POLITECNICO DI TORINO

Master Degree course in Communications and Computer Networks Engineering

Master Degree Thesis

Simulating a distributed cloud gaming engine

Supervisors
Prof. Paolo Giaccone

Candidate
XinQi Yuan

Academic Year 2021-2022

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance.

I would first like to thank my supervisor, Prof Paolo Giaccone, whose expertise was invaluable in formulation the research questions and methodology. you insightful feedback pushed me to sharpen my thinking and brought my work to a higher level.

I would particularly like to acknowledge my colleagues and supervisor, Lotfimahyari Iman, for their wonderful collaboration and patient support.

In addition, I would like to thank my parents for their support, You are always there for me.

Abstract

This thesis designs a resource allocation algorithms for distributed cloud gaming engines with the aim of improving QoS (Quality of Service) and QoE (Quality of Experience) for cloud gaming. Cloud gaming is an emerging technology that allows players to play games without high-end hardware. Games are run on cloud servers via containers and rendered images are pushed to the user. Cloud gaming technology has many advantages over traditional gaming technologies, but there are still factors that affect the player experience.

In the face of such challenges, resources must be dynamically aggregated in an on-demand manner in a cloud architecture. The first step is to analyse the design goals for a new generation of distributed cloud gaming engines and then to design offline algorithms. In the simulation of the algorithms, two different models, namely "bottom-up" and "top-down", are to be built, corresponding to different allocation schemes of the cloud gaming engine to users, and using multiple sets of different user combination data, to be experimented, compared and analysed with the project partners, and to be compared with the traditional cloud gaming engine. Based on the QoE and cost analysis, the best approach is selected.

List of acronyms

CN	Cloud Node
CODEG	Cloud-Oriented Distributed Engine for Gaming
DB	Database
EN	Edge Node
FPS	Frames Per Second
GEM	Game Engine Module
QoS	Quality of Service
QoE	Quality of Experience
TEs	Terminal equipments
VM	Virtual Machine

Contents

Chapter 1

Introduction

1.1 Overview

This thesis designs a resource allocation algorithms to improve the QoS of a specific Cloud Gaming Engine (CODEG) for improving the QoE of users.

Video games are a hugely popular form of entertainment, with the global video game market generating an estimated 159 billion in annual revenues from hardware, software and services by 2020. This is three times the size of the global music industry in 2019 and four times the size of the film industry in 2019.According to [1], users want to run games, they need a set of physical computing resources (game console or PC), these hardware is usually expensive, inconvenient to carry and needs to be upgraded regularly. many gamers will forgo buying equipment because of cost or other reasons. since 2015, CPUs, GPU, virtualisation, video codecs and streaming technologies are maturing, allowing cloud gaming to break through the technical barriers to cloudification. Since the commercialisation of 5G in 2019, the rapid construction of 5G networks and large-scale pilots of edge computing have enabled cloud gaming to break through the channel barrier of network bandwidth. For gamers or those in the gaming industry, cloud gaming is no longer a novelty.

Cloud gaming is a relatively new technology where in-game simulations are performed in high-end private data centres over the internet and the rendered results are streamed to the end user [2]. There are already a number of cloud gaming services in operation, such as Xbox CLoud Gaming [3], Google Stadia [4], Geforce Now [5] and Amazon Luna [6], allowing gamers to play without owning and maintain cutting edge hardware for big budget games. This technology has many advantages, such as gamers can store and play many games on their PCs without having to spend money on high-end hardware and large storage disks, and data is not easily lost.

This chapter continues as follows. In section 1.2, the problem of a cloud game engine is explained. Section 1.3 describes the research goal of this thesis, which is to create a model to implement a distributed game engine so as to protect the quality of service for users. Section 1.4 summarises the contributions of the thesis. Section 1.5 provides an overview of the structure of the thesis.

1.2 Problem and motivation

For existing cloud gaming engine operations, cloud gaming systems typically run in clusters of servers in data centres and are available to multiple users at the same time. Players can connect to the cloud gaming service through multiple servers. However, there are problems with such a configuration.

1. The delay in connecting to the server will be uneven due to the physical distance between the player and the server. Generally, players with lower latency will have more advantage in multiplayer games, which will affect the fairness of the game.

2. Fluctuations in the user's network state or the user's region can cause connection loss or excessive fluctuations in delay.

The user experience of cloud gaming is usually quantified by QoE. As defined by Qualinet in 2012 [7], QoE measures the level of enjoyment or frustration a user experiences when using a service or application. It is based on the user's personality and current situation in order to meet the user's expectations of the utility and functionality of the application or service used.

The parameter that measures the performance of the cloud gaming engine is known as Quality of Service, which is a measure of the overall performance of the service, usually considering some parameters of the network quality itself. QoS is also an important condition that affects QoE, and usually high QoS also leads to high QoE [7]. When a good QoS is already in place, QoE is measured only in relation to the user's own mood.

In traditional cloud gaming engines, the only way to improve user QoS and QoE is to increase hardware performance, for example by replacing faster networks and interfaces in data centres and using more powerful computing and rendering devices. But this approach can add significantly to budgets and can result in wasted computing resources.

And now there is a newly proposed cloud-oriented distributed gaming engine called CODEG, which aims to leverage modern cloud technologies and edge computing and the heterogeneity of these network resources to deliver a better cloud gaming experience and reduce costs for service providers.

1.3 Research objects

Section 1.2 introduces the problems faced by traditional cloud gaming engines and a new distributed solution, CODEG, which combines traditional solutions with distributed computing to make full use of edge server resources, which can reduce round-trip times, relieve server pressure on data centres and automatically adapt and adjust the load across the cloud gaming engine. Therefore, this thesis proposes that by allocating computing resources appropriately and focusing on specific QoS metrics, QoS and QoE levels for users can be improved. These concerns are targeted at edge computing and distributed architectures. The overall objective of this paper is to simulate a CODEG scenario, using appropriate resource allocation algorithms to make decisions about user devices, network node state and network state. These decisions are made to improve the QoS of the cloud gaming engine and hence the QoE of the users, and the following are the objectives of the study.

1. Review the traditional cloud gaming engine operation model to clarify the process of cloud gaming and the type of data that is transmitted by cloud gaming.
2. To identify the key parameters that affect the QoS of the cloud gaming engine.
3. Design a resource allocation algorithms for the cloud gaming engine based on the idea of the CODEG scheme.
4. Design a testbed with adaptive architecture to improve the QoS of the cloud gaming engine using edge computing and distributed architecture, where the allocation algorithms we design will help to manage resources automatically. This can then be done for different data and connection type variations. Multiple simulations are performed.

1.4 Thesis contributions

This thesis proposes an adaptive architecture for the CODEG distributed cloud gaming engine, which, by using a distributed architecture at the core and edge of the network, will reduce the probability of server crashes by allowing users to use devices closer to the client, and will make fuller use of network resources, while QoS will be improved.

The main objective of this research is to develop an adaptive architecture to emulate the CODEG distributed cloud gaming engine, which will use distributed architecture and edge computing to improve the QoS provided. the underlying assumption is that distributed architecture and edge computing A new generation of cloud gaming engines can be used to improve the QoS for cloud gaming users.

The main contributions of this study are:

1. evaluate current approaches to cloud gaming operations and provide QoS and QoE improvements.
2. Develop a test platform for a gaming engine using distributed architecture and edge computing (CODEG) and combine these concepts with adaptive allocation algorithms to improve the overall QoS of the cloud gaming platform.
3. document the architectural analysis and experimental testing and compare it to a traditional cloud gaming engine.

1.5 Thesis structure

The rest of the thesis also includes five chapters. Background and literature review, Architecture, Experimental methods and system setup, Experimental/numerical evaluation, and Conclusion.

Chapter 2 is a literature review of related work. The relevant research atmosphere is in four fields: cloud gaming, distributed environments, and edge computing. It focuses on the cloud and user data push.

Chapter 3 explains the Distributed Cloud Gaming Architecture (CODEG) for improving user QoS using distributed resources and edge computing, which is discussed in detail from the high level to the low level, from the scenarios customers may face to the impact on the overall architecture and then to the role of each part in the overall architecture.

Chapter 4 focuses on the architecture's ability to make decisions on how to allocate resources to help customers and the overall service achieve the best QoS, and explores and tests a number of different allocation approaches.

Chapter 5 presents the results of the tests and the impact of the allocation methods on overall performance. As different methods were tested under different conditions, we were able to analyse the allocation methods with better performance and cost.

Chapter 6 concludes the thesis by evaluating the research objectives and test results identified in Section 1.3. It also makes recommendations for future work to improve the current work and expand the scope of the study.

Chapter 2

Background and literature review

2.1 Overview

This chapter presents background research and literature reviews conducted in various areas. The relevant research can be divided into five areas: cloud gaming, distributed environments, edge computing and heterogeneous computing. Each area relates to the overall goal of this work, which is to develop an architecture that leverages distributed architectures to improve QoE for cloud gaming engines.

2.2 Introduction

In this chapter, existing research related to cloud gaming engines and related research is discussed. In this study, user-perceived QoE can be improved by monitoring and improving QoS parameters within the network, as described in Chapter 1.

The domain of cloud gaming is mainly focused on the cloud. The user connects to a server in a data centre provided by the cloud provider and sends operational instructions to the server, which passes the rendered data back to the client. The data flowing to the client can take one of two forms, video or graphics instructions. Cloud gaming services have been found to be highly adaptable to the current state of the network. One of the research objectives in Chapter 1 highlighted the need for such an adaptive architecture. A distributed environment means that multiple or groups of servers need to cooperate to achieve a common goal. In the current network environment, an adaptive algorithms is needed to manage the distribution of data across various devices.

There are many different approaches to distributed management and the research in this paper will focus on structural and adaptive algorithms. These algorithms can be better understood as a set of rules. In normal service operations, the adaptive algorithms will act if one or more conditions are not met. For example, if a server does not have enough resources to add new users, a resource allocation will be made to move the users to other free servers.

Edge computing is very similar to cloud computing, with the difference that in edge computing, data is processed close to the server at the user's device end. The advantage is that the client receives the results of the execution process faster than the results transferred from the cloud.

With a rational allocation algorithms, network performance (QoS) can be improved, user experience (QoS) can be improved, and by using edge computing, the load on cloud servers can be reduced, reducing costs and energy consumption.

The final section of this chapter summarises the literature review and identifies approaches to developing adaptive architectures.

2.3 Cloud gaming

When it comes to cloud games, we have to start with streaming media. As early as the 1980s, a team of engineers tried streaming to display media on computers. However, due to the limitations of hardware performance and network infrastructure at the time, streaming media did not Success. Streaming media has gained huge popularity in recent years with improvements in network connection speeds and user device performance.

The operation mode of the cloud game platform is very similar to that of live streaming. Games are stored and executed on the dedicated hardware of the service provider, and the image or video data is streamed to the user's device. The client's device needs to process the player's input, These inputs will be transmitted back to the server, the server will convert the data into actions and execute them in the game, and render images and sounds back to the player.

The first commercial success in the cloud gaming industry was Nvidia. in May 2012, Nvidia released the cloud gaming service Nvidia Grid, later renamed GeForce Now [5]. It is considered the best cloud gaming service available, as Nvidia is the best known GPU manufacturer and their strength is the best combination of hardware and software. 2019 saw the launch of xCloud (now called xbox Cloud Gaming) [3], who have the advantage of having their own console, Xbox [8] with their own exclusive games and experience in running a console business, and their own subscription service, XGP (xbox game pass), which allows hundreds of games to be played for free[/ref]. Google announced its cloud-based gaming service Stadia [4] in the same year, and Amazon launched its cloud-based gaming service Luna [6] in 2020.

2.3.1 Classification of cloud games

By computing platform

The three main markets for gaming are PC gaming, console gaming and mobile gaming. Cloud gaming can also be divided into two categories based on the computing platform, X86 architecture and ARM architecture (X86 architecture and ARM architecture here refer to the cloud platform on which the games are actually run. The next generation consoles PS5 and XBOX series also use X86 architecture [9] and Nintendo Switch uses ARM architecture [10]). Therefore, the X86 architecture cloud platform is mainly for PC terminal games and console games in the cloud, while the ARM architecture cloud platform is mainly for mobile games in the cloud. This thesis also focuses on the analysis and description of cloud gaming due to the high demand for terminal performance and the stronger demand for cloudification for PC games and console games.

By resource form: virtual machine or physical machine

At present, there are two mainstream resource forms in the market, virtual machine mode and physical machine mode. The virtual machine method generally adopts a combination of cloud resources of servers and professional graphics cards, and allocates resources in a virtualized way, which is more flexible. The cloud resources of the physical machine genre exist in the form of PCs, and the graphics cards are home game graphics cards, which are better adapted to game drivers, and players are less likely to be interfered by other users.

Compare	Virtual	Physical
Cloud resources	Server + Professional Graphics Card	PC + home gaming graphics card
Resource allocation	Virtualization allocation, one server to many user	1 to 1 assignment
Advantage	Flexible resource allocation, rapid instance creation and destruction	Home game graphics cards are better suited for game drivers
Disadvantage	The hardware cost is high; the call delay of virtualized resources is larger than that of physical machines, resulting in a large end-to-end delay of cloud games;	Resource allocation is not flexible enough; home gaming graphics cards are less reliable than professional graphics cards
representative	Google Stadia, NVIDIA GeForceNow	Amazon AWS

Table 2.1: Two main forms of cloud gaming

2.3.2 Traditional game setup

The traditional process of running video games is localized and requires specific computing resources to perform gaming tasks (such as game consoles or PCs).

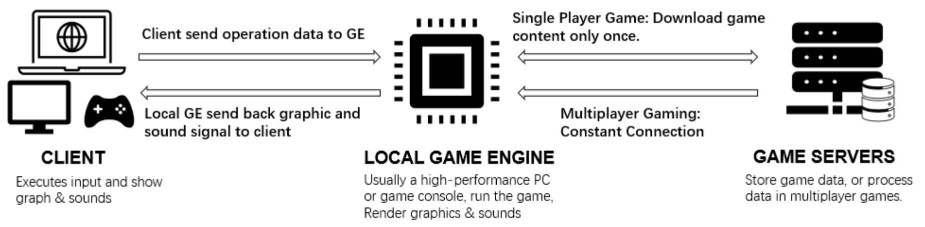

Client send operation data to GE
Local GE send back graphic and sound signal to client
Single Player Game: Download game content only once.
Multiplayer Gaming: Constant Connection
CLIENT
Executes input and show graph & sounds
LOCAL GAME ENGINE
Usually a high-performance PC or game console, run the game, Render graphics & sounds
GAME SERVERS
Store game data, or process data in multiplayer games.

Figure 2.1: An illustration of traditional game engine

Figure 2.1 represents the basic setup of a traditional single player game, where the game runs locally, except for the download phase. The user and the local game engine are physically in the same place. The user runs the game and sends commands to the game engine via the input device. The game engine renders the image and delivers it to the user. There is a variant of the traditional local game called an online game. The difference with stand-alone games is that part of the data exchange and computation of the game is given to the server, but the interaction and rendering of the game is still given to the local game engine. Usually online games serve multiple users at the same time.

2.3.3 Cloud gaming setup

Cloud gaming is another way of streaming, as Figure 2.2

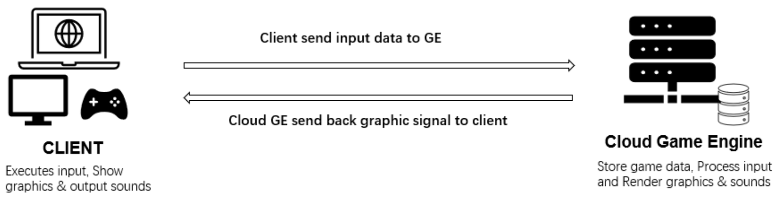

Client send input data to GE
Cloud GE send back graphic signal to client
CLIENT
Executes input, Show graphics & output sounds
Cloud Game Engine
Store game data, Process input and Render graphics & sounds

Figure 2.2: An illustration of cloud game engine

As we can see, allowing users to play video games remotely, the user does not need a high performance PC or console, the game runs on a remote server, the user sends commands to the server using any controller and the server streams the rendered image to the user. Compared to local gaming, cloud gaming adds the major processes of encoding, network transmission and decoding.

2.4 Limitations

While cloud services are currently considered adequate replacements for single and multi-player gaming on PC or consoles, most gamers will not feel the difference in experience between cloud and local gaming. However, there is a significant gap between the experience of cloud and local gaming in eSports games, where players are typically looking for input latency and frame rates, and players with lower latency usually have an advantage.

Cloud gaming services typically have more latency than local gaming (PC or console). From Figure 2.1 and Figure 2.2, we can see that user input is not processed directly on the client device, but is all sent over the network to the server, which then transmits the rendered graphics and generated sounds to the client, an action that is cycled through the entire service. This design inevitably introduces latency, in this case to the player's internet connection. Bandwidth and latency, as well as the physical distance to connect to the cloud gaming server, are the main keys. In addition, the encoding bitrate of cloud games is typically much higher than that of mass streaming due to the need to maintain minute detail in the picture, which also means that the bandwidth requirements for cloud gaming games are also much higher than for mass streaming, which is a significant challenge for the overall network.

2.5 Distribution system

As the saying goes, "many people are more powerful". Distributed computing mainly faces parallel computing. There are a group of networked component servers in a distributed system [11]. They can all run in parallel, and can also exchange information and coordinate their Actions, components interact with each other to achieve goals. Users generally do not perceive the logic behind this architecture, just like accessing a single server, the advantage is that more machines can be fully utilized, and more data can be processed in parallel, for component servers Faults are highly tolerant, and the amount of resources in the system can be flexibly increased or decreased.

2.6 Edge computing

Edge computing is not a specific technology, but an architecture, which is a way of distributed computing that brings computing and data storage closer to the data source [12]. This is expected to improve response time and save bandwidth. Since the server used by edge computing is very close to the user, the delay between the server and the user can be minimized. Edge computing is very meaningful for delay-sensitive cloud gaming services. Edge computing and cloud computing work together to significantly improve user QoE indicators.

The advantages of edge computing and distributed systems working together are as follows:

- Powerful scalability. Because edge computing runs on a distributed system, compared to servers in the data centre, edge computing servers can be different performance, different types of devices and are also very easy to replace.
- Robust reliability. Edge servers ensure high reliability. If a single node fails and becomes inaccessible, the user can switch to another edge server and the component servers in the distributed system ensure that the user's data is not lost.
- Fast: The responsiveness and throughput of cloud gaming services can be greatly improved by having edge servers very close to users. There are many games that must have low latency in order to have a good experience. For these games, edge computing has an advantage over cloud computing.
- Higher efficiency. As the edge servers are close to the users, many gaming processes can be run directly on the edge servers, which increases the efficiency of the service, saves cloud server resources and saves significantly on bandwidth, which can maintain high efficiency and improve QoE for users.

2.7 Related work

Based on the QoE evaluation of cloud gaming [13] by Asif Ali Laghari et al. the main factors known to affect QoE in cloud gaming are: bit rate, data rate, frame rate, throughput and network latency. , it is known that the main QoS factors affecting cloud gaming QoE are: bit rate, data rate, frame rate, throughput, packet loss and network latency. These parameters have an impact on the QoE of cloud gaming users. This thesis focuses on the network transmission component. The frame rate is related to the performance of the virtual machine assigned to the cloud service. The higher the performance of the virtual machine, the higher the frame rate will be. Bit rate is the number of bits transferred end-to-end per unit of time and can be used as a metric for engine design. Throughput is the number of bits transmitted by the network in a given time and is a parameter observed in actual tests. For gamers, a high frame rate represents smoother graphics, a high bit rate and throughput means sharper, more detailed and higher resolution graphics , the higher these parameters are, the better the gaming experience, which means that a tighter engine allocation and network quality is required to maintain a high QoE. Packet loss represents unrecoverable data loss, as in streaming media, the UDP protocol is often used in order to maintain connection speeds, and data loss in the UPD protocol is unrecoverable. Packet delay is the late arrival of data between the server and the user. Both packet loss and delay can lead to a dramatic drop in QoE. According to a study by Suznjevic et al [14] in 2013, QoE in the face of players in large-scale role-playing. simulated various loss and delay conditions and found that jitter and packet loss significantly reduced QoE, while delay also affected QoE, but no datagram was severely lost. In a 2003 study by Beigbeder et al [15], they analysed the impact of latency and packet loss on QoE in multiplayer games.

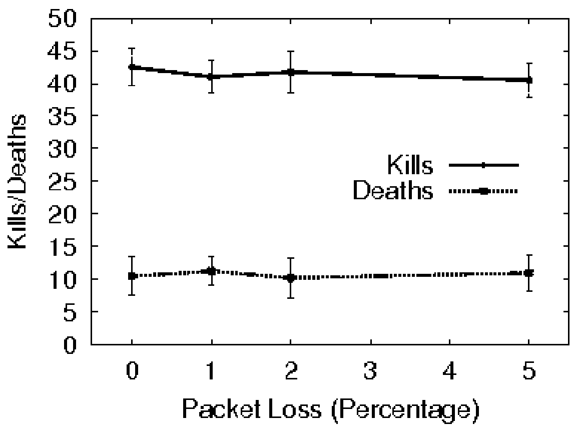


Figure 2.3: Impact of network packet loss on QoE [15]

As shown in the figure 2.3, as the packet loss rate increases, the number of player deaths increases and the number of kills decreases, as shown in the figure 2.4, as the latency increases, the number of player deaths increases significantly and the number of kills decreases significantly.

According to Lai, Phu and He, end-to-end latency is a key determinant of the QoE of the user experience, especially for latency-sensitive online or cloud games, and edge computing has emerged as a promising solution to the high latency problem [16].

Based on [17], CODEG was proposed by L. De Giovanni and Prof. Giaccone et al. It aims to provide a network-wide abstraction for each game engine.CODEG divides each GE into a decomposed set of modules, the Game Engine Modules (GEMs).The GEMs represent the basic elements that give the GEs and are activated when their function is activated when it is needed. The individual GEMs must be connected together to function properly. For a given game, the GEMs and their interconnections together constitute

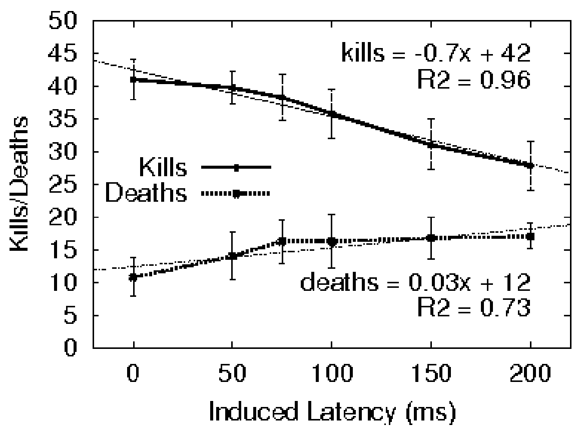


Figure 2.4: Impact of network delay on QoE [15]

the complete GE generated by CODEG. during execution, information is continuously exchanged between the GEMs to enable the GE processing workflow.

