    function onReceiveMessageCallback(event) {…}
    function onIceCandidate(event){…}
    function gotRemoteStream(event){…}
    function FileChosen() {…}
    function onSendChannelStateChange() {…}
    function SendData() {…}
    function StreamVisualizer(remoteStream, canvas) {…};

    StreamVisualizer.prototype.draw = function() {…};
    StreamVisualizer.prototype.getFrequencyValue = function(freq) {…};

    </script>


  </body>
</html>
```

CALLCENTER.HTML

```
<!DOCTYPE HTML>
<html>
<head>
<div>
<h1 style="font-size: 50px;color: rgb(0,0,0);">CallCenter</h1>
</div>
</head>

<body style="background-image: url(); background-repeat: no-repeat; background-position: 300px 50px; ;background-size: 800px 700px;">

<div>
<a href="https://192.168.1.4:8000/Caller">Visit Caller page</a>
</div>
<div>
<a href="https://192.168.1.4:8000/agent">Visit Agent page</a>
</div>


</body>
</html>
```