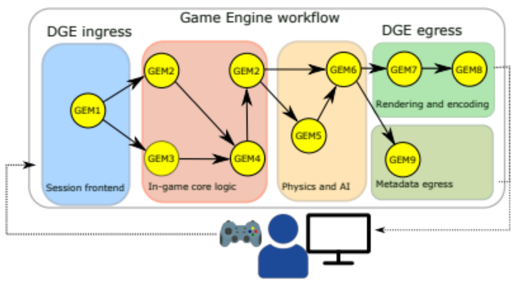


Figure 2.5: Game session workflow in CODEG [17]

Figure 2.5 shows an example of a CODEG instance, the set of GEMs needed to run a single game. Such instances may be associated with one or more game sessions or players. Each game session is assigned a unique pair of entry and exit GEMs. The ingress GEM is responsible for managing information for each session and allowing different sessions to be differentiated. Whenever player input is received at the ingress GEM, the information contained in the request is passed through the entire GE, while session-specific outputs are generated at the egress GEM. [17]

2.8 Summary

In this thesis, the focus is on simulating a distributed cloud gaming service where multiple gaming services can run simultaneously, each using the CODEG architecture, by combining multiple modules (GEMs) and placing them separately in available gaming network facilities, combining cloud and edge computing approaches to utilise network resources to achieve a complete architecture. The goal of the allocation algorithms is to meet the maximum response latency to reduce costs and ensure high QoE parameters per player.

Chapter 3

Design and implementation

3.1 Overview

In this chapter, we present an architecture where multiple gaming services can run concurrently, each using the CODEG architecture, by combining multiple modules (GEMs) with cloud computing and placing them individually into available network facilities, edge computing, and using network resources to achieve a complete architecture with allocation algorithms aimed at meeting maximum response latency to reduce costs and ensure high QoS. this chapter first discusses the overall architecture and then analyses some possible scenarios to be met. determine some of the parameters required for the next step of the simulation and finally illustrates the role of combining cloud and edge computing and its impact on the architecture.

3.2 Introduction

The level of QoE and QoS parameters can be used as an indicator of how well the system is performing. The most important QoS metric in traditional games is FPS, as FPS affects the regularity of the game [18] and is very easy to measure. In cloud gaming, FPS is guaranteed as long as the performance of each VM container is sufficient, so our proposed architecture does not consider the FPS per VM. we assume that FPS is guaranteed as long as each user is served, so only latency, bandwidth, number of users and resource consumption need to be considered. In this architecture, resource allocation is implemented in an adaptive form. If the user requests a compute node that does not have enough resources, the adaptive algorithms will try to adapt to the current situation.

Based on research areas such as cloud and edge computing, as well as QoS guarantees such as user latency guarantees, and load balancing guarantees for the entire architecture. This architecture will ensure a high QoS for users playing different types of games. architecture Combines different configurations to maintain a good QoS: high quality cloud server connections (cloud computing), high quality low latency edge server connections (edge computing), and when no network resources are available (adaptive). The cloud in the architecture diagram is represented by a server. The server can perform processing for the user and transmit the results to the client. Edge computing is represented by a computing node close to the user, which can also provide services to the user. Edge computing is a similar approach to cloud computing. The difference lies in the distance between the user and the computing resources. Adaptive algorithms are included in the system itself that regulate which server in the cloud service each user accesses and will strive to maintain a good QoS. If all network resources are exhausted throughout the system, the system will reject new users and add them to a waiting list. If new resources are available, users on the waiting list will be allowed to access the compute nodes on a first in first out (FIFO) basis. This process applies to all user scenarios.

The three main elements of the architecture are as follows.

1. the client (user). This includes information about the device the user is running on and the game being run, each game has different resource requirements.

2. The network. This includes everything between the client and the server, some of the less important devices will be omitted (e.g. routers, switches etc.).

3. Server side (cloud servers and edge servers). Servers that provide services, different devices have different amounts of resources, and the amount of available resources changes as users join.

This architecture needs to monitor changes in information to effectively improve QoS and QoE for users. game data, device data and network data are all values that need to be measured, such as the amount of resources required for a game run by the client, the resource usage of the server and the game latency of the user, all of which are stored in a database (DB) on each node. What is used in the data schema lock is discussed further in the DB. Based on this stored data, the allocation algorithms will adaptively assign users to the cloud gaming system and ensure QoS and QoE for the users.

The remainder of this chapter will go on to explain the architecture and then explain the adaptive algorithms and the possible scenarios and solutions, before the chapter moves on to explain the database and network and focus on the impact of each element on the architecture.

3.3 Architecture description

Figure 3.1 shows the proposed architecture. The aim of the architecture is to provide high QoS to all users through a distributed engine, independent of network and user conditions. The distributed resources are represented by CNs and ENs. These nodes can serve multiple TEs to provide high QoS. three elements (user, network and server) work together in the DB, each with its own attributes, and each server (cloud and edge) nodes will maintain the state of the users connected to itself and in the server's local DB Storing data related to its own state and network state. The adaptive system calls on each server's local DB to maintain the QoS of the entire system.

In this architecture, server resources are hierarchical and the logic is based on the level of distance between the server and the user. Layer 0 represents the edge nodes closest to the users. Their role is to provide the lowest delay, but each edge node has system resources and the lowest network bandwidth resources of all nodes connected to the user. Layer 1 represents the logically medium distance distributed cloud server Node. the server resources on layer 1 have more capacity than the edge nodes and can serve more edge nodes having more users with slightly higher network bandwidth resources between users, but also slightly higher latency. The Cloud Node on Layer 2 represents the core data centre for cloud gaming, which has the largest amount of resources and network bandwidth, but also has the highest latency of all servers.

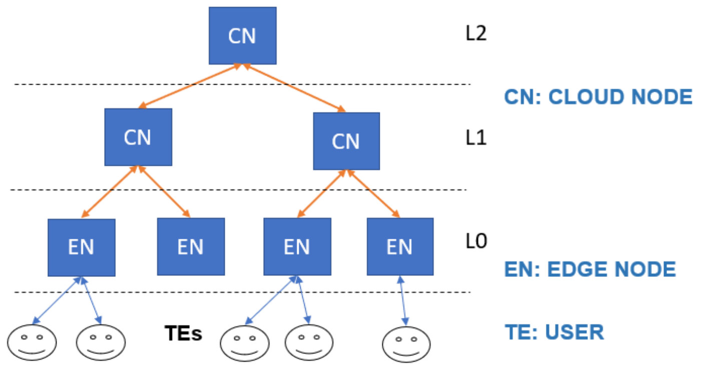


Figure 3.1: Proposed architecture of distributed engine

The network covers all devices between the client and the server, each with different functions and different connection types and speeds. The network devices feed their connection data and resource status to a DB and serve the adaptive algorithms to allocate available network resources, and if a server does not have the resources available to serve the client, the adaptive algorithms can transfer the user to other nodes on the network that have the required resources and reliable connectivity. The DB and the adaptive allocation algorithms are described in detail in the next section.

3.4 How to access server node

Figure 3.2 illustrate the logic about TE access server.

The GEM is a Docker-like container that can be thought of as a virtual machine that runs the game and renders the graphics. When an end device is allowed to access the service node, the service node will allocate the resources required for the TE to run the game to a new GEM and allow the end device to access the GEM, which will render the images and sound streamed by the GEM to the user in real time.

3.5 Database

In this section we will discuss the role of the database and the contents of the database in the different nodes. The role of the database is to store information about the device itself, as well as the games and networks that run in it. The adaptive allocation algorithms mentioned later will be all about the database. Integration is managed and allocation decisions are made.

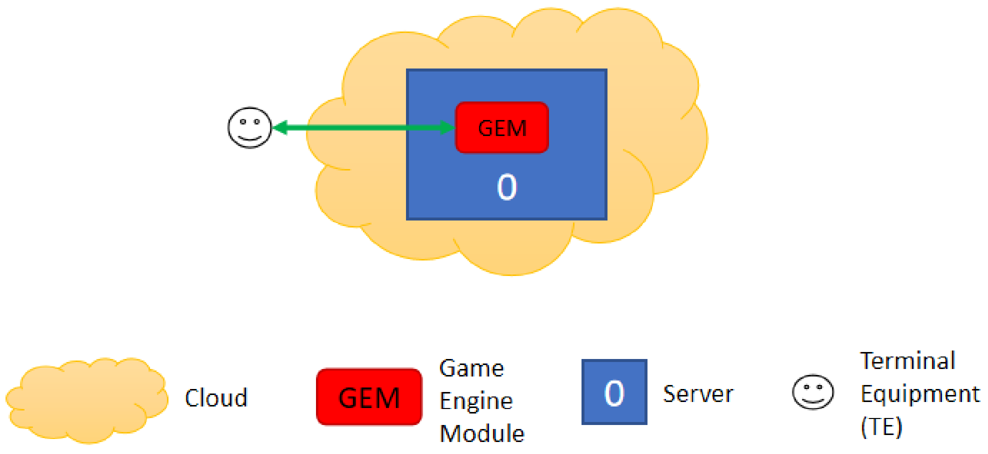


Figure 3.2: Terminal Equipment access Server

The database stores a lot of information related to the schema. The table below summarises the data required and where it is stored.

Uer(Te)	Edge Node	Cloud Node	Network
1.Resource Requirement of Game.(eg. CPU, RAM, Storage) 2.Network Resource Requirement 3.Delay Requirement (will decide weight with Users) 4.Bandwidth Requirement	1.Process Delay 2.CPU Usage 3.RAM Usage 4.Accept User List 5.Bandwidth Requirement	1.Process Delay 2.CPU Usage 3.RAM Usage 4.Accept User List 5.Bandwidth Requirement	1.RTT delay 2.Bandwidth

Table 3.1: The databases and information for this Architecture

In the user section:

1. The resource requirements of the game. Different games have different resource requirements. If the resources to run the game are not available, the game will be moved to run elsewhere. For example, a game A may require the use of 3Ghz CPU resources and 4GB of running memory.

2. Network resource requirements. Each user sends commands to the cloud game engine and the game engine's feedback screen requires network bandwidth.

3. Maximum delay requirements. Games need to stay below the maximum delay delay requirements for a good gaming experience, each game has a different maximum delay requirement and the delay affects the weighting of the game distribution.

4. Bandwidth requirements. Users send requests and actions to the Node, which require network bandwidth.

In Edge Node and Cloud Node: Since Edge Node and Cloud Node are only different in distance from the user, they have almost the same database.

1. Process Delay. The unavoidable delays caused by the processing time to deliver services to users.
2. CPU usage. The percentage of CPU being used, if this value increases too much it will affect the performance of the server and if this value reaches 100 percent the node will not be able to accept new users.
3. Memory usage. The percentage of running memory being used. If this value increases too much, it will also affect the performance of the server. If this value reaches 100 percent, the node will not be able to accept new users.
4. Accepted Users List. Displays a list of users that have joined the Node, used to display and count user data for the service.
5. Bandwidth requirements. Network bandwidth is required to stream images to users.

In the network section:

1. Link delay: The delay caused between nodes or between users and nodes.
2. Bandwidth: Sending data between users and nodes requires network bandwidth.

With this architecture, the first action is typically the user. The user first issues a command to the entire cloud gaming architecture, and the adaptive resource allocation algorithms works at this point, looking at the node's database and making a determination to assign the user to the appropriate node. The details of this are described in the next section.

3.6 Maximum delay limits

In the QoS measurement study for cloud gaming, network quality is an important metric that affects user QoE, with delay and packet loss being the most important. As packet loss is not considered in our ideal architecture, delay is the most important parameter in determining user QoE. [19]. Therefore, it is essential to set a maximum delay limit as a TE parameter to ensure that users entering the service node can maintain a high QoS.

3.7 Summary

This chapter discusses architectures that aim to improve QoE for users by providing overall QoS. The technologies currently in use (described in Chapter 2), including cloud and edge computing, are analysed to create an architecture that provides overall high performance and high QoS and QoE for users by combining their approaches and using resource adaptive allocation algorithms.

Chapter 4

Experimental method and setup

4.1 Overview

This chapter describes the methodology and system setup used to test the architecture presented in Chapter 3. The focus is on the decision making capabilities of the architecture (resource adaptive allocation algorithms). The decision within the architecture is to provide services to the TE based on metrics such as network conditions within the architecture and the occupancy of device resources, and to specify which Service Node helps which TE.

4.2 introduction

In Chapter 3, an architecture is presented that aims to provide high QoS for all running users in the engine. High QoS can be provided through the use of distributed resources, combined with cloud and edge computing. with technical improvements in hardware and software, it is also possible to increase the number of devices in the engine, especially edge computing devices, and the resources available. This distributed architecture can access the database of each node and aggregate it so that available resources in the nodes can be found and users can access these resources through adaptive allocation algorithms. The focus of this chapter is to develop the architecture and the adaptive algorithm into an experimental scenario and to explain the results of the study. First, Section 5.3 describes the behavioural logic of the service nodes. From the description in the previous chapter, it is clear that TEs can learn from other nodes when resources are insufficient. receiving help. Section 5.4 describes two resource adaptive allocation algorithms (Bottom-Up and Top-down) and Section 5.5 discusses how the overall QoS can be improved in this architecture.

4.3 Behavior of service node

In this architecture, edge nodes and cloud nodes are essentially service nodes; they are both processing-capable nodes, so their behaviour is similar. When the TE accesses the service node, the node checks its own resource state and network state and reads data (resource requirements and latency constraints) from the TE. If the resource requirements and latency constraints are satisfied, the service node will immediately serve the TE and perform the game flow. If the resource requirements or latency constraints are not met, the request is forwarded to an upper or lower tier server (which server depends on the resource allocation of the algorithm being used at the time).

4.4 Resource adaptive allocation algorithms

In this section, the resource adaptive allocation algorithms is described in detail.

Figure 4.1 below illustrates two different algorithms strategies, Bottom-Up and Top-Down, and their entire logic is very different.

And there are also different advanced algorithms for these two algorithms, which we generally call online algorithms. The high QoS requirements of all users are met by migrating the running GEMs and meeting the dynamic cloud game engine system where users can enter/exit dynamically. Parts of this new architecture will be presented in a paper by my supervisor Prof. Paolo Giaccone and colleagues Iman Lotfimahyari and others, which will be covered in future work.

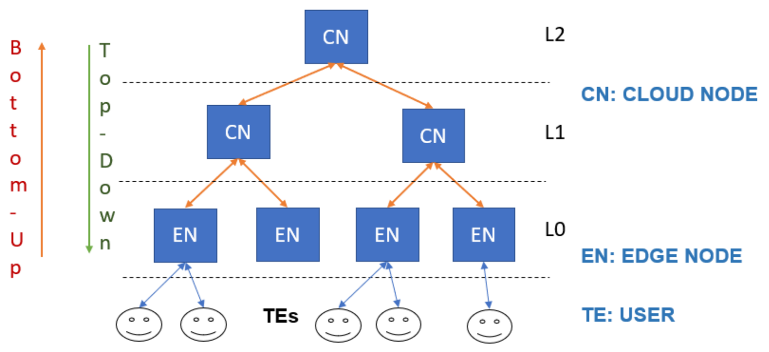


Figure 4.1: Adaptive algorithms and architecture of distributed engine

4.4.1 Bottom - Up

Figure 4.2 below shows the logic of the Bottom-Up offline algorithms.

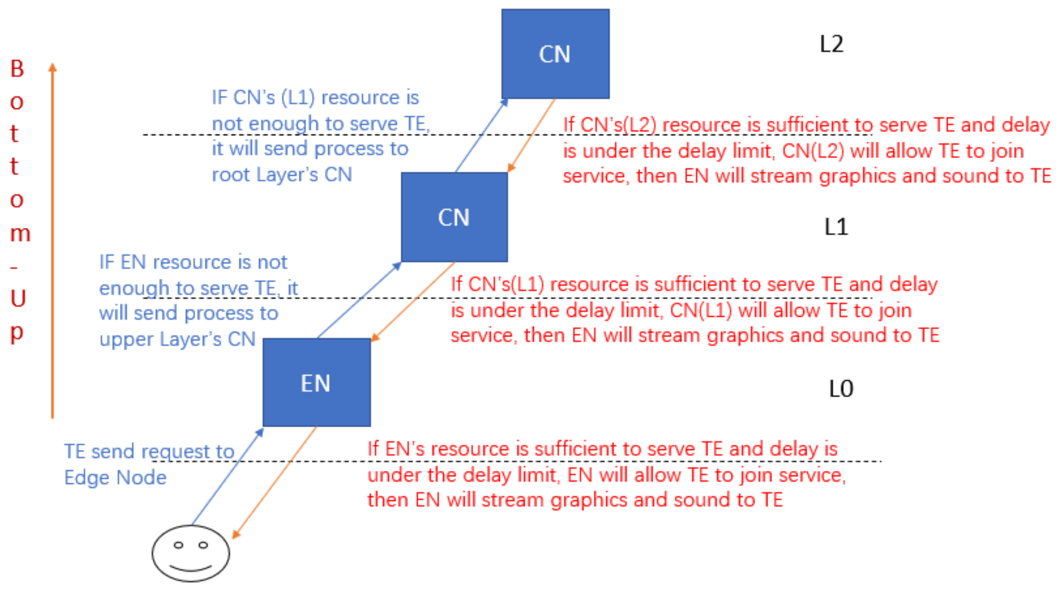

Bottom - Up
CN
L2
IF CN's (L1) resource is not enough to serve TE, it will send process to root Layer's CN
If CN's(L2) resource is sufficient to serve TE and delay is under the delay limit, CN(L2) will allow TE to join service, then EN will stream graphics and sound to TE
CN
L1
IF EN resource is not enough to serve TE, it will send process to upper Layer's CN
If CN's(L1) resource is sufficient to serve TE and delay is under the delay limit, CN(L1) will allow TE to join service, then EN will stream graphics and sound to TE
EN
L0
TE send request to Edge Node
If EN's resource is sufficient to serve TE and delay is under the delay limit, EN will allow TE to join service, then EN will stream graphics and sound to TE

Figure 4.2: Bottom-up algorithms

In the bottom-up offline algorithm, the user first initiates a service request to the nearest edge node. If the edge node has sufficient resources, delay less than the maximum delay limit and sufficient downlink bandwidth, the edge node can allow the TE to join and serve the TE, and the edge node will provide the TE with a stream of graphics signals and sound signals. If the edge node does not have sufficient resources or insufficient downlink bandwidth, the TE's request will be forwarded to the upper layer cloud node (L1). Similarly, if the cloud node (L1) has sufficient resources, latency less than the maximum latency limit and sufficient downlink bandwidth, the cloud node (L1) can allow TEs to join and serve TEs and the cloud node (L1) will serve the TE stream. It can transmit graphics signals and sound signals. If the cloud node (L1) does not have enough resources or insufficient downlink bandwidth, the TE request will be forwarded to the upper cloud node (L2). If the root node (cloud node L2) has sufficient resources and the latency is less than the maximum delay limit, the root node will provide streaming services to the TE, otherwise access to the user will be blocked. The loop runs until all users are in the loop.

algorithms 1 illustates the logic of Bottom-Up offline.

Algorithm 1 Bottom-UP offline

```
EdgeNode = EN, CloudNode(L1) = CN1, CloudNode(L2) = CN2
User = TE, ProcessDelay = ProcDelay, LinkDelay*2 = RTTDelay
totDelay = ProcDelay + RTTDelay
TE send Request to EdgeNode
if totDelay<=TE.DelayLimit and EN.Resource>=TE.Resource
and EN.bw>=TE.bw then                    ▷ Check resource, delay and bandwidth
    EN.Resource = EN.Resource-TE.Resource                    ▷ update resource
    EN.bw = EN.bw-TE.bw                                      ▷ update bandwidth
    CurrentNode = EN                                          ▷ now EN is server
    EN.games.append(TE.gameName,TE.totDelay)   ▷ update game total delay to DB
else
                                      ▷ if resource or bandwidth are not efficient
    CurrentNode = CN1                               ▷ Send TE's Request To CN1
    if totDelay<=TE.DelayLimit and CN1.Resource>=TE.Resource
and CN1.bw>=TE.bw then                   ▷ Check resource, delay and bandwidth
        CN1.Resource = CN1.Resource-TE.Resource              ▷ update resource
        CN1.bw = CN1.bw-TE.bw                                ▷ update bandwidth
        CN1.games.append(TE.gameName,TE.totDelay)  ▷ update game total delay to
DB
    else
                                      ▷ if resource or bandwidth are not efficient
        CurrentNode = CN2                           ▷ Send TE's Request To CN2
        if totDelay<=TE.DelayLimit and CN2.Resource>=TE.Resource
and CN2.bw>=TE.bw then                   ▷ Check resource, delay and bandwidth
            CN2.Resource = CN2.Resource-TE.Resource          ▷ update resource
            CN2.bw = CN2.bw-TE.bw                            ▷ update bandwidth
            CN2.games.append(TE.gameName,TE.totDelay)
                                               ▷ update game total delay to DB
        else
            return Fail                    ▷ Mark This User Cannot join in engine
        end if
    end if
end if
```

4.4.2 Top - Down

Figure 4.3 below shows the logic of the Top-Down offline algorithms.

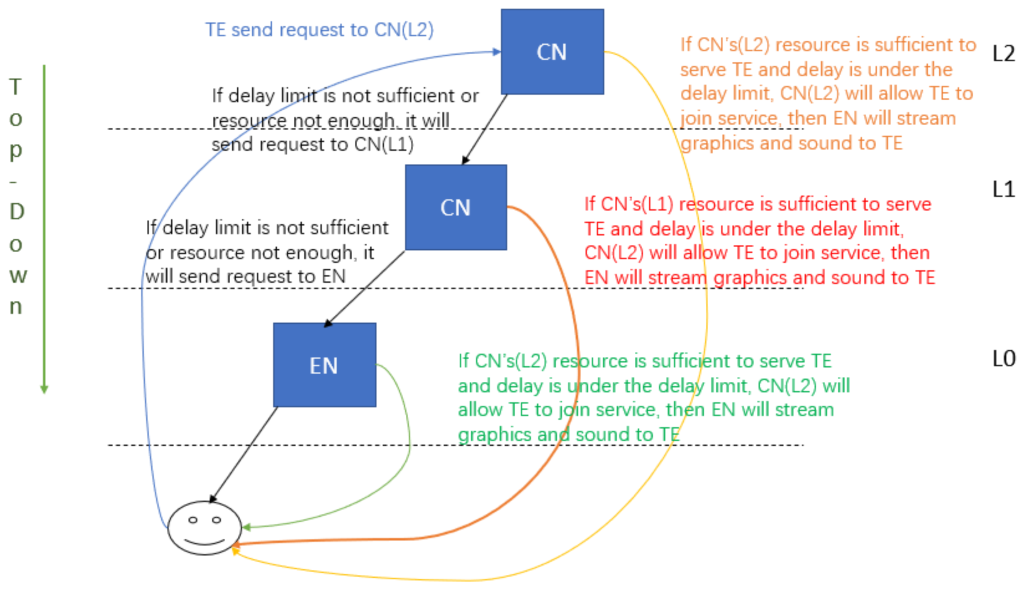

TE send request to CN(L2)
CN
If CN's(L2) resource is sufficient to serve TE and delay is under the delay limit, CN(L2) will allow TE to join service, then EN will stream graphics and sound to TE
L2
T o p - D o w n
If delay limit is not sufficient or resource not enough, it will send request to CN(L1)
CN
L1
If CN's(L1) resource is sufficient to serve TE and delay is under the delay limit, CN(L2) will allow TE to join service, then EN will stream graphics and sound to TE
If delay limit is not sufficient or resource not enough, it will send request to EN
EN
If CN's(L2) resource is sufficient to serve TE and delay is under the delay limit, CN(L2) will allow TE to join service, then EN will stream graphics and sound to TE
L0

Figure 4.3: Top-Down algorithms

In the top-down offline algorithm, the user first initiates a service request to the root server cloud node (L2). If the cloud node (L2) has sufficient resources with delay less than the maximum delay limit and sufficient downlink bandwidth, the cloud node (L2) may allow the TE to join and provide the TE with a stream of graphics and audio signals. If the latency of the TE connection to the cloud node (L2) is calculated or if the cloud node (L2) does not have sufficient resources or insufficient downlink bandwidth, the TE request is forwarded to the next level cloud node (L1). Similarly, if the cloud node (L1) has sufficient resources, delay less than the maximum delay limit and sufficient downlink bandwidth, the cloud node (L1) can allow TEs to join and serve TEs and the cloud node (L1) will serve the TE stream. It can transmit graphics signals and sound signals. If the delay of the TE connected to the cloud node (L1) is still above the maximum delay limit, or if there are insufficient resources or insufficient downlink bandwidth, the TE's request will be forwarded to the next level of the edge node. If the edge node has sufficient resources and the latency is less than the maximum delay limit, the edge node will provide streaming services to the TE, otherwise the user will be blocked from accessing. The loop runs until all users are in the loop.

algorithms 2 illustates the logic of Top-Down offline.

Algorithm 2 Top-Down offline

```
EdgeNode = EN, CloudNode(L1) = CN1, CloudNode(L2) = CN2
User = TE, ProcessDelay = ProcDelay, LinkDelay*2 = RTTDelay
totDelay = ProcDelay + RTTDelay
TE send Request to CN2
if totDelay<=TE.DelayLimit and CN2.Resource>=TE.Resource
and CN2.bw>=TE.bw then                ▷ Check resource, delay and bandwidth
    CN2.Resource = CN2.Resource-TE.Resource                ▷ update resource
    CN2.bw = EN.bw-CN2.bw                                 ▷ update bandwidth
    CurrentNode = CN2                                     ▷ now EN is server
    CN2.games.append(TE.gameName,TE.totDelay) ▷ update game total delay to DB
else
                          ▷ if delayLimit or resource or bandwidth are not efficient
    CurrentNode = CN1                              ▷ Send TE's Request To CN1
    if totDelay<=TE.DelayLimit and CN1.Resource>=TE.Resource
and CN1.bw>=TE.bw then                ▷ Check resource, delay and bandwidth
        CN1.Resource = CN1.Resource-TE.Resource            ▷ update resource
        CN1.bw = CN1.bw-TE.bw                             ▷ update bandwidth
        CN1.games.append(TE.gameName,TE.totDelay)  ▷ update game total delay to
DB
    else
                          ▷ if delayLimit or resource or bandwidth are not efficient
        CurrentNode = EN                            ▷ Send TE's Request To EN
        if totDelay<=TE.DelayLimit and EN.Resource>=TE.Resource
and EN.bw>=TE.bw then                 ▷ Check resource, delay and bandwidth
            EN.Resource = EN.Resource-TE.Resource          ▷ update resource
            EN.bw = EN.bw-TE.bw                           ▷ update bandwidth
            EN.games.append(TE.gameName,TE.totDelay)
                                        ▷ update game total delay to DB
        else
            return Fail                ▷ Mark This User Cannot join in engine
        end if
    end if
end if
```

4.5 Simulation data-set

Due to the needs of the comparative experiment, my colleague Iman Lotfimahyar and I use the same three sets of user data, namely high-rate, Mid-rate and Low-rate, each user in each set of data has a different size of game type , and different game types have different resource requirements and maximum delay limits. The link delay between all service nodes is fixed at 5ms, and the processing delay between edge nodes and cloud nodes is fixed at 10ms. And in the simulation we will use CPU resources as the main representative. We will change the server resources at each level in multiple experiments, and there are five different allocation strategies in the Bottom-Up and Top-Down algorithms:

1. Random distribution
2. High-CPU first
3. Low-CPU first
4. High-delay first
5. Low-delay first

4.6 Implementation of the model

This section is dedicated to explaining the core components of the architecture and simulations mentioned in the paper. Each component in the scheme is explained in the following subsections.

4.6.1 Design logic of the model

Figure 4.4 illustrates the design logic of the model.

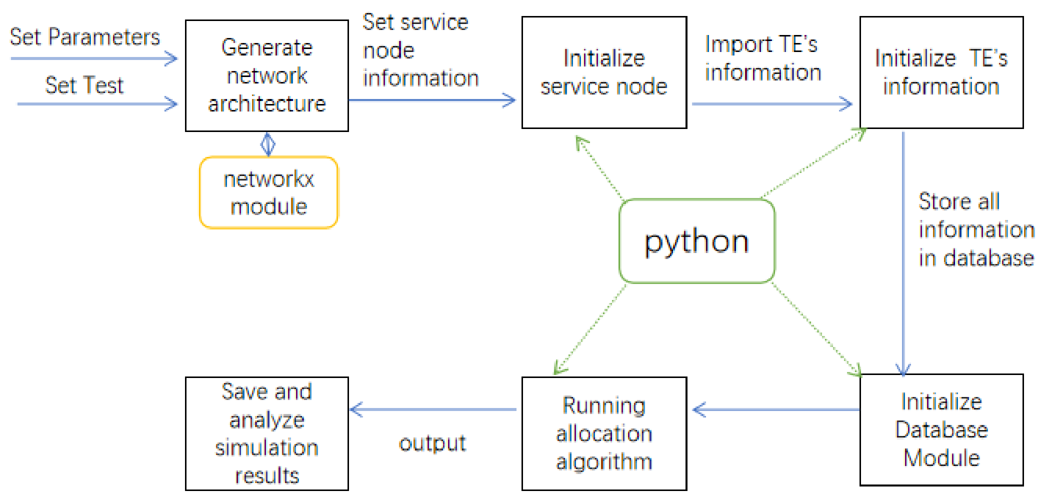

Set Parameters
Set Test
Generate network architecture
networkx module
Set service node information
Initialize service node
Import TE's information
Initialize TE's information
Store all information in database
python
Initialize Database Module
Running allocation algorithm
output
Save and analyze simulation results

Figure 4.4: Design logic of the model

In the architecture simulation, python is used as the development language, and the python extension package Networkx is used when generating the network. At the beginning of the simulation, first set the parameters and the number of tests, use python to traverse each service node and the network between the nodes, then initialize the database data of the service node, then import the TE data group generated by the computer, and then use the TE data group to initialize Go to the TE database, and then run the adaptive allocation algorithms from the first TE until the end of the simulation. Finally, analyze the simulation results.

4.6.2 Network implementation

In the network architecture generation part, using the networkx package (based on python), it is implemented by traversing each node and connecting them. The traversal process requires the use of the numpy package.

```
import networkx as nx
import numpy as np

def BuildNetwork():
G = nx.Graph() # networkx initialize
nodes.append(CN(6, node_kind[2], 2*(depth)*link_dly)) # initialize root
nodes[0].layer=2 # set node into root
top_node.append(nodes[0]) # top node is the root
G.add_node(nodes[0].ID) # add root to the graph
up_lim = 0
for level in range(depth): # loop over each node level
    n_level = n**level # number of nodes on a given level
    low_lim = up_lim + 1 # index of first node on a given level
    up_lim = up_lim + n_level # index of last node on a given level
    for i in range(n_level): # loop over nodes (parents) on a given level
        parentID = 6 - (low_lim + i - 1) #CloudNode(low_lim+i,
            node_kind[level+1], 2*(depth-level-1)*link_dly)
        offset = up_lim + i*n + 1 # index pointing to node just before first
            child
        for k in range(n): # loop over children for a given node (parent)
            j = n - k
            child = CN(5 - offset + j, node_kind[depth - level - 1], 2*(depth -
                level -1)*link_dly) # initialize child
            child.parentID = parentID # set parent ID
            child.layer = depth-level-1 # set layer
            nodes.append(child) # add child to the list of nodes
            nodes[low_lim+i-1].childIDs.append(5-offset+j) # add child ID to the
                list of children of the parent
            if level==depth-1:
                edge_layer.append(child)
            edges.append((parentID,child.ID))
            G.add_node(child.ID)
            G.add_edge(nodes[low_lim+i-1].ID, child.ID)
```

4.6.3 Database implementation

The database part is divided into two directions, service nodes and user nodes. The method of assigning attributes is to use python's class-oriented programming. When initializing the network, assign values to the class of each node, and the algorithms will call the value of each node. The corresponding code is as follows:

```
class CN:
    def __init__(self, ID, info, delay):
        self.ID = ID
        self.layer = info[0]
        self.parentID = 0.5 # Means that is root(Cloud L2)
        self.childIDs = [] # initialize child
        self.ProcDly = info[1]
        self.cpu = info[2] # CPU resources
        self.RTTdlyForUE = delay
        self.games = [] # store accessed games
```

In the user part, the method of assigning attributes is similar to that of the service node. First, the user list in text format is converted into a list, and then they are assigned one-to-one correspondence with the generated users.

```

games = [list(map(float, line.strip('\n').split(','))) for line in
    open('9_P0.txt')] # Convert text to list

class Player:
    def __init__(self, info):
        self.migrated = 0
        self.name = info[0]
        self.ID = info[0]
        self.GEM = info[1]
        self.TotDly = info[2]# Delay constraint
        self.cpu = info[3]
        self.expDly = 0
```

4.7 algorithms implentation

According to the logic of the custom allocation algorithms in the previous section, the algorithms can be easily implemented. The custom allocation will first read the data of the TE that is about to access the service, and determine whether to join the service node or which service node to join.

So here is a function written in python, which is triggered when the user who enters the node meets the conditions, the service node will record the user who enters the node and update the resource balance in the database.

```
# In the example, cpu is used to refer to the node's resources.
def acceptPlayer(self, player, test): # test = [game, max delay, Cpu]
```

```
a = self.ProcDly+self.RTTdlyForUE # a = ProcDly + RTTdlyForUE
if ((a <= player.TotDly) and (self.cpu >= player.cpu)):
    if test==0:
        self.cpu = self.cpu-player.cpu
        self.games.append([player.name,player.TotDly])
    return 1
else:
    return 0
```

At the beginning of the whole simulation, since the algorithms needs to be repeated many times in one experiment, a variable next_experiment is defined at the beginning of the experiment, and TE will access the engine according to the order of the list. If the TE is allocated to the service node by the custom allocation algorithms, the acceptPlayer function is called and the user is marked as having accessed the service, and the next user in the list is called. If the user does not meet the conditions for accessing the service node, follow the resource allocation algorithms to enter the upper/lower node. The operation loops until all users are called. At this point, next_experiment will be changed to true, and while next_experiment = true is satisfied, the network and service node properties will be reinitialized and the algorithms will be restarted and the loop will start. When the loop reaches the preset number of experiments, the flag next_experiment is false and the process is terminated.

Chapter 5

Experimental evaluation

The last chapter is dedicated to the analysis of the results of the different experiments. Since the structure of the experiments has already been discussed. Therefore, this chapter is only concerned with the experimental results. All experimental results are evaluated using tables and matrices. After the experiment, the results will be compared with those of my colleague iman, and the data from the traditional cloud game engine will be added for comparison.

5.1 Methodology

In this experiment, three sets of user data (high rate, medium rate and low rate) are used to represent the operation of the game engine under different load conditions for comparison with the results of later experiments. To simplify the description, all ServeNodes are numbered. edgeNodes for L0 are Node0 to Node4 respectively, CloudNodes for L1 are Node4 and Node5, and CloudNodes for L2 are Node6. The same constants for all experiments are: processing delay for all nodes is set to 10ms, connection between nodes The delay is set to 5ms, the upstream and downstream bandwidth between both TE and EdgeNode is 25 Mbps, and the upstream and downstream bandwidth between all EdgeNodes and CloudNode is 100 Mbps. the bandwidth between CLoudNode (L1) and the highest level CloudeNode (L2), which is theoretically unlimited, is actually set to 1, 000,000 Mbps. 50 Mbps of downlink bandwidth is used by the service node to provide streaming services to each user, while 0.1 Mbps of uplink bandwidth is required for each user requesting to join the service node. Each user is placed sequentially in each ServiceNode in the engine in the order in which they are listed. When all users have been tried, they are re-initialised for the next experiment. This process will be repeated 100 times in a single experiment to reduce errors and to eliminate eventualities.

Here we introduce the concept of Delay-slack, which is expressed as AvgDelayslack=AvgMaxDelay−AvgDel
where AvgMaxDelay represents the maximum average delay that users can tolerate and AvgDelay represents the average actual delay of users.

5.2 Experiments with first setting

The following Figure 5.1 shows the first set of data, it is set to 250GHz total CPU resources per EdgeNode for L0 and 1000GHZ total resources per Node for L1. L2 has theoretically unlimited resources, so it is set to 10000000GHz.

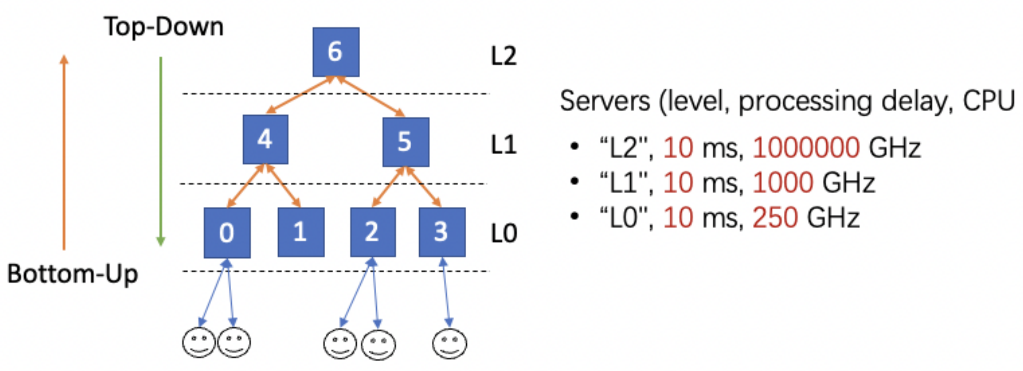


Figure 5.1: First Simulation Structure

In the first experiment, there are two sections, Bottom-Up and Top-Down, depending on the algorithms, and each section is divided into Random, High-CPU first, Low-CPU first, High-Delay first, and Low-Delay first, depending on the allocation strategy of the species.

5.2.1 Simulation result with high-rate user

In the same experimental environment, the AvgDelay for users in the traditional cloud game engine is 30ms and AvgDelaySlack is 8.53ms, which can serve all users under ideal conditions, but requires much higher hardware resources than the distributed cloud game engine, and users who need low delay to have a good QoE game will have a very poor gaming experience.

BU-Random with high-rate and first set

The following figure 5.2 represents the results of the simulation performed by the high-rate player in the BU-Random strategy. The BU-Random policy means that users access the cloud game engine in random order.

After 100 iterations, an average of 955 users were allowed to access the service node each time, the average delay of users was 14.96ms, the average delay of Slack was 28.94ms, and an average of 202.35 users were unable to access the service, with a failure rate of approximately 17 percent.These are summarised in a table later.

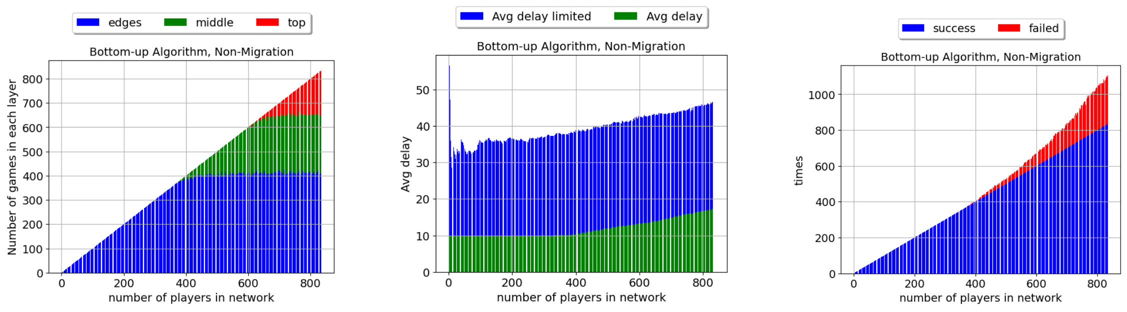


Figure 5.2: BU-Random with high-rate and first set

BU-CPU-HighFirst with high-rate and first set

The following figure 5.3 represents the results of the simulation performed by the high-rate player in the BU-CPU-HighFirst strategy.The BU-CPU-HighFirst policy means that users access the cloud game engine in order of largest to smallest CPU usage.

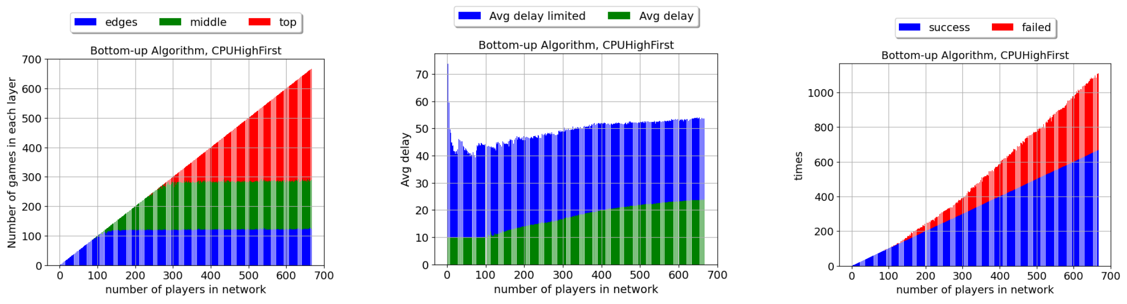


Figure 5.3: BU-CPU-HighFirst with high-rate and first set

After 100 iterations, an average of 855 users were allowed to access the service node each time, the average delay of users was 17.97ms, the average delay of Slack was 29.06ms, and an average of 306 users were unable to access the service, a failure rate of approximately 26 percent. These are summarised in a table later.

BU-CPU-LowFirst with high-rate and first set

The following figure 5.4 represents the results of the simulation performed by the high-rate player in the BU-CPU-LowFirst strategy. The BU-CPU-LowFirst policy policy means that users access the cloud game engine in order of smallest to largest CPU usage.

After 100 iterations of the experiment, an average of 1064 users were allowed to access the service node each time, the average delay of the users was 29.08ms, the average delay Slack was 29.77ms, and an average of 96 users were unable to access the service, with a failure rate of approximately 8 percent. These are summarised in a table later.

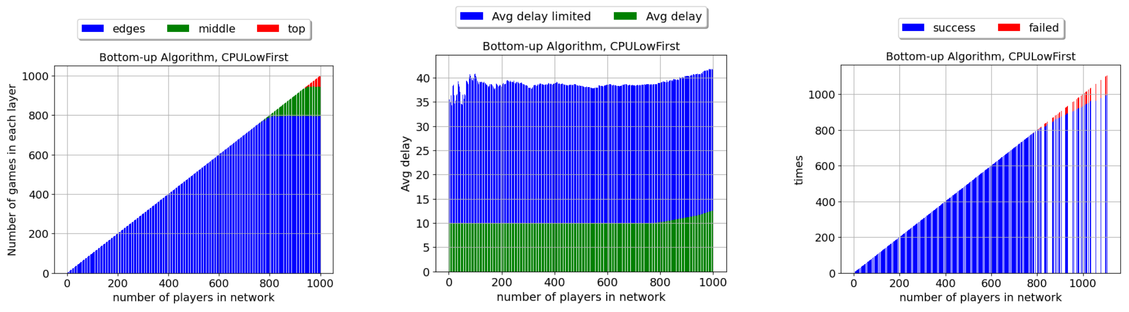


Figure 5.4: BU-CPU-LowFirst with high-rate and first set

BU-Delay-HighFirst with high-rate and first set

The following figure 5.5 represents the results of the simulation performed by the high-rate player in the BU-Delay-HighFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of highest to lowest user tolerable delay.

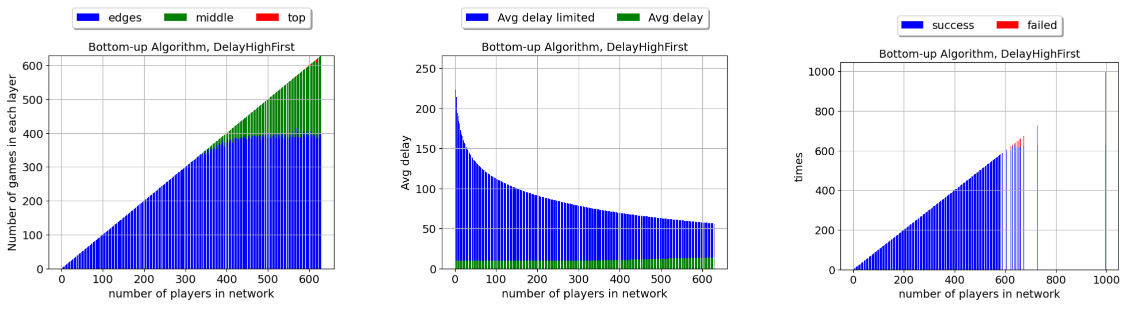


Figure 5.5: BU-Delay-HighFirst with high-rate and first set

After 100 iterations of the experiment, an average of 791 users were allowed to access the service node each time, with an average delay of 13.21ms, an average delay Slack of 36.42ms, and an average of 396 users unable to access the service, for a failure rate of roughly 32 percent. These are summarised in a table later.

BU-Delay-LowFirst with high-rate and first set

The following Figure 5.6 represents the results of the simulation performed by the high-rate player in the BU-Delay-LowFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of lowest to highest user tolerable delay.

After 100 iterations, an average of 1160 users were allowed to access the service node each time, with an average delay of 15.85ms and an average delay Slack of 22.68ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

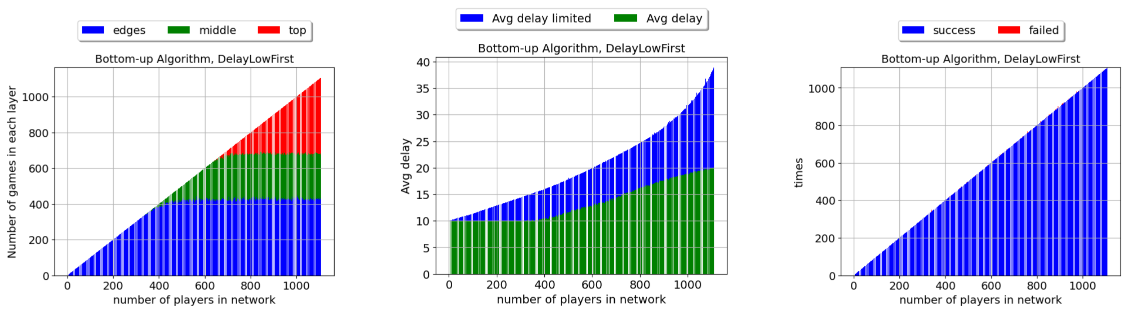


Figure 5.6: BU-Delay-LowFirst with high-rate and first set

TD-Random with high-rate and first set

The following Figure 5.7 below shows the results of a simulation performed by a high-rate player under the TD-Random strategy, which means that the cloud game engine is accessed in a random order.

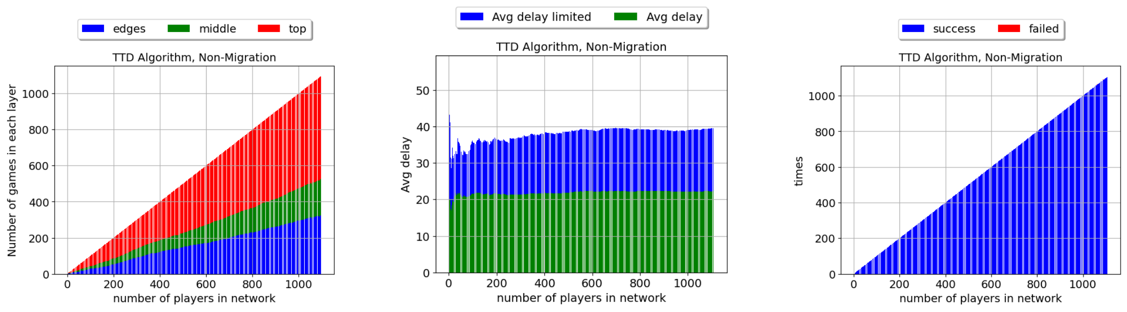


Figure 5.7: TD-Random with high-rate and first set

After 100 iterations, an average of 1160 users were allowed to access the service node each time, with an average delay of 21.38ms and an average delay Slack of 17.15ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

TD-CPU-HighFirst with high-rate and first set

The following Figure 5.8 below shows the simulation results for high-rate players under the TD-CPU-HighFirst policy, which means that the Cloud Game Engine is accessed from the highest to the lowest user CPU usage.

Very similar to TD-Random, after 100 iterations of the experiment, an average of 1160 users were allowed to access the service node each time, with an average delay of 21.38ms and an average delay Slack of 17.15ms. There were no users who could not access the service, so the failure rate was ZERO. This is summarised in a table later.

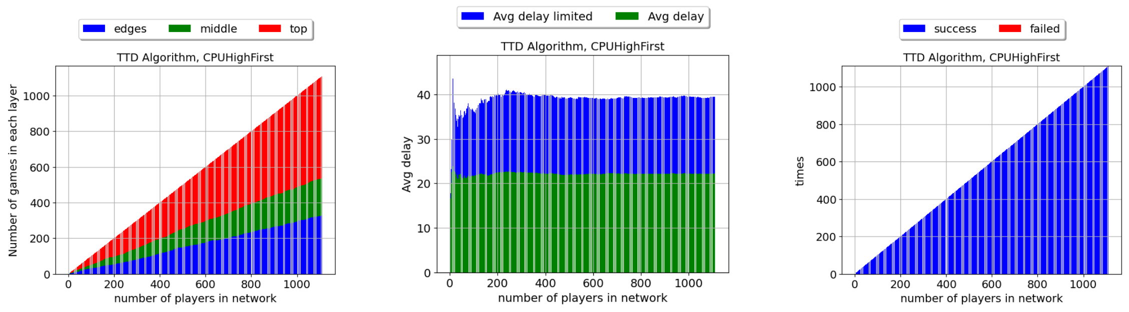


Figure 5.8: TD-CPU-HighFirst with high-rate and first set

TD-CPU-LowFirst with high-rate and first set

The following Figure 5.9 shows the simulation results for high-rate players under the TD-CPU-LowFirst policy. tTD-CPU-LowFirst policy means that the Cloud Game Engine is accessed from the lowest to the highest user CPU usage.

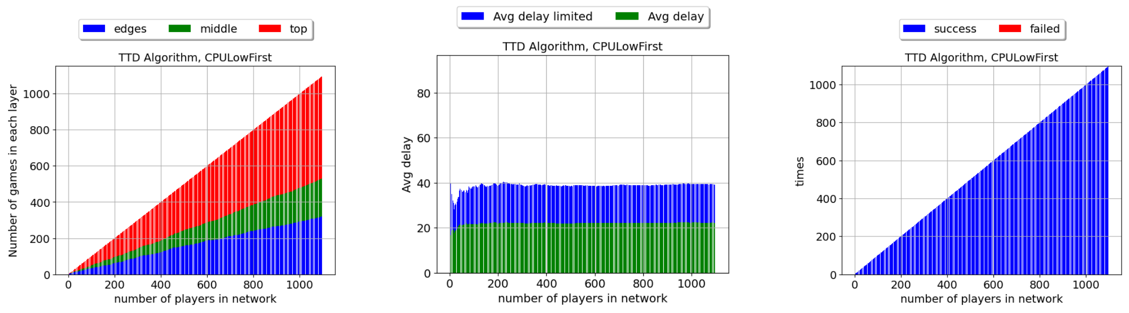


Figure 5.9: TD-CPU-LowFirst with high-rate and first set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.38ms, the average delay Slack is 17.15ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-HighFirst with high-rate and first set

The following Figure 5.10 represents the simulation results for high-rate players under the TD-Delay-HighFirst policy, which means that the Cloud Game Engine is accessed in descending order of the maximum delay tolerated by the user.

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.38ms, the average delay Slack is 17.15ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

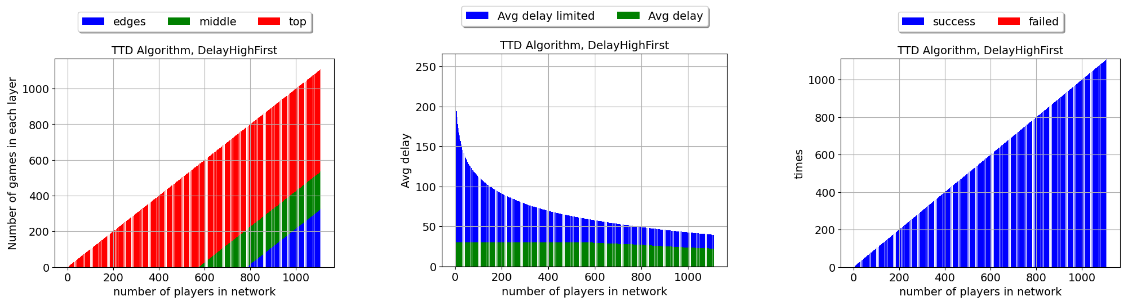


Figure 5.10: TD-Delay-HighFirst with high-rate and first set

TD-Delay-LowFirst with high-rate and first set

The following Figure 5.11 shows the simulation results for high-rate players under the TD-Delay-LowFirst policy, which means that the Cloud Game Engine is accessed in descending order of the highest delay tolerated by the user.

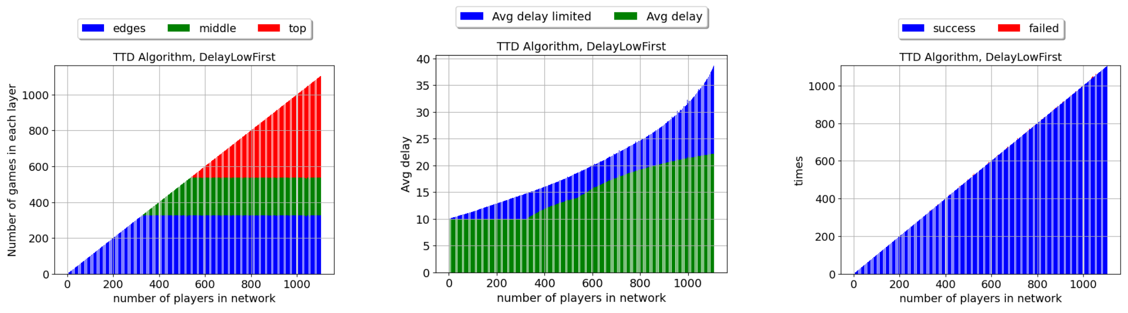


Figure 5.11: TD-Delay-LowFirst with high-rate and first set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.38ms, the average delay Slack is 17.15ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

5.2.2 Simulation result with mid-rate user

In the same experimental environment, the AvgDelay for users in the traditional cloud game engine is 30ms and AvgDelaySlack is 7.38ms, which can serve all users under ideal conditions, but requires much higher hardware resources than the distributed cloud game engine, and users who need low delay to have a good QoE game will have a very poor gaming experience.

BU-Random with Mid-rate and first set

The following figure 5.12 represents the results of the simulation performed by the mid-rate player in the BU-Random strategy. The BU-Random policy means that users access the cloud game engine in random order.

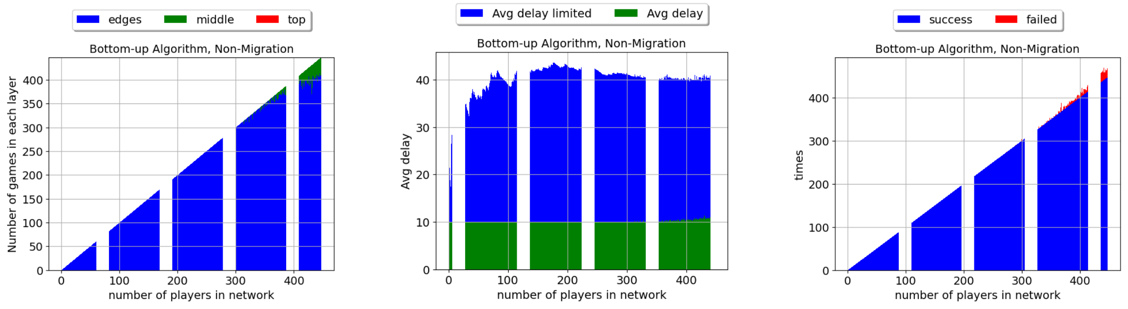


Figure 5.12: BU-Random with Mid-rate and first set

After 100 iterations, an average of 504 users were allowed to access the service node each time, the average delay of users was 10.45ms, the average delay of Slack was 27.49ms, and an average of 10.54 users were unable to access the service, with a failure rate of approximately 2 percent.These are summarised in a table later.

BU-CPU-HighFirst with Mid-rate and first set

The following figure 5.13 represents the results of the simulation performed by the mid-rate player in the BU-CPU-HighFirst strategy.The BU-CPU-HighFirst policy means that users access the cloud game engine in order of largest to smallest CPU usage.

After 100 iterations, an average of 474 users were allowed to access the service node each time, the average delay of users was 12.49ms, the average delay of Slack was 27.2ms, and an average of 48 users were unable to access the service, a failure rate of approximately 9 percent. These are summarised in a table later.

BU-CPU-LowFirst with Mid-rate and first set

The following figure 5.14 represents the results of the simulation performed by the mid-rate player in the BU-CPU-LowFirst strategy. The BU-CPU-LowFirst policy policy means that users access the cloud game engine in order of smallest to largest CPU usage.

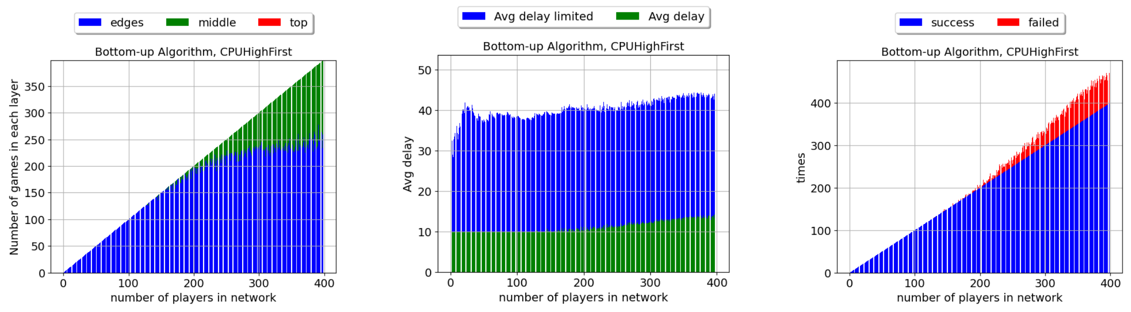


Figure 5.13: BU-CPU-HighFirst with Mid-rate and first set

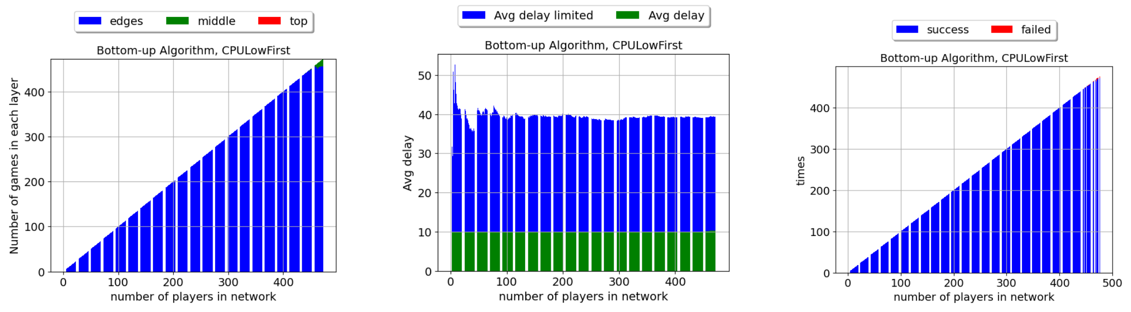


Figure 5.14: BU-CPU-LowFirst with Mid-rate and first set

After 100 iterations of the experiment, an average of 517 users were allowed to access the service node each time, the average delay of the users was 10.19ms, the average delay Slack was 27.41ms, and an average of 5 users were unable to access the service, with a failure rate of approximately 1 percent. These are summarised in a table later.

BU-Delay-HighFirst with Mid-rate and first set

The following figure 5.15 represents the results of the simulation performed by the mid-rate player in the BU-Delay-HighFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of highest to lowest user tolerable delay.

After 100 iterations of the experiment, an average of 478 users were allowed to access the service node each time, with an average delay of 10ms, an average delay Slack of 29.69ms, and an average of 44 users unable to access the service, for a failure rate of roughly 8 percent. These are summarised in a table later.

BU-Delay-LowFirst with Mid-rate and first set

The following figure 5.16 represents the results of the simulation performed by the mid-rate player in the BU-Delay-LowFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of lowest to highest user tolerable delay.

After 100 iterations, an average of 522 users were allowed to access the service node each time, with an average delay of 10.82ms and an average delay Slack of 26.56ms. There

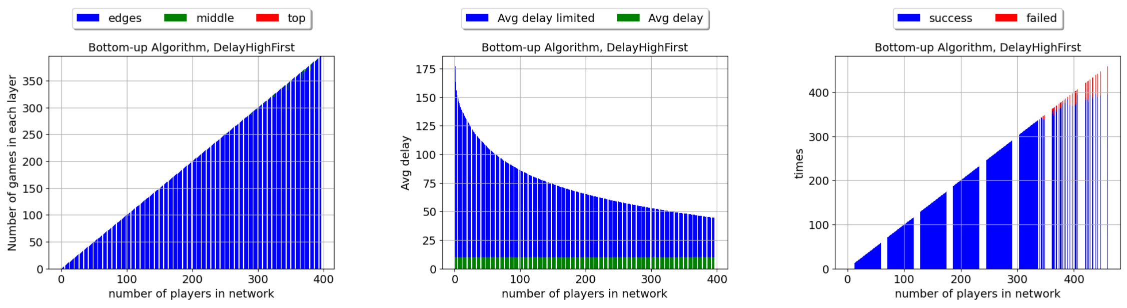


Figure 5.15: BU-Delay-HighFirst with Mid-rate and first set

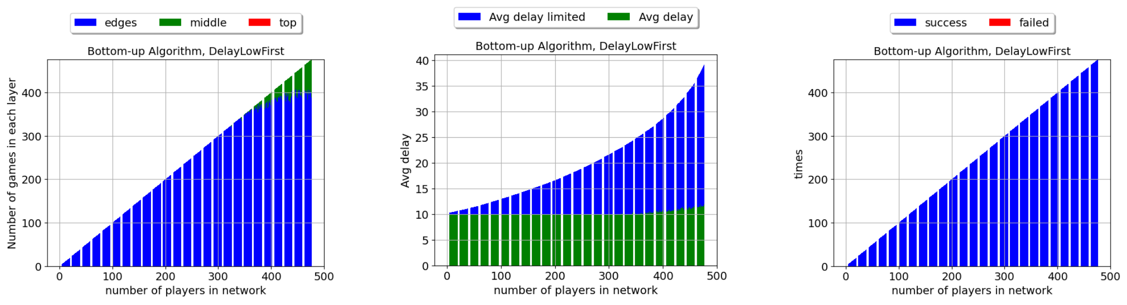


Figure 5.16: BU-Delay-LowFirst with Mid-rate and first set

were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

TD-Random with Mid-rate and first set

The following Figure 5.17 below shows the results of a simulation performed by a mid-rate player under the TD-Random strategy, which means that the cloud game engine is accessed in a random order.

After 100 iterations, an average of 522 users were allowed to access the service node each time, with an average delay of 21.95ms and an average delay Slack of 15.43ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

TD-CPU-HighFirst with Mid-rate and first set

The following Figure 5.18 below shows the simulation results for mid-rate players under the TD-CPU-HighFirst policy, which means that the Cloud Game Engine is accessed from the highest to the lowest user CPU usage.

Very similar to TD-Random, after 100 iterations of the experiment, an average of 522 users were allowed to access the service node each time, with an average delay of 21.95ms and an average delay Slack of 15.43ms. There were no users who could not access the service, so the failure rate was ZERO. This is summarised in a table later.

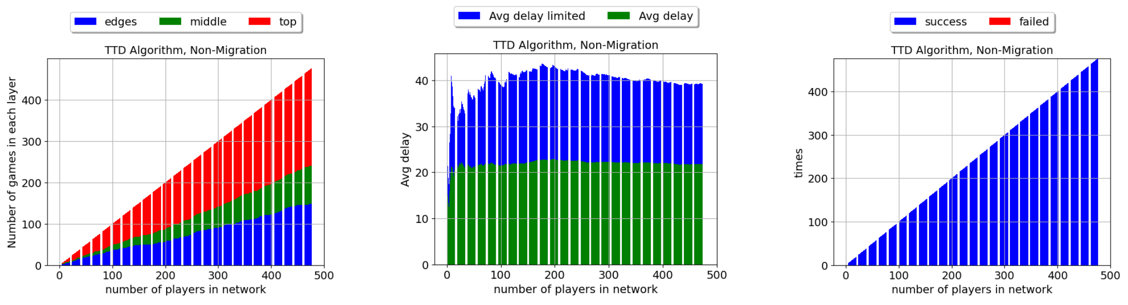


Figure 5.17: TD-Random with Mid-rate and first set

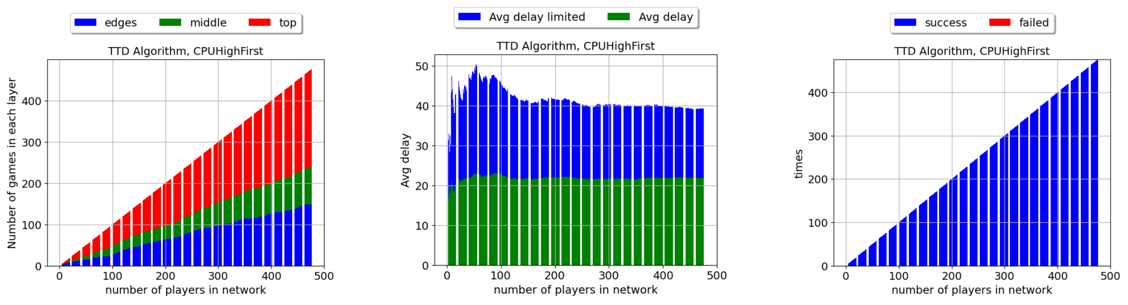


Figure 5.18: TD-CPU-HighFirst with Mid-rate and first set

TD-CPU-LowFirst with Mid-rate and first set

The following Figure 5.19 shows the simulation results for mid-rate players under the TD-CPU-LowFirst policy. tTD-CPU-LowFirst policy means that the Cloud Game Engine is accessed from the lowest to the highest user CPU usage.

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.95ms, the average delay Slack is 15.43ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-HighFirst with Mid-rate and first set

The following Figure 5.20 represents the simulation results for mid-rate players under the TD-Delay-HighFirst policy, which means that the Cloud Game Engine is accessed in descending order of the maximum delay tolerated by the user.

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.95ms, the average delay Slack is 15.43ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

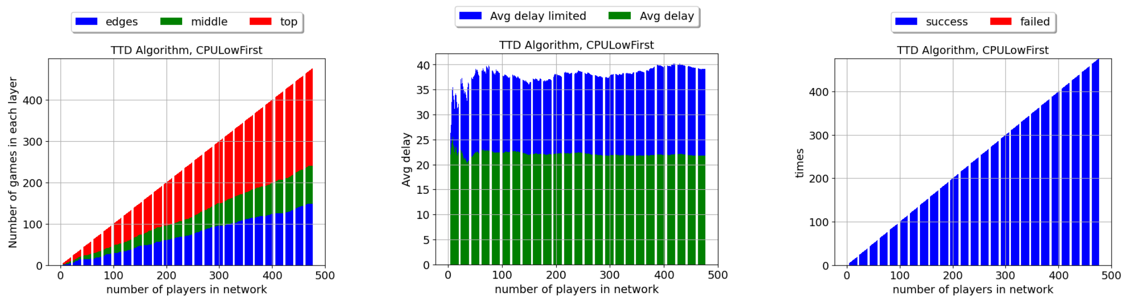


Figure 5.19: TD-CPU-LowFirst with Mid-rate and first set

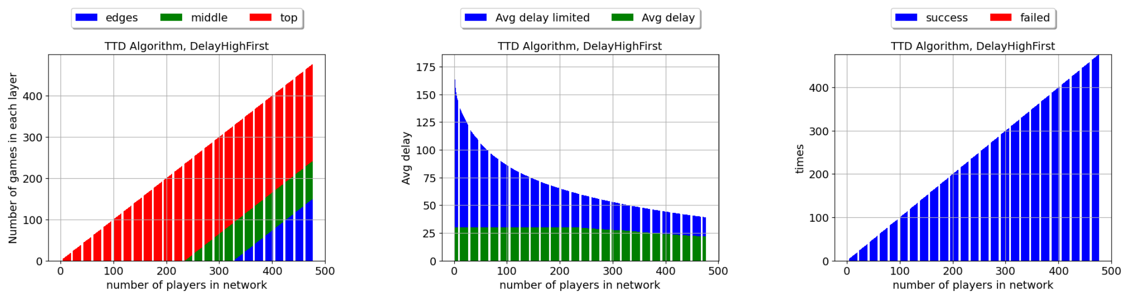


Figure 5.20: TD-Delay-HighFirst with Mid-rate and first set

TD-Delay-LowFirst with Mid-rate and first set

The following Figure 5.21 shows the simulation results for mid-rate players under the TD-Delay-LowFirst policy, which means that the Cloud Game Engine is accessed in descending order of the highest delay tolerated by the user.

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.95ms, the average delay Slack is 15.43ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

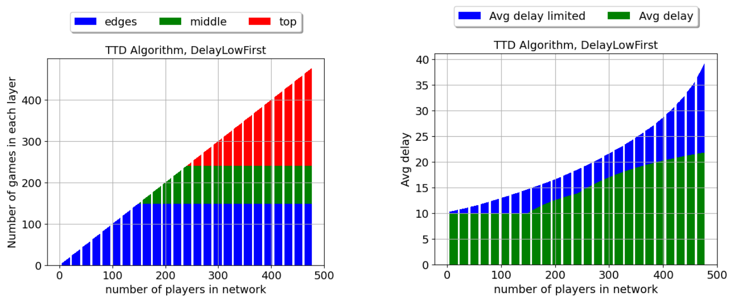


Figure 5.21: TD-Delay-LowFirst with Mid-rate and first set

5.2.3 Simulation result with low-rate user

In the same experimental environment, the AvgDelay for users in the traditional cloud game engine is 30ms and AvgDelaySlack is 12.968ms, which can serve all users under ideal conditions, but requires much higher hardware resources than the distributed cloud game engine, and users who need low delay to have a good QoE game will have a very poor gaming experience.

BU-Random with Low-rate and first set

The following figure 5.22 represents the results of the simulation performed by the Low-rate player in the BU-Random strategy. The BU-Random policy means that users access the cloud game engine in random order.

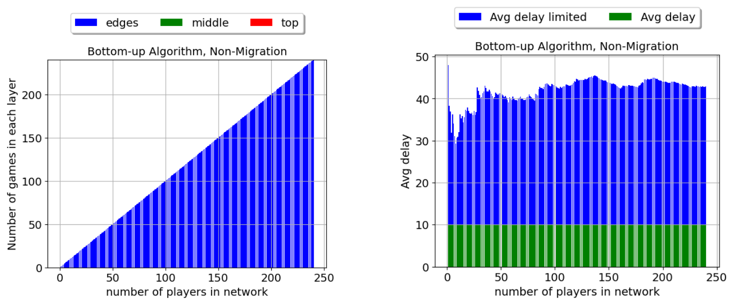


Figure 5.22: BU-Random with Low-rate and first set

After 100 iterations, an average of 240 users were allowed to access the service node each time, the average delay of users was 10ms, the average delay of Slack was 32.95ms, and have no users were unable to access the service, with failure rate is zero.These are summarised in a table later.

BU-CPU-HighFirst with Low-rate and first set

The following figure 5.23 represents the results of the simulation performed by the Low-rate player in the BU-CPU-HighFirst strategy.The BU-CPU-HighFirst policy means that users access the cloud game engine in order of largest to smallest CPU usage.

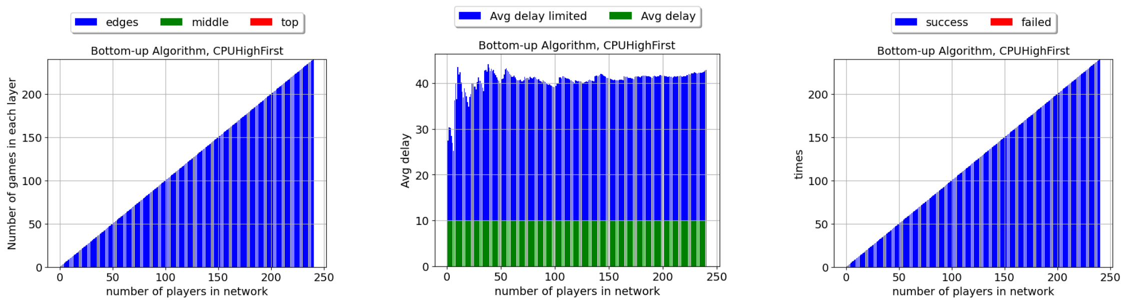


Figure 5.23: BU-CPU-HighFirst with Low-rate and first set

After 100 iterations, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 32.95ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

BU-CPU-LowFirst with Low-rate and first set

The following figure 5.24 represents the results of the simulation performed by the Low-rate player in the BU-CPU-LowFirst strategy. The BU-CPU-LowFirst policy policy means that users access the cloud game engine in order of smallest to largest CPU usage.

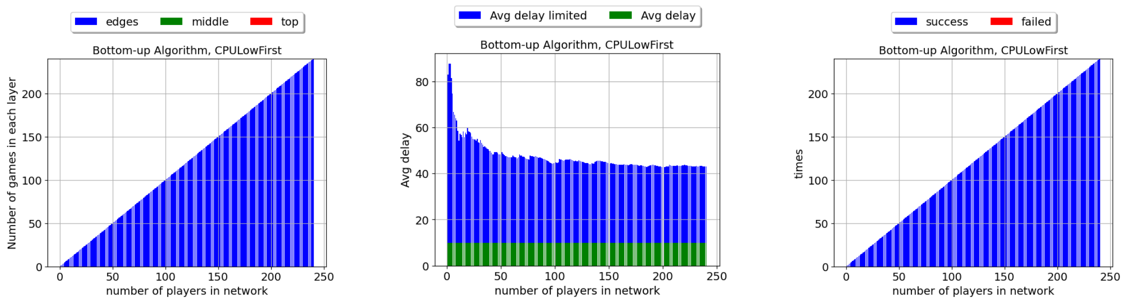


Figure 5.24: BU-CPU-LowFirst with Low-rate and first set

After 100 iterations of the experiment, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 32.95ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

BU-Delay-HighFirst with Low-rate and first set

The following figure 5.25 represents the results of the simulation performed by the Low-rate player in the BU-Delay-HighFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of highest to lowest user tolerable delay.

After 100 iterations of the experiment, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 32.95ms, and there are

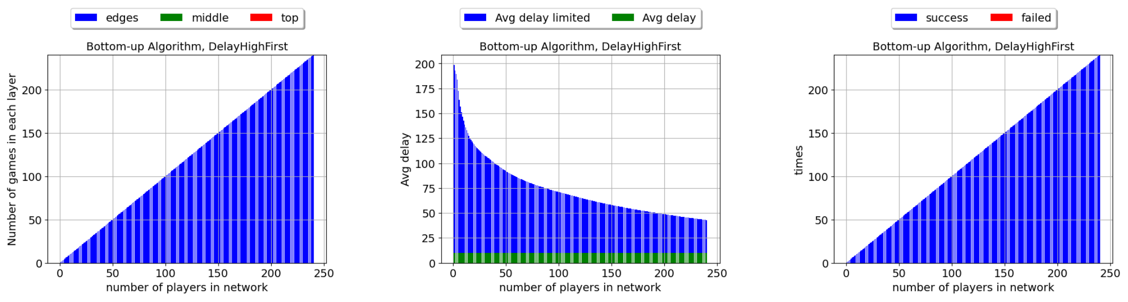


Figure 5.25: BU-Delay-HighFirst with Low-rate and first set

no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

BU-Delay-LowFirst with Low-rate and first set

The following figure 5.26 represents the results of the simulation performed by the Low-rate player in the BU-Delay-LowFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of lowest to highest user tolerable delay.

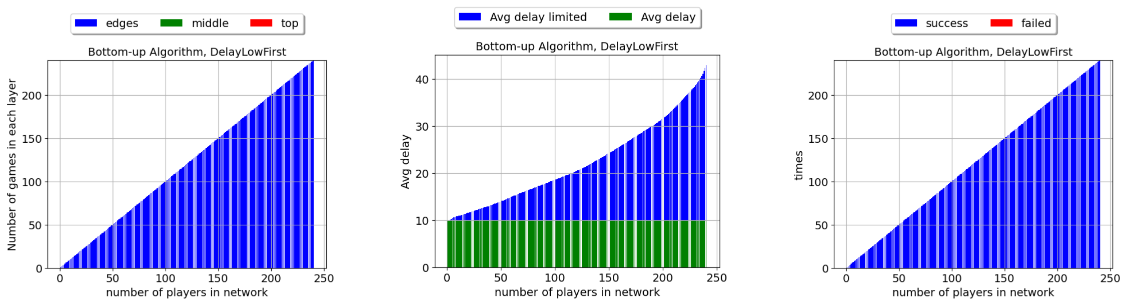


Figure 5.26: BU-Delay-LowFirst with Low-rate and first set

After 100 iterations, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 32.95ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Random with Low-rate and first set

The following Figure 5.27 below shows the results of a simulation performed by a Low-rate player under the TD-Random strategy, which means that the cloud game engine is accessed in a random order.

After 100 iterations, an average of 240 users were allowed to access the service node each time, with an average delay of 22.99ms and an average delay Slack of 19.99ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

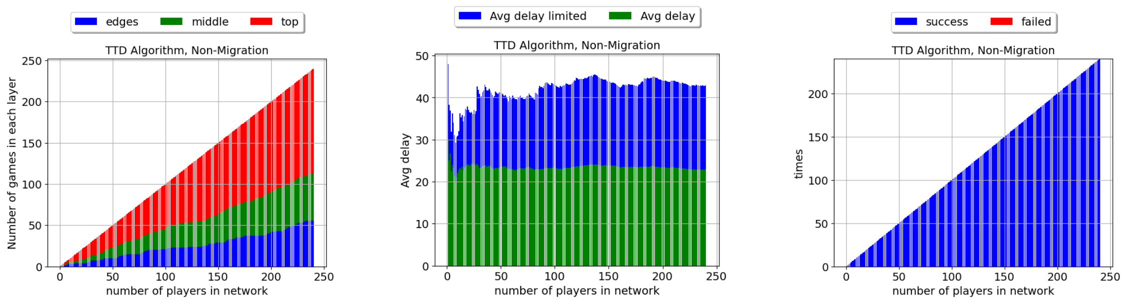


Figure 5.27: TD-Random with Low-rate and first set

TD-CPU-HighFirst with Low-rate and first set

The following Figure 5.28 below shows the simulation results for Low-rate players under the TD-CPU-HighFirst policy, which means that the Cloud Game Engine is accessed from the highest to the lowest user CPU usage.

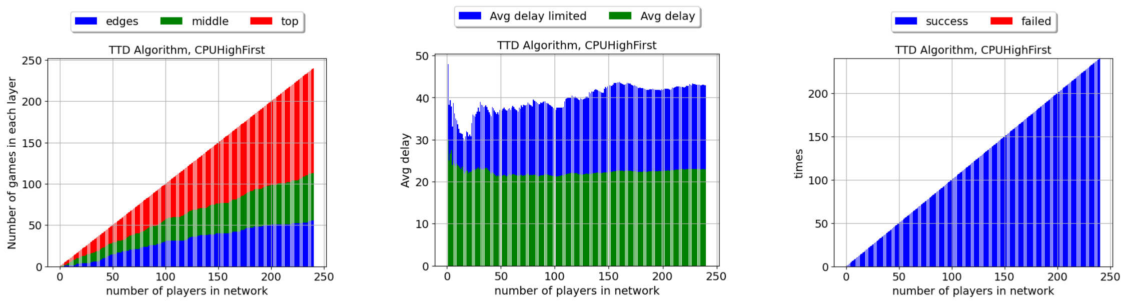


Figure 5.28: TD-CPU-HighFirst with Low-rate and first set

Very similar to TD-Random, after 100 iterations of the experiment, an average of 240 users were allowed to access the service node each time, with an average delay of 22.99ms and an average delay Slack of 19.99ms. There were no users who could not access the service, so the failure rate was ZERO. This is summarised in a table later.

TD-CPU-LowFirst with Low-rate and first set

The following Figure 5.29 shows the simulation results for Low-rate players under the TD-CPU-LowFirst policy. TD-CPU-LowFirst policy means that the Cloud Game Engine is accessed from the lowest to the highest user CPU usage.

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.99ms, the average delay Slack is 19.99ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

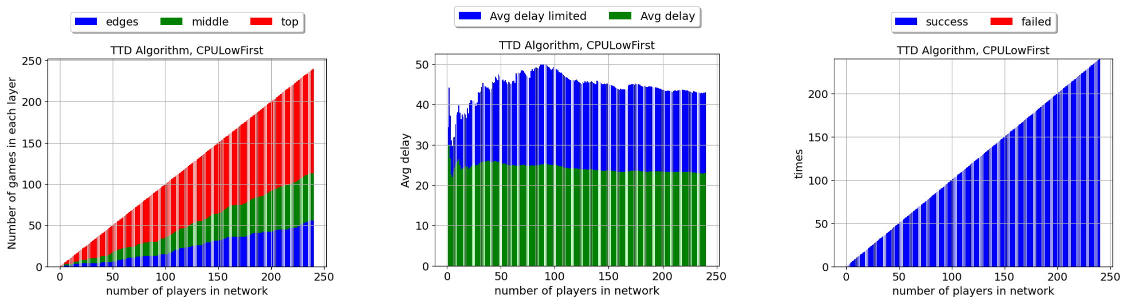


Figure 5.29: TD-CPU-LowFirst with Low-rate and first set

TD-Delay-HighFirst with Low-rate and first set

The following Figure 5.30 represents the simulation results for Low-rate players under the TD-Delay-HighFirst policy, which means that the Cloud Game Engine is accessed in descending order of the maximum delay tolerated by the user.

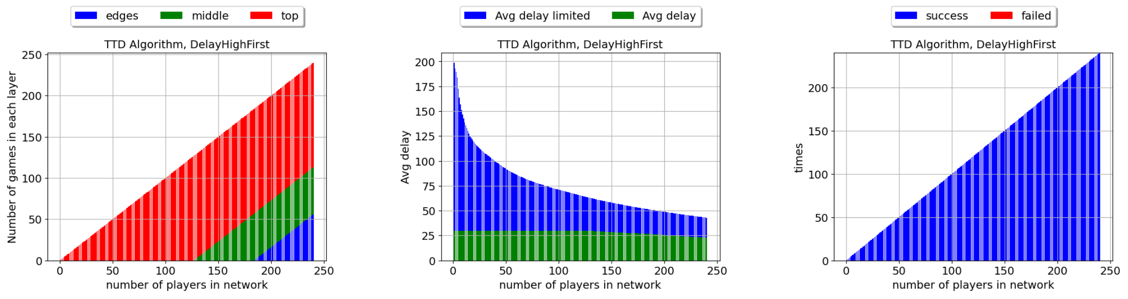


Figure 5.30: TD-Delay-HighFirst with Low-rate and first set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.99ms, the average delay Slack is 19.99ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-LowFirst with Low-rate and first set

The following Figure 5.31 shows the simulation results for Low-rate players under the TD-Delay-LowFirst policy, which means that the Cloud Game Engine is accessed in descending order of the highest delay tolerated by the user.

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.99ms, the average delay Slack is 19.99ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

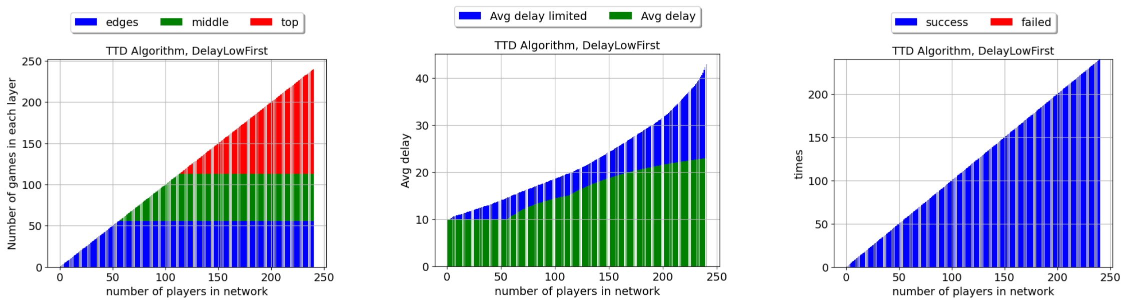


Figure 5.31: TD-Delay-LowFirst with Low-rate and first set

5.3 Experiments with second setting

The following Figure 5.32 shows the second set of data, it decrease to 200GHz total CPU resources per EdgeNode for L0 and 290GHZ total resources per Node for L1. L2 has theoretically unlimited resources, so it is set to 10000000GHz.

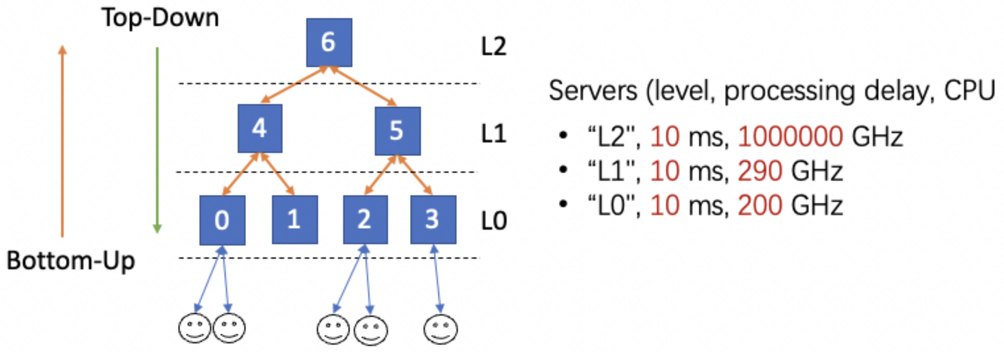


Figure 5.32: Second Simulation Structure

In the first experiment, there are two sections, Bottom-Up and Top-Down, depending on the algorithms, and each section is divided into Random, High-CPU first, Low-CPU first, High-Delay first, and Low-Delay first, depending on the allocation strategy of the species.

5.3.1 Simulation result with high-rate user

In the same experimental environment, the AvgDelay for users in the traditional cloud game engine is 30ms and AvgDelaySlack is 10.13ms, which can serve all users under ideal conditions, but requires much higher hardware resources than the distributed cloud game engine, and users who need low delay to have a good QoE game will have a very poor gaming experience.

BU-Random with High-rate and second set

The following figure 5.33 represents the results of the simulation performed by the high-rate player in the BU-Random strategy. The BU-Random policy means that users access the cloud game engine in random order.

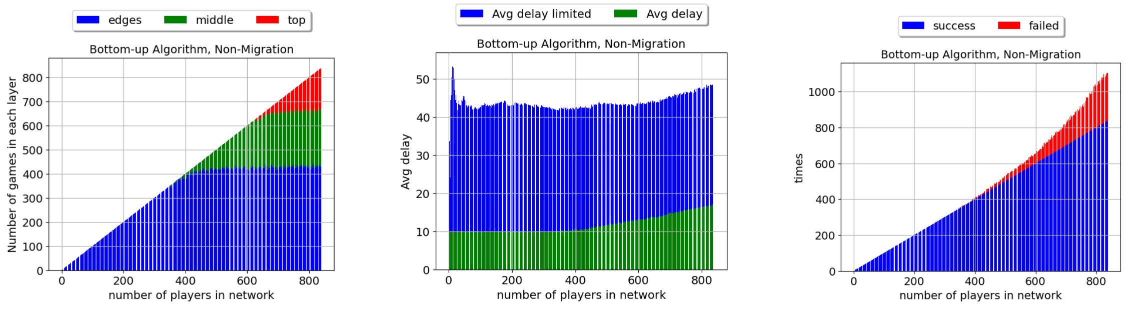


Figure 5.33: BU-Random with High-rate and second set

After 100 iterations, an average of 848 users were allowed to access the service node each time, the average delay of users was 17.68ms, the average delay of Slack was 30.9ms, and an average of 314 users were unable to access the service, with a failure rate of approximately 27 percent.These are summarised in a table later.

BU-CPU-HighFirst with High-rate and second set

The following figure 5.34 represents the results of the simulation performed by the high-rate player in the BU-CPU-HighFirst strategy.The BU-CPU-HighFirst policy means that users access the cloud game engine in order of largest to smallest CPU usage.

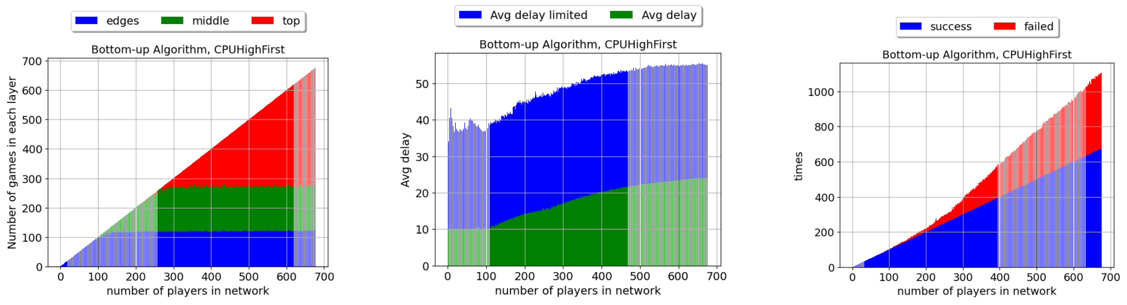


Figure 5.34: BU-CPU-HighFirst with High-rate and second set

After 100 iterations, an average of 725 users were allowed to access the service node each time, the average delay of users was 24.33ms, the average delay of Slack was 29.56ms, and an average of 446 users were unable to access the service, a failure rate of approximately 38 percent. These are summarised in a table later.

BU-CPU-LowFirst with High-rate and second set

The following figure 5.35 represents the results of the simulation performed by the high-rate player in the BU-CPU-LowFirst strategy. The BU-CPU-LowFirst policy policy means that users access the cloud game engine in order of smallest to largest CPU usage.

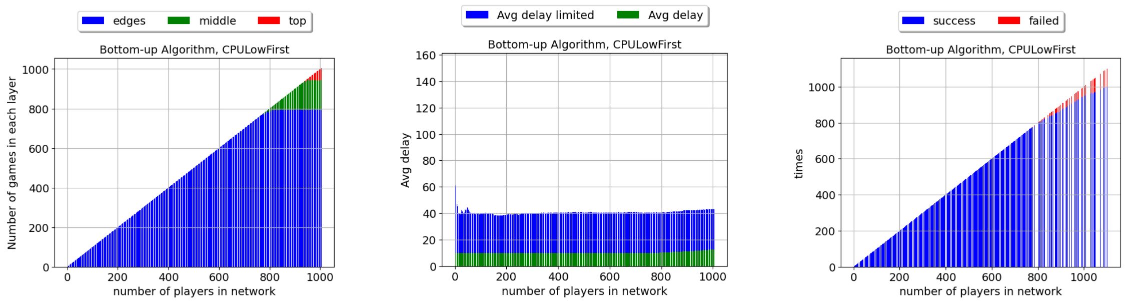


Figure 5.35: BU-CPU-LowFirst with High-rate and second set

After 100 iterations of the experiment, an average of 1032 users were allowed to access the service node each time, the average delay of the users was 13.05ms, the average delay Slack was 30.26ms, and an average of 139 users were unable to access the service, with a failure rate of approximately 12 percent. These are summarised in a table later.

BU-Delay-HighFirst with High-rate and second set

The following figure 5.36 represents the results of the simulation performed by the high-rate player in the BU-Delay-HighFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of highest to lowest user tolerable delay.

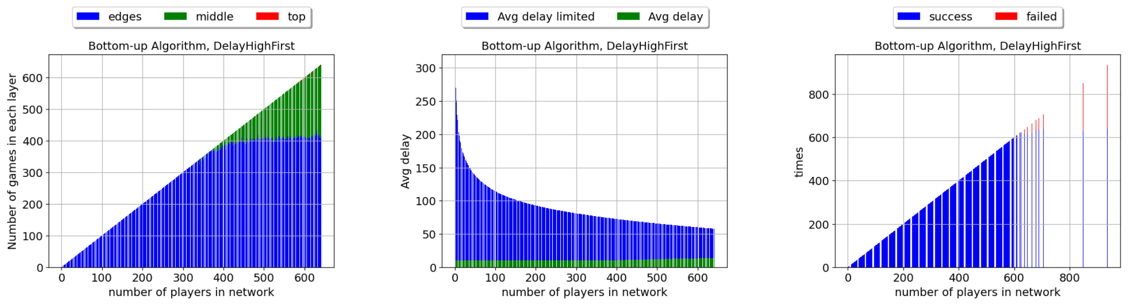


Figure 5.36: BU-Delay-HighFirst with High-rate and second set

After 100 iterations of the experiment, an average of 678 users were allowed to access the service node each time, with an average delay of 14.17ms, an average delay Slack of 42.65ms, and an average of 493 users unable to access the service, for a failure rate of roughly 42 percent. These are summarised in a table later.

BU-Delay-LowFirst with High-rate and second set

The following Figure 5.37 represents the results of the simulation performed by the high-rate player in the BU-Delay-LowFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of lowest to highest user tolerable delay.

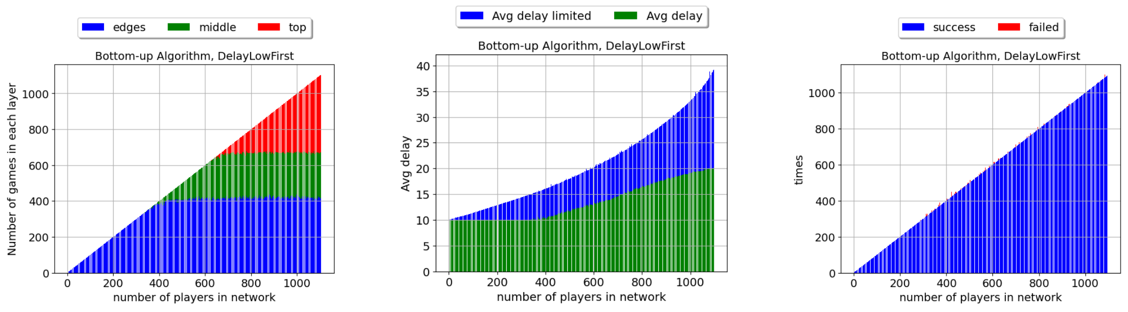


Figure 5.37: BU-Delay-LowFirst with High-rate and second set

After 100 iterations, an average of 1165 users were allowed to access the service node each time, with an average delay of 21.05ms and an average delay Slack of 19.18ms. There were 6 unable to access the service, so the failure rate was lower than 1 percent. These are summarised in a table later.

TD-Random with High-rate and second set

The following Figure 5.38 below shows the results of a simulation performed by a high-rate player under the TD-Random strategy, which means that the cloud game engine is accessed in a random order.

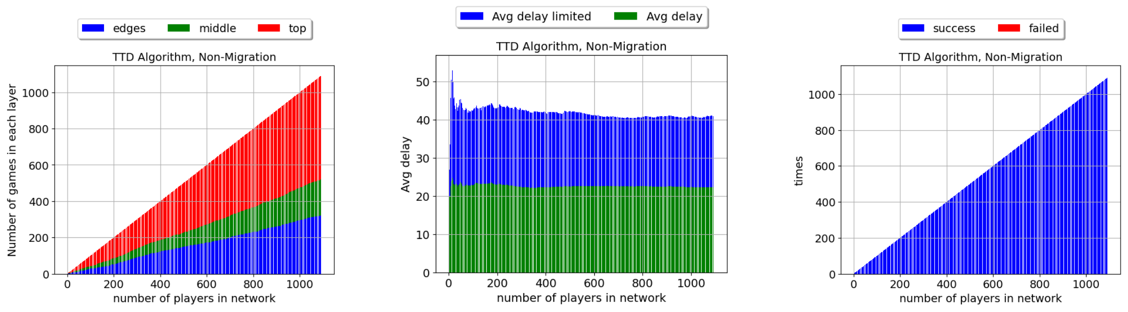


Figure 5.38: TD-Random with High-rate and second set

After 100 iterations, an average of 1150 users were allowed to access the service node each time, with an average delay of 22.16ms and an average delay Slack of 18.23ms. There were 12 unable to access the service, so the failure rate was 1 percent. These are summarised in a table later.

TD-CPU-HighFirst with High-rate and second set

The following Figure 5.39 below shows the simulation results for high-rate players under the TD-CPU-HighFirst policy, which means that the Cloud Game Engine is accessed from the highest to the lowest user CPU usage.

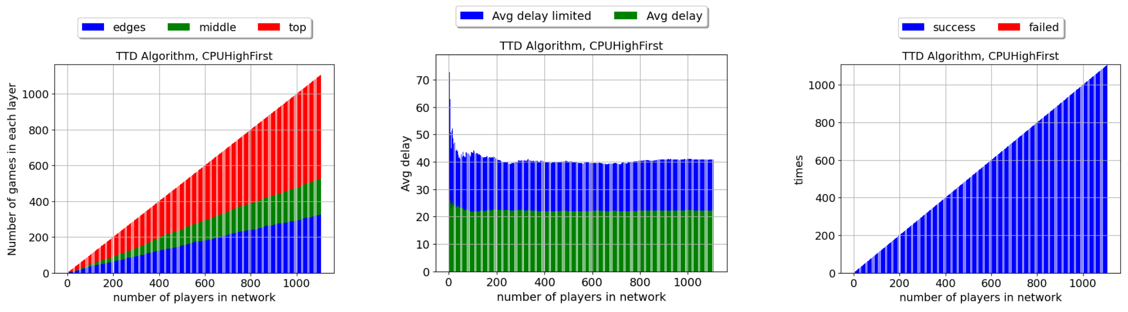


Figure 5.39: TD-CPU-HighFirst with High-rate and second set

Very similar to TD-Random, after 100 iterations of the experiment, an average of 1171 users were allowed to access the service node each time, with an average delay of 22.03ms and an average delay Slack of 18.10ms. There were no users who could not access the service, so the failure rate was ZERO. This is summarised in a table later.

TD-CPU-LowFirst with High-rate and second set

The following Figure 5.40 shows the simulation results for high-rate players under the TD-CPU-LowFirst policy. tTD-CPU-LowFirst policy means that the Cloud Game Engine is accessed from the lowest to the highest user CPU usage.

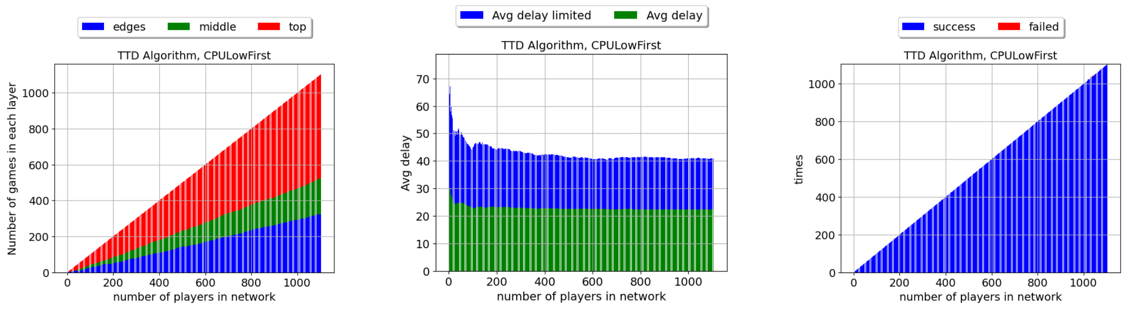


Figure 5.40: TD-CPU-LowFirst with High-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.03ms, the average delay Slack is 18.10ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-HighFirst with High-rate and second set

The following Figure 5.41 represents the simulation results for high-rate players under the TD-Delay-HighFirst policy, which means that the Cloud Game Engine is accessed in descending order of the maximum delay tolerated by the user.

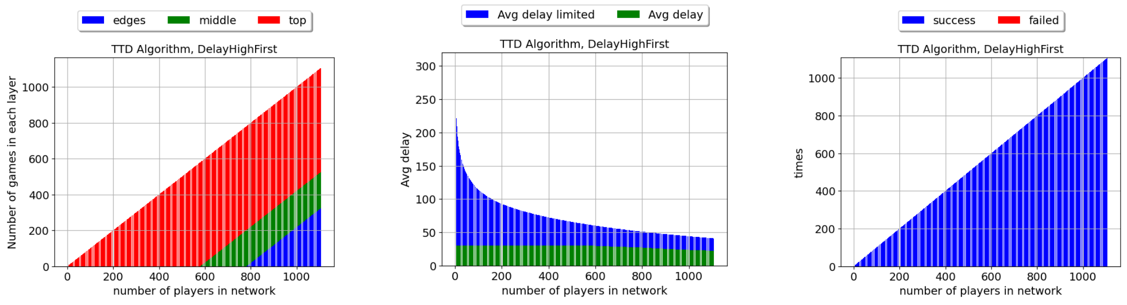


Figure 5.41: TD-Delay-HighFirst with High-rate and second set

The simulation results for this policy are still very similar to other TDs, with all users allowed to access the service node, the average delay of users is 22.03ms, the average delay Slack is 18.10ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-LowFirst with High-rate and second set

The following Figure 5.42 shows the simulation results for high-rate players under the TD-Delay-LowFirst policy, which means that the Cloud Game Engine is accessed in descending order of the highest delay tolerated by the user.

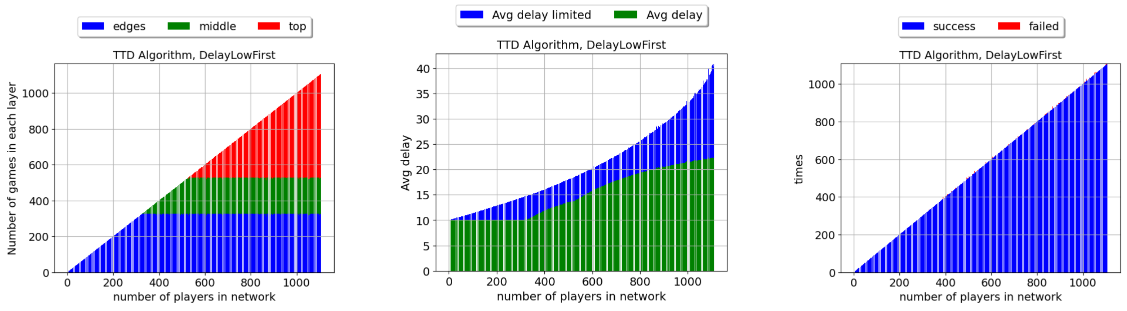


Figure 5.42: TD-Delay-LowFirst with High-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.03ms, the average delay Slack is 18.10ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

5.3.2 Simulation result with mid-rate user

In the same experimental environment, the AvgDelay for users in the traditional cloud game engine is 30ms and AvgDelaySlack is 7.38ms, which can serve all users under ideal conditions, but requires much higher hardware resources than the distributed cloud game engine, and users who need low delay to have a good QoE game will have a very poor gaming experience.

BU-Random with Mid-rate and second set

The following figure 5.43 represents the results of the simulation performed by the mid-rate player in the BU-Random strategy. The BU-Random policy means that users access the cloud game engine in random order.

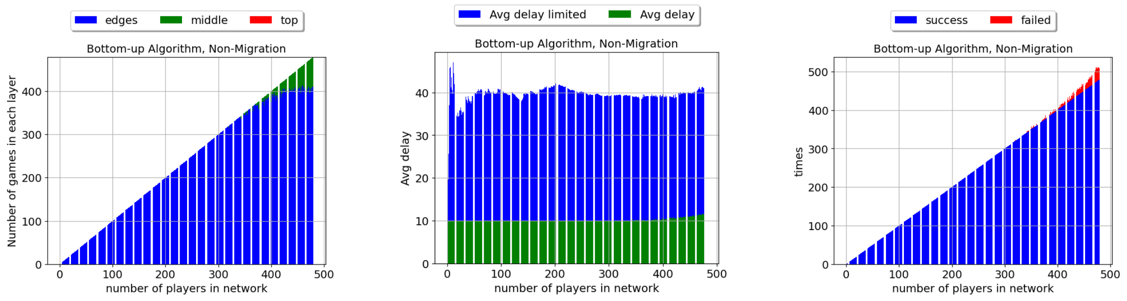


Figure 5.43: BU-Random with Mid-rate and second set

After 100 iterations, an average of 481 users were allowed to access the service node each time, the average delay of users was 11.71ms, the average delay of Slack was 27.38ms, and an average of 34 users were unable to access the service, with a failure rate of approximately 7 percent.These are summarised in a table later.

BU-CPU-HighFirst with Mid-rate and second set

The following figure 5.44 represents the results of the simulation performed by the mid-rate player in the BU-CPU-HighFirst strategy.The BU-CPU-HighFirst policy means that users access the cloud game engine in order of largest to smallest CPU usage.

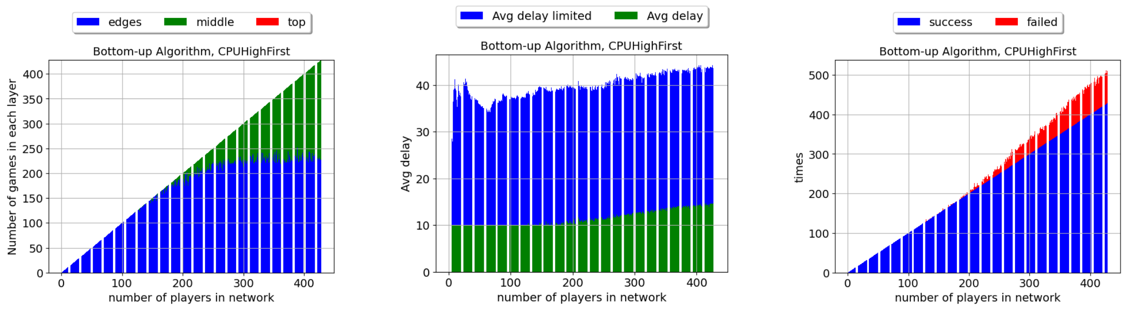


Figure 5.44: BU-CPU-HighFirst with Mid-rate and second set

After 100 iterations, an average of 437 users were allowed to access the service node each time, the average delay of users was 14.58ms, the average delay of Slack was 27.2ms, and an average of 85 users were unable to access the service, a failure rate of approximately 16 percent. These are summarised in a table later.

BU-CPU-LowFirst with Mid-rate and second set

The following figure 5.45 represents the results of the simulation performed by the mid-rate player in the BU-CPU-LowFirst strategy. The BU-CPU-LowFirst policy policy means that users access the cloud game engine in order of smallest to largest CPU usage.

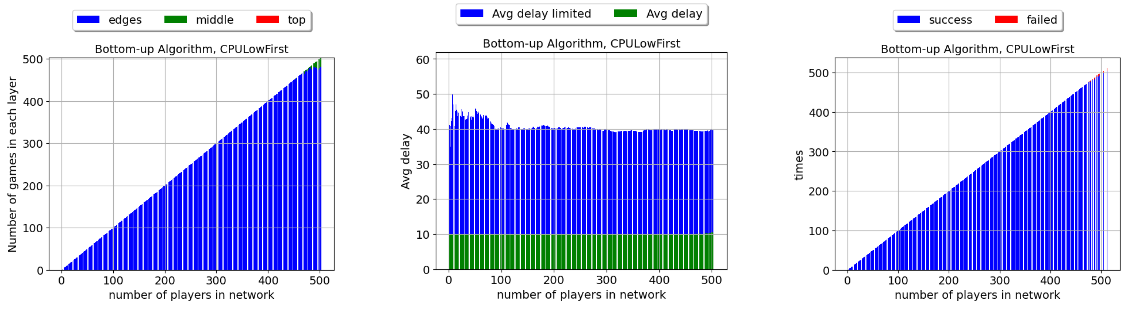


Figure 5.45: BU-CPU-LowFirst with Mid-rate and second set

After 100 iterations of the experiment, an average of 503 users were allowed to access the service node each time, the average delay of the users was 10.5ms, the average delay Slack was 27.75ms, and an average of 19 users were unable to access the service, with a failure rate of approximately 4 percent. These are summarised in a table later.

BU-Delay-HighFirst with Mid-rate and second set

The following figure 5.46 represents the results of the simulation performed by the mid-rate player in the BU-Delay-HighFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of highest to lowest user tolerable delay.

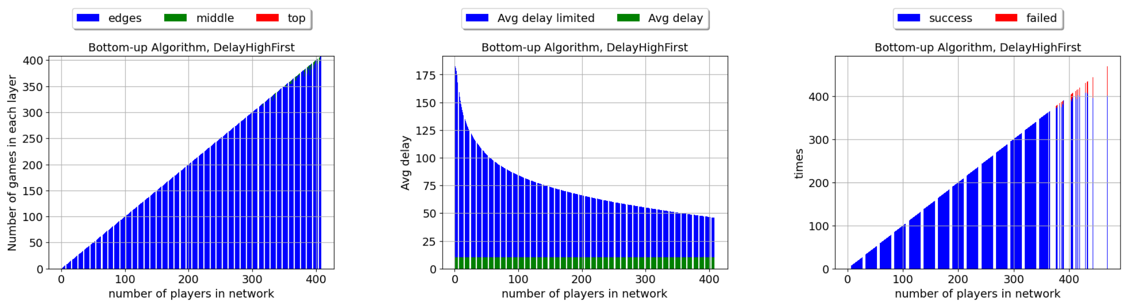


Figure 5.46: BU-Delay-HighFirst with Mid-rate and second set

After 100 iterations of the experiment, an average of 431 users were allowed to access the service node each time, with an average delay of 10.14ms, an average delay Slack of 32.2ms, and an average of 91 users unable to access the service, for a failure rate of roughly 17 percent. These are summarised in a table later.

BU-Delay-LowFirst with Mid-rate and second set

The following figure 5.47 represents the results of the simulation performed by the mid-rate player in the BU-Delay-LowFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of lowest to highest user tolerable delay.

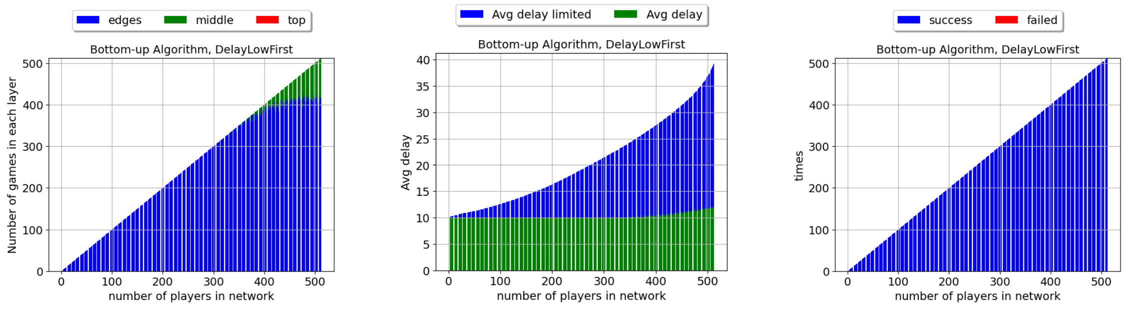


Figure 5.47: BU-Delay-LowFirst with Mid-rate and second set

After 100 iterations, an average of 522 users were allowed to access the service node each time, with an average delay of 12.24ms and an average delay Slack of 25.14ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

TD-Random with Mid-rate and second set

The following Figure 5.48 below shows the results of a simulation performed by a mid-rate player under the TD-Random strategy, which means that the cloud game engine is accessed in a random order.

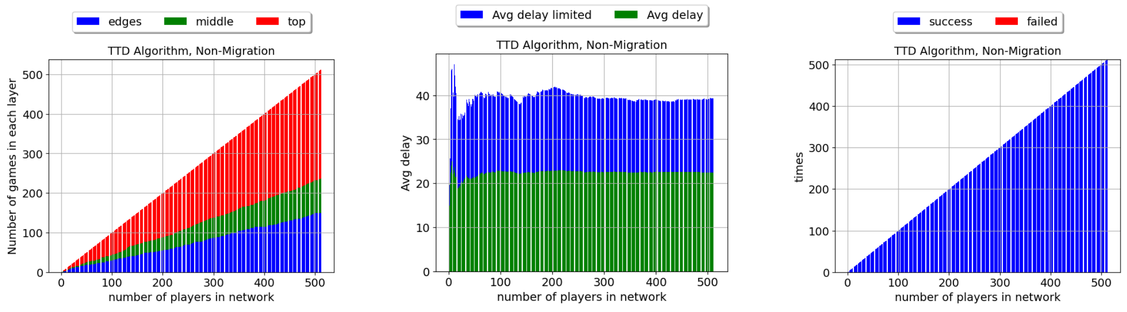


Figure 5.48: TD-Random with Mid-rate and second set

After 100 iterations, an average of 522 users were allowed to access the service node each time, with an average delay of 21.95ms and an average delay Slack of 15.43ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

TD-CPU-HighFirst with Mid-rate and second set

The following Figure 5.49 below shows the simulation results for mid-rate players under the TD-CPU-HighFirst policy, which means that the Cloud Game Engine is accessed from the highest to the lowest user CPU usage.

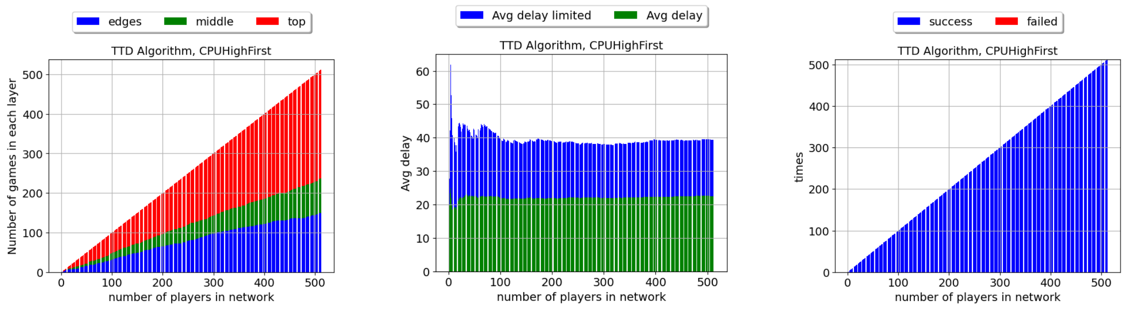


Figure 5.49: TD-CPU-HighFirst with Mid-rate and second set

Very similar to TD-Random, after 100 iterations of the experiment, an average of 522 users were allowed to access the service node each time, with an average delay of 21.95ms and an average delay Slack of 15.43ms. There were no users who could not access the service, so the failure rate was ZERO. This is summarised in a table later.

TD-CPU-LowFirst with Mid-rate and second set

The following Figure 5.50 shows the simulation results for mid-rate players under the TD-CPU-LowFirst policy. tTD-CPU-LowFirst policy means that the Cloud Game Engine is accessed from the lowest to the highest user CPU usage.

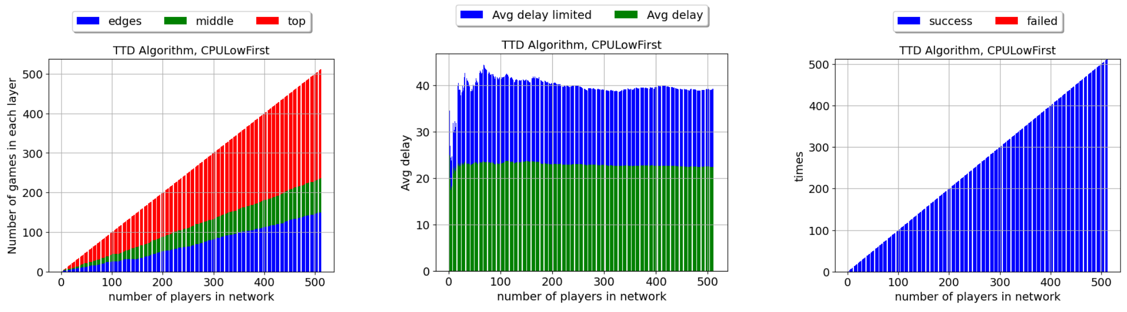


Figure 5.50: TD-CPU-LowFirst with Mid-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.95ms, the average delay Slack is 15.43ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-HighFirst with Mid-rate and second set

The following Figure 5.51 represents the simulation results for mid-rate players under the TD-Delay-HighFirst policy, which means that the Cloud Game Engine is accessed in descending order of the maximum delay tolerated by the user.

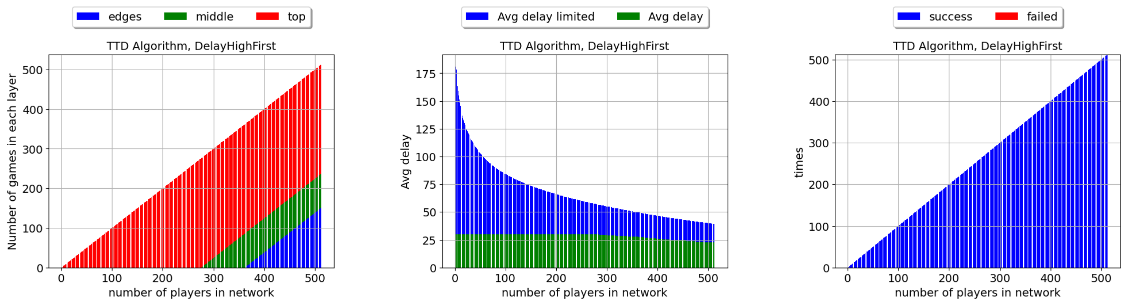


Figure 5.51: TD-Delay-HighFirst with Mid-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.95ms, the average delay Slack is 15.43ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-LowFirst with Mid-rate and second set

The following Figure 5.52 shows the simulation results for mid-rate players under the TD-Delay-LowFirst policy, which means that the Cloud Game Engine is accessed in descending order of the highest delay tolerated by the user.

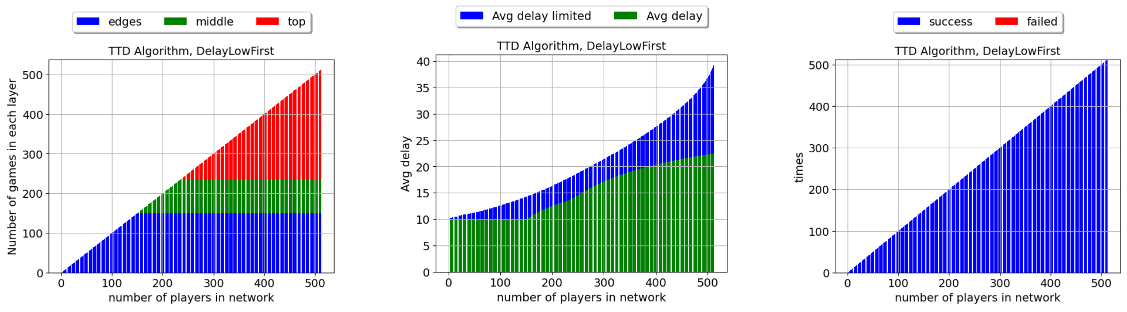


Figure 5.52: TD-Delay-LowFirst with Mid-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 21.95ms, the average delay Slack is 15.43ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

5.3.3 Simulation result with low-rate user

In the same experimental environment, the AvgDelay for users in the traditional cloud game engine is 30ms and AvgDelaySlack is 9.48ms, which can serve all users under ideal conditions, but requires much higher hardware resources than the distributed cloud game engine, and users who need low delay to have a good QoE game will have a very poor gaming experience.

BU-Random with Low-rate and second set

The following figure 5.53 represents the results of the simulation performed by the Low-rate player in the BU-Random strategy. The BU-Random policy means that users access the cloud game engine in random order.

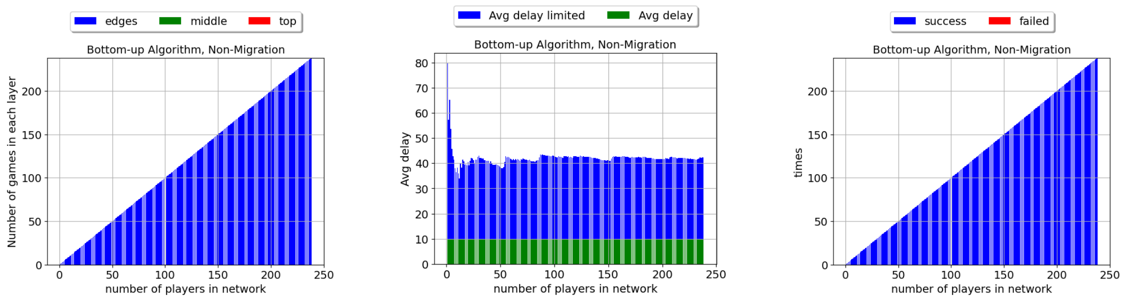


Figure 5.53: BU-Random with Low-rate and second set

After 100 iterations, an average of 254 users were allowed to access the service node each time, the average delay of users was 10ms, the average delay of Slack was 29.48ms, and have no users were unable to access the service, with failure rate is zero.These are summarised in a table later.

BU-CPU-HighFirst with Low-rate and second set

The following figure 5.54 represents the results of the simulation performed by the Low-rate player in the BU-CPU-HighFirst strategy.The BU-CPU-HighFirst policy means that users access the cloud game engine in order of largest to smallest CPU usage.

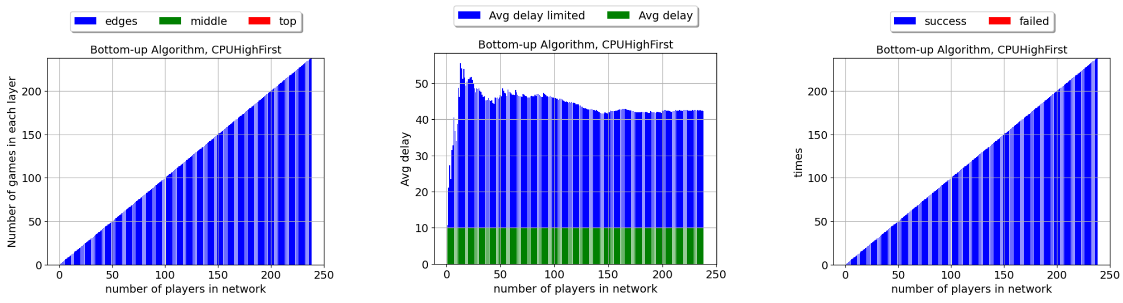


Figure 5.54: BU-CPU-HighFirst with Low-rate and second set

After 100 iterations, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 29.48ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

BU-CPU-LowFirst with Low-rate and second set

The following figure 5.55 represents the results of the simulation performed by the Low-rate player in the BU-CPU-LowFirst strategy. The BU-CPU-LowFirst policy policy means that users access the cloud game engine in order of smallest to largest CPU usage.

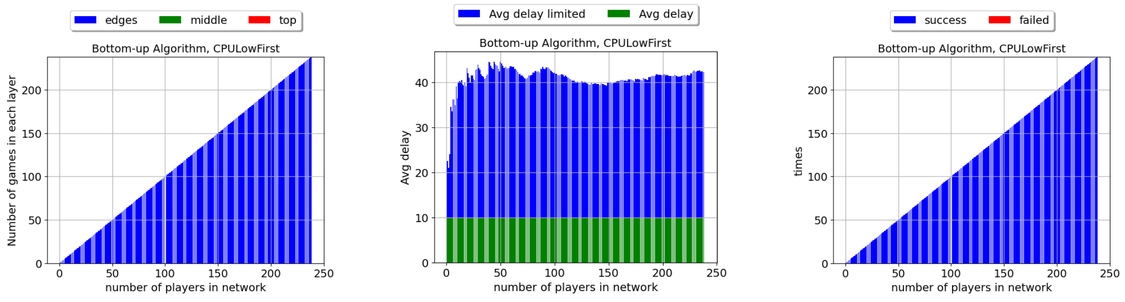


Figure 5.55: BU-CPU-LowFirst with Low-rate and second set

After 100 iterations of the experiment, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 29.48ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

BU-Delay-HighFirst with Low-rate and second set

The following figure 5.56 represents the results of the simulation performed by the Low-rate player in the BU-Delay-HighFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of highest to lowest user tolerable delay.

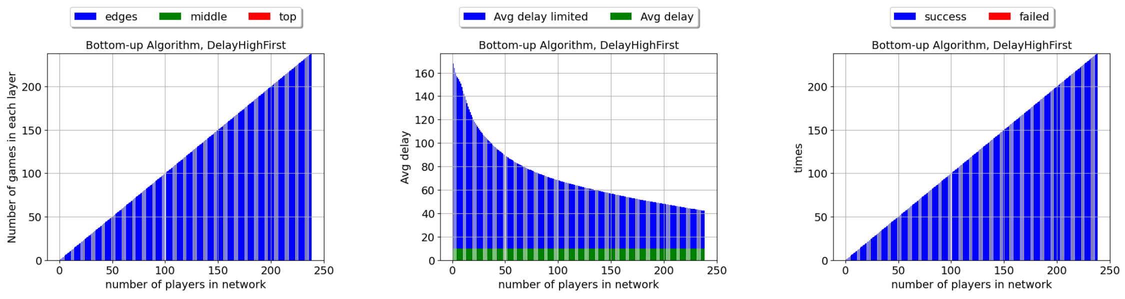


Figure 5.56: BU-Delay-HighFirst with Low-rate and second set

After 100 iterations of the experiment, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 29.48ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

BU-Delay-LowFirst with Low-rate and second set

The following figure 5.57 represents the results of the simulation performed by the Low-rate player in the BU-Delay-LowFirst strategy. The BU-Delay-HighFirst policy means that the cloud game engine is accessed in order of lowest to highest user tolerable delay.

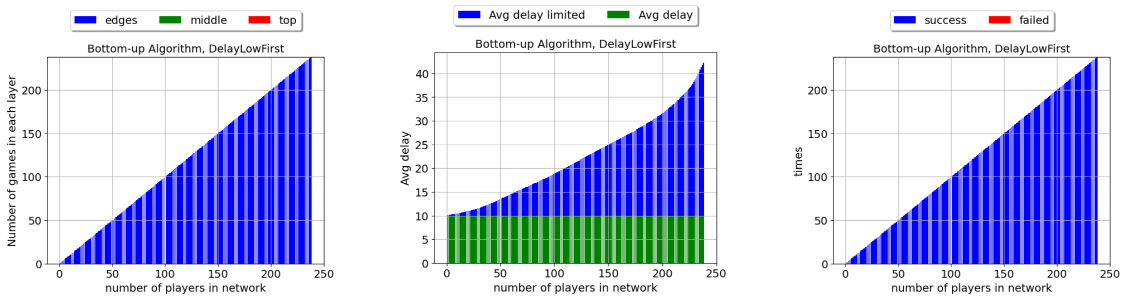


Figure 5.57: BU-Delay-LowFirst with Low-rate and second set

After 100 iterations, with all users allowed to access the service node, the average delay of users is 10ms, the average delay Slack is 29.48ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Random with Low-rate and second set

The following Figure 5.58 below shows the results of a simulation performed by a Low-rate player under the TD-Random strategy, which means that the cloud game engine is accessed in a random order.

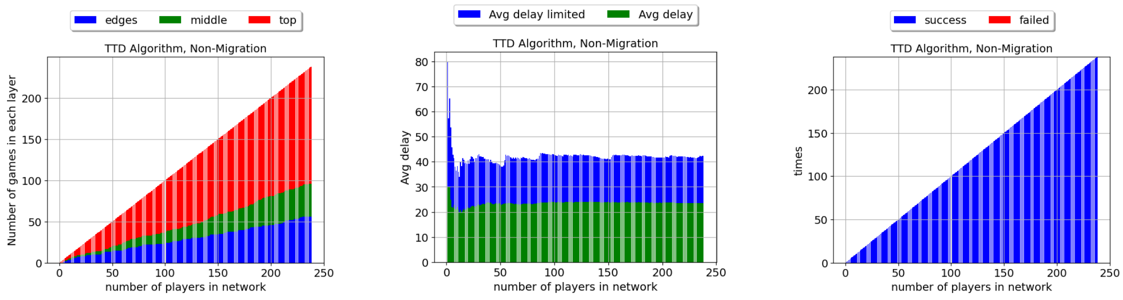


Figure 5.58: TD-Random with Low-rate and second set

After 100 iterations, an average of 254 users were allowed to access the service node each time, with an average delay of 22.01ms and an average delay Slack of 17.47ms. There were no users unable to access the service, so the failure rate was ZERO. These are summarised in a table later.

TD-CPU-HighFirst with Low-rate and second set

The following Figure 5.59 below shows the simulation results for Low-rate players under the TD-CPU-HighFirst policy, which means that the Cloud Game Engine is accessed from the highest to the lowest user CPU usage.

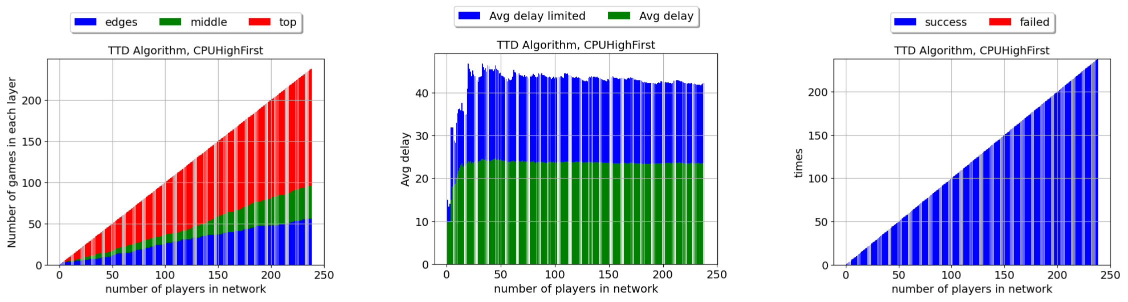


Figure 5.59: TD-CPU-HighFirst with Low-rate and second set

Very similar to TD-Random, after 100 iterations of the experiment, an average of 254 users were allowed to access the service node each time, with an average delay of 22.01ms and an average delay Slack of 17.47ms. There were no users who could not access the service, so the failure rate was ZERO. This is summarised in a table later.

TD-CPU-LowFirst with Low-rate and second set

The following Figure 5.60 shows the simulation results for Low-rate players under the TD-CPU-LowFirst policy. TD-CPU-LowFirst policy means that the Cloud Game Engine is accessed from the lowest to the highest user CPU usage.

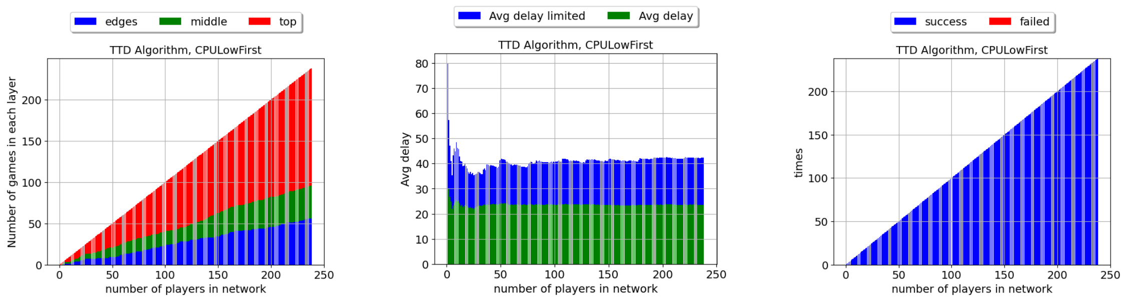


Figure 5.60: TD-CPU-LowFirst with Low-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.01ms, the average delay Slack is 17.47ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-HighFirst with Low-rate and second set

The following Figure 5.61 represents the simulation results for Low-rate players under the TD-Delay-HighFirst policy, which means that the Cloud Game Engine is accessed in descending order of the maximum delay tolerated by the user.

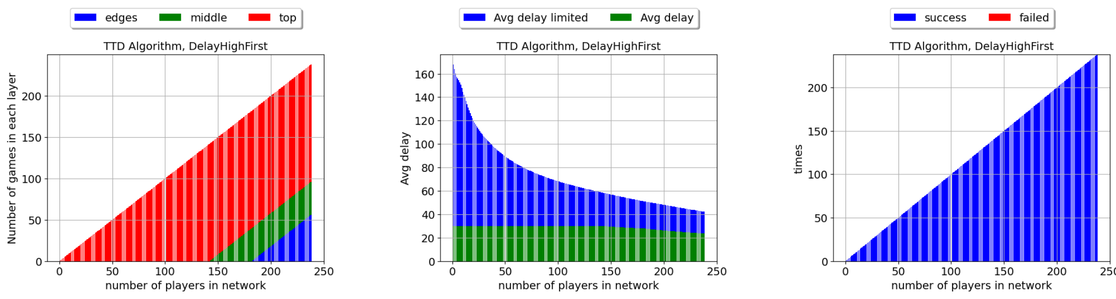


Figure 5.61: TD-Delay-HighFirst with Low-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.01ms, the average delay Slack is 17.47ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

TD-Delay-LowFirst with Low-rate and second set

The following Figure 5.62 shows the simulation results for Low-rate players under the TD-Delay-LowFirst policy, which means that the Cloud Game Engine is accessed in descending order of the highest delay tolerated by the user.

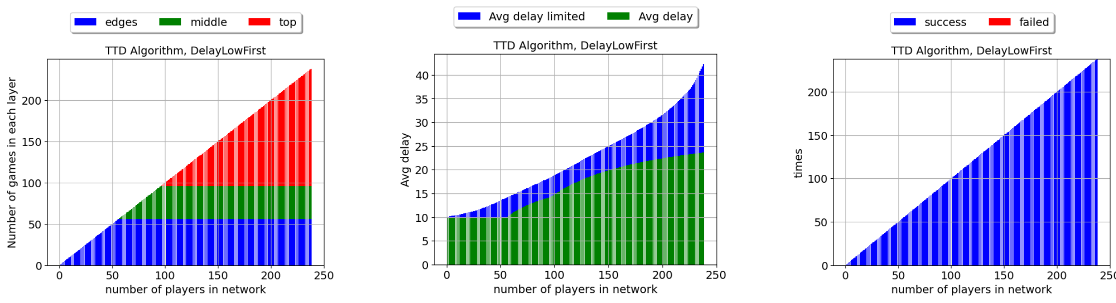


Figure 5.62: TD-Delay-LowFirst with Low-rate and second set

The simulation results for this policy are still very similar to TD-Random, with all users allowed to access the service node, the average delay of users is 22.01ms, the average delay Slack is 17.47ms, and there are no users unable to access the service, so the failure rate is zero. These are summarised in a table later.

5.4 Analyse result of simulation

The results shown in the previous section were obtained from the first and second set of experimental setup simulations, this section groups all the results and analyses them in terms of system load size. The first experiment is to simulate the operation of the distributed engine with sufficient service resources. The second experiment drastically reduced the amount of resources available to the edge and medium distance service nodes, in order to simulate the operation of the service at a lower capital consumption.

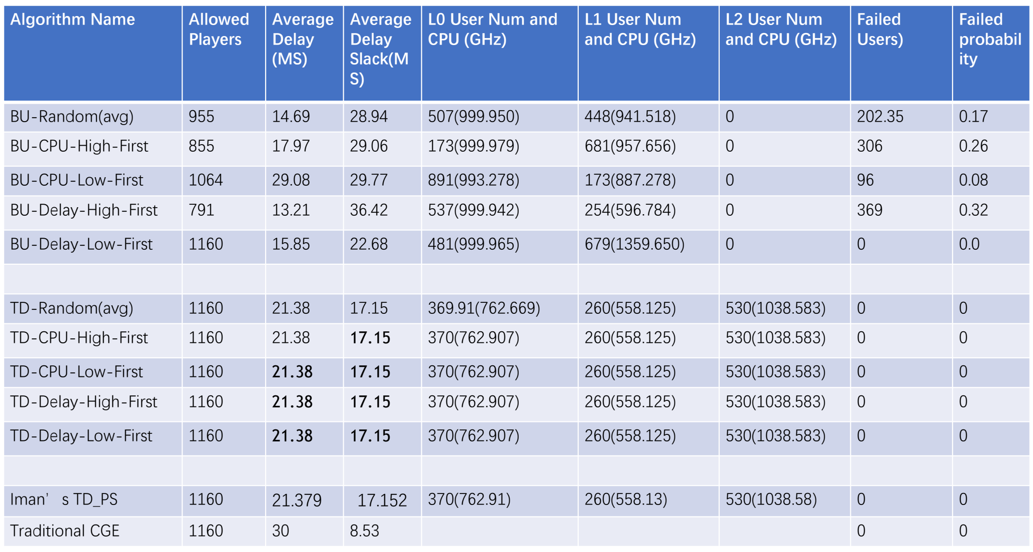
5.4.1 Analyse high-rate user profile

The following Figure 5.63 shows the simulation results for the first set of high-rate user profiles. The high-rate user profile represents a relatively high-load scenario with different allocation policies, and I have added my colleague Iman's best simulation results and data from an ideal traditional game engine for comparison purposes. It can be seen that in the Bottom-Up allocation policy, users are placed in EdgeNodes and low-level CloudNodes as much as possible. However, in order to guarantee high QoE and QoS for each user, the users who enter first have a higher priority, which results in some users who do not need low delay to guarantee QoE taking over the low-delay service nodes, and some users who really need low delay will be crowded out by them, and thus cannot be served by the game engine. This can lead to significant resource wastage and performance degradation. One of the better performing BU policies can be seen in the BU-DelayLow-First method, which preempts users who need low delay to guarantee QoE and thus has the most users in service. All other BU strategies have too many users to serve, so the average QoE is relatively low, but all have a performance advantage over traditional cloud gaming engines.

With the Top-Down strategy, all algorithms have very similar performance and are nearly identical to Iman's TD strategy. It can be seen that the entire adaptive allocation algorithms is very stable under the TD allocation policy, and the adaptive algorithms can consistently serve users regardless of how they enter the system. However, compared to the BU-Delay-Low-First policy, the average user delay is higher and the average delay Slack is lower, which means that the TD policy is less powerful than the former, but still much better than the performance of traditional cloud gaming engines.

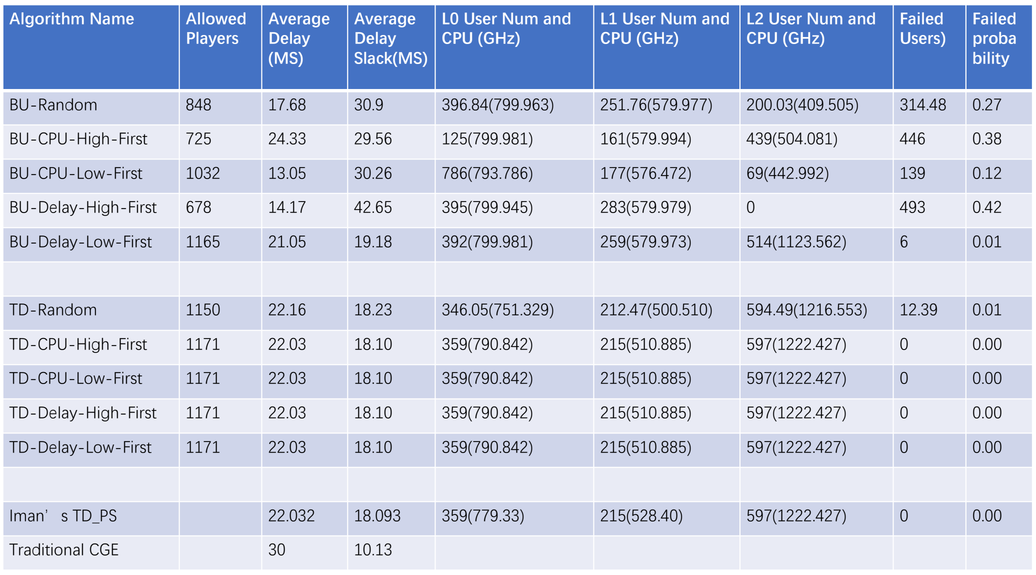
The following Figure shows the simulation results for a second set of high-rate user profiles. In the second set of experiments, the resources of the EdgeNode and Layer1 CloudNodes are drastically reduced to simulate the case where the resources of the service nodes are reduced and the cloud server nodes take on more services. It can be seen that as the resources are reduced, the performance of the BU policy decreases significantly, as does the number of users in the services that can be accommodated. The overall performance of the TD policy does not degrade, but TD-Random has some errors due to the random entry method. The best performing BU policy is still the BU-Delay-Low-First policy, which has a very low probability of user failure. However, there is no significant performance advantage compared to the TD strategy. Compared to traditional cloud game engines, the performance advantage is huge.

The following Figure shows the simulation results for a second set of high-rate user



Algorithm Name	Allowed Players	Average Delay (MS)	Average Delay Slack(MS)	L0 User Num and CPU (GHz)	L1 User Num and CPU (GHz)	L2 User Num and CPU (GHz)	Failed Users)	Failed probability
BU-Random(avg)	955	14.69	28.94	507(999.950)	448(941.518)	0	202.35	0.17
BU-CPU-High-First	855	17.97	29.06	173(999.979)	681(957.656)	0	306	0.26
BU-CPU-Low-First	1064	29.08	29.77	891(993.278)	173(887.278)	0	96	0.08
BU-Delay-High-First	791	13.21	36.42	537(999.942)	254(596.784)	0	369	0.32
BU-Delay-Low-First	1160	15.85	22.68	481(999.965)	679(1359.650)	0	0	0.0
TD-Random(avg)	1160	21.38	17.15	369.91(762.669)	260(558.125)	530(1038.583)	0	0
TD-CPU-High-First	1160	21.38	**17.15**	370(762.907)	260(558.125)	530(1038.583)	0	0
TD-CPU-Low-First	1160	**21.38**	**17.15**	370(762.907)	260(558.125)	530(1038.583)	0	0
TD-Delay-High-First	1160	**21.38**	**17.15**	370(762.907)	260(558.125)	530(1038.583)	0	0
TD-Delay-Low-First	1160	**21.38**	**17.15**	370(762.907)	260(558.125)	530(1038.583)	0	0
Iman' s TD_PS	1160	21.379	17.152	370(762.91)	260(558.13)	530(1038.58)	0	0
Traditional CGE	1160	30	8.53				0	0

Figure 5.63: Comparison with strategy for high-rate users in Simulation 1.



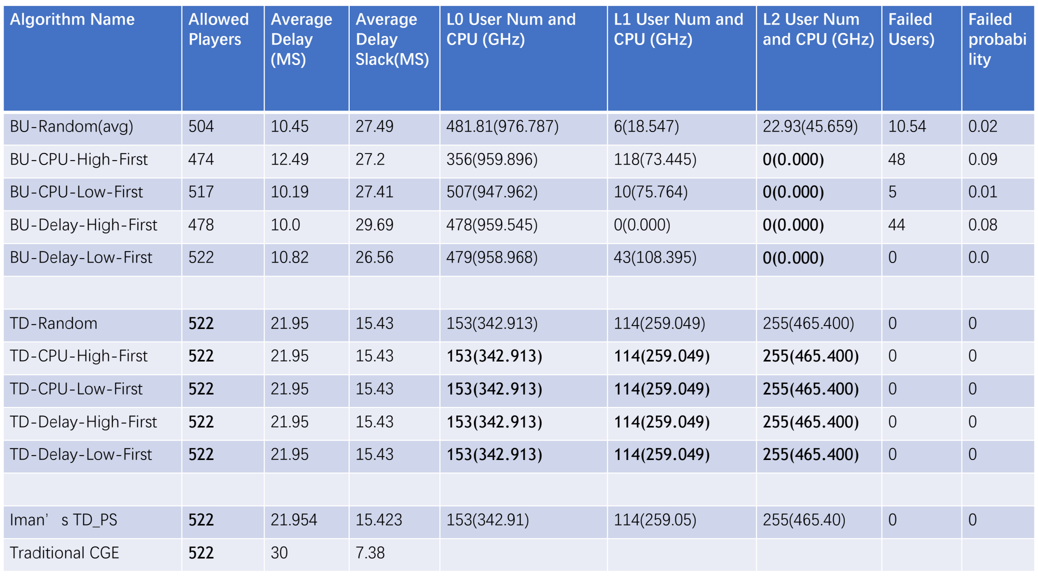
Algorithm Name	Allowed Players	Average Delay (MS)	Average Delay Slack(MS)	L0 User Num and CPU (GHz)	L1 User Num and CPU (GHz)	L2 User Num and CPU (GHz)	Failed Users)	Failed probability
BU-Random	848	17.68	30.9	396.84(799.963)	251.76(579.977)	200.03(409.505)	314.48	0.27
BU-CPU-High-First	725	24.33	29.56	125(799.981)	161(579.994)	439(504.081)	446	0.38
BU-CPU-Low-First	1032	13.05	30.26	786(793.786)	177(576.472)	69(442.992)	139	0.12
BU-Delay-High-First	678	14.17	42.65	395(799.945)	283(579.979)	0	493	0.42
BU-Delay-Low-First	1165	21.05	19.18	392(799.981)	259(579.973)	514(1123.562)	6	0.01
TD-Random	1150	22.16	18.23	346.05(751.329)	212.47(500.510)	594.49(1216.553)	12.39	0.01
TD-CPU-High-First	1171	22.03	18.10	359(790.842)	215(510.885)	597(1222.427)	0	0.00
TD-CPU-Low-First	1171	22.03	18.10	359(790.842)	215(510.885)	597(1222.427)	0	0.00
TD-Delay-High-First	1171	22.03	18.10	359(790.842)	215(510.885)	597(1222.427)	0	0.00
TD-Delay-Low-First	1171	22.03	18.10	359(790.842)	215(510.885)	597(1222.427)	0	0.00
Iman' s TD_PS		22.032	18.093	359(779.33)	215(528.40)	597(1222.427)	0	0.00
Traditional CGE		30	10.13					

Figure 5.64: Comparison with strategy for high-rate users in Simulation 2.

profiles. In the second set of experiments, the resources of the EdgeNode and Layer1 CloudNodes are drastically reduced to simulate the case where the resources of the service nodes are reduced and the cloud server nodes take on more services. It can be seen that as the resources are reduced, the performance of the BU policy decreases significantly, as does the number of users in the services that can be accommodated. The overall performance of the TD policy does not degrade, but TD-Random has some errors due to the random entry method. The best performing BU policy is still the BU-Delay-Low-First policy, which has a very low probability of user failure. However, there is no significant performance advantage compared to the TD strategy. The idealised average delay of 30ms for the traditional cloud game engine under the same experimental conditions is higher than that of the TD and BU strategies, and the average QoS for users is also very low, so the performance advantage of the BU and TD strategies using a combination of edge and cloud computing is very significant compared to the traditional cloud game engine.

5.4.2 Analyse mid-rate user profile

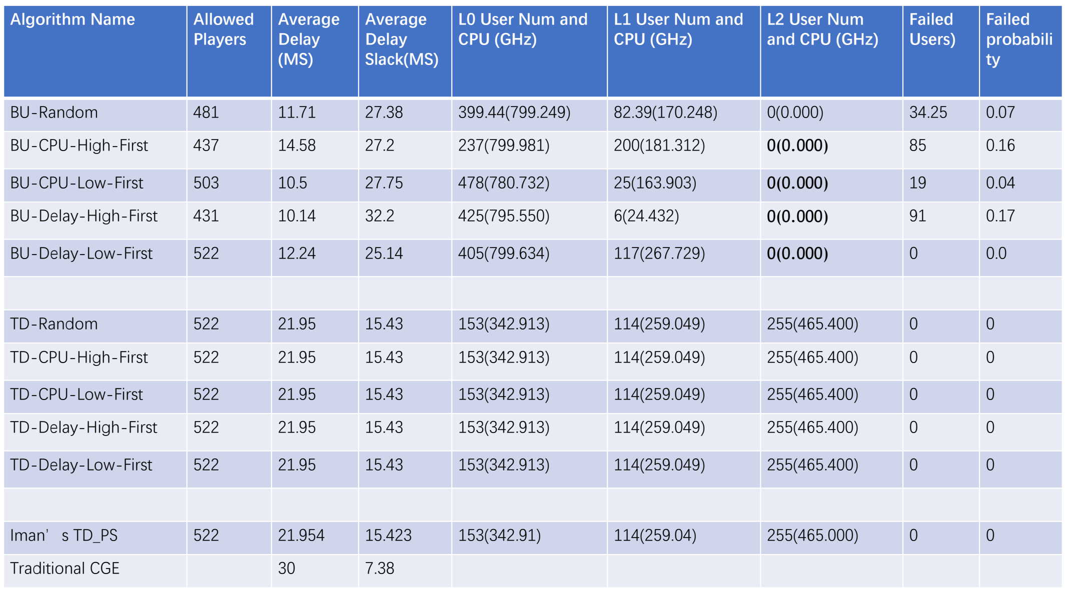
The following Figure 5.65 and Figure 5.66 show the mid-rate user profile, which represents the distributed engine for a medium load scenario.



Algorithm Name	Allowed Players	Average Delay (MS)	Average Delay Slack(MS)	L0 User Num and CPU (GHz)	L1 User Num and CPU (GHz)	L2 User Num and CPU (GHz)	Failed Users)	Failed probability
BU-Random(avg)	504	10.45	27.49	481.81(976.787)	6(18.547)	22.93(45.659)	10.54	0.02
BU-CPU-High-First	474	12.49	27.2	356(959.896)	118(73.445)	**0(0.000)**	48	0.09
BU-CPU-Low-First	517	10.19	27.41	507(947.962)	10(75.764)	**0(0.000)**	5	0.01
BU-Delay-High-First	478	10.0	29.69	478(959.545)	0(0.000)	**0(0.000)**	44	0.08
BU-Delay-Low-First	522	10.82	26.56	479(958.968)	43(108.395)	**0(0.000)**	0	0.0
TD-Random	**522**	21.95	15.43	153(342.913)	114(259.049)	255(465.400)	0	0
TD-CPU-High-First	**522**	21.95	15.43	**153(342.913)**	**114(259.049)**	**255(465.400)**	0	0
TD-CPU-Low-First	**522**	21.95	15.43	**153(342.913)**	**114(259.049)**	**255(465.400)**	0	0
TD-Delay-High-First	**522**	21.95	15.43	**153(342.913)**	**114(259.049)**	**255(465.400)**	0	0
TD-Delay-Low-First	**522**	21.95	15.43	**153(342.913)**	**114(259.049)**	**255(465.400)**	0	0
Iman' s TD_PS	**522**	21.954	15.423	153(342.91)	114(259.05)	255(465.40)	0	0
Traditional CGE	**522**	30	7.38					

Figure 5.65: Comparison with strategy for mid-rate users in Simulation 1.

BU has a higher probability of failure than the TD policy in both the fully resourced and under-resourced cases, but the average delay of the users being served is lower than that of TD, and BU has chosen BU-Delay-Low-First as a proxy because it can serve all users, and its performance in this respect is no better than that of TD. From the Figure 5.65 and the Figure 5.66, we can see that the performance of the TD policies is very similar, with higher average delay than BU-Delay-Low-First and lower average delay slack than BU, indicating that the performance of BU-Delay-Low-First is lower than that of BU-Delay-Low-First. performance is worse than BU-Delay-Low-First. However, they both outperform traditional cloud gaming engines.



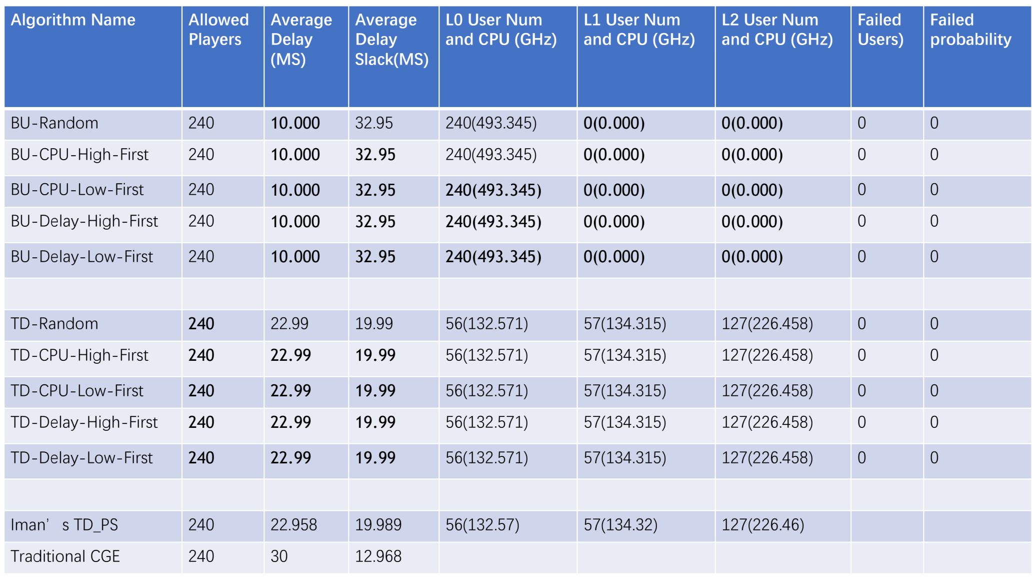
Algorithm Name	Allowed Players	Average Delay (MS)	Average Delay Slack(MS)	L0 User Num and CPU (GHz)	L1 User Num and CPU (GHz)	L2 User Num and CPU (GHz)	Failed Users)	Failed probability
BU-Random	481	11.71	27.38	399.44(799.249)	82.39(170.248)	0(0.000)	34.25	0.07
BU-CPU-High-First	437	14.58	27.2	237(799.981)	200(181.312)	0(0.000)	85	0.16
BU-CPU-Low-First	503	10.5	27.75	478(780.732)	25(163.903)	0(0.000)	19	0.04
BU-Delay-High-First	431	10.14	32.2	425(795.550)	6(24.432)	0(0.000)	91	0.17
BU-Delay-Low-First	522	12.24	25.14	405(799.634)	117(267.729)	0(0.000)	0	0.0
TD-Random	522	21.95	15.43	153(342.913)	114(259.049)	255(465.400)	0	0
TD-CPU-High-First	522	21.95	15.43	153(342.913)	114(259.049)	255(465.400)	0	0
TD-CPU-Low-First	522	21.95	15.43	153(342.913)	114(259.049)	255(465.400)	0	0
TD-Delay-High-First	522	21.95	15.43	153(342.913)	114(259.049)	255(465.400)	0	0
TD-Delay-Low-First	522	21.95	15.43	153(342.913)	114(259.049)	255(465.400)	0	0
Iman’ s TD_PS	522	21.954	15.423	153(342.91)	114(259.04)	255(465.000)	0	0
Traditional CGE		30	7.38					

Figure 5.66: Comparison with strategy for mid-rate users in Simulation 2.

5.4.3 Analyse low-rate user profile

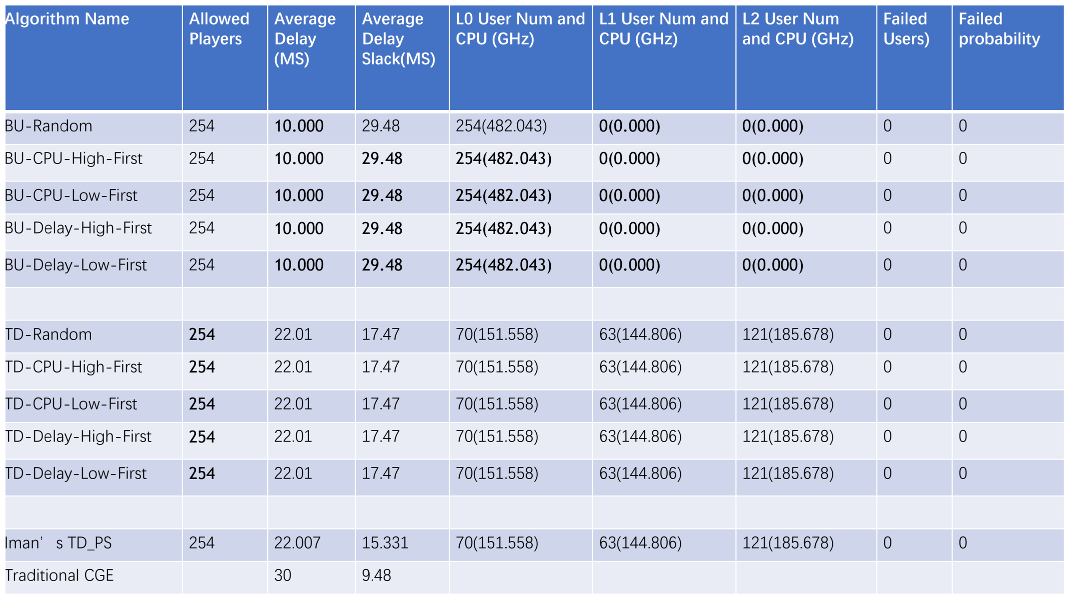
The following Figure 5.67 and the Figure 5.68 show the low-rate user profile, and the low-rate profile represents the distributed engine in the low-load case. It can be seen that both BU and TD policies can perform well in both the fully resourced and under-resourced cases under low load, but the average delay of the users being served is lower compared to TD, and the average delay of Slack is higher. The Figure 5.67 and the Figure 5.68 show that the TD strategies have very similar performance and are much worse than the BU strategies at low loads. However, they both outperform traditional cloud gaming engines.

The Figure and the Figure show that the TD strategies have very similar performance and are much worse than the BU strategies at low loads. However, they both outperform traditional cloud gaming engines.



Algorithm Name	Allowed Players	Average Delay (MS)	Average Delay Slack(MS)	L0 User Num and CPU (GHz)	L1 User Num and CPU (GHz)	L2 User Num and CPU (GHz)	Failed Users)	Failed probability
BU-Random	240	10.000	32.95	240(493.345)	0(0.000)	0(0.000)	0	0
BU-CPU-High-First	240	10.000	32.95	240(493.345)	0(0.000)	0(0.000)	0	0
BU-CPU-Low-First	240	10.000	32.95	240(493.345)	0(0.000)	0(0.000)	0	0
BU-Delay-High-First	240	10.000	32.95	240(493.345)	0(0.000)	0(0.000)	0	0
BU-Delay-Low-First	240	10.000	32.95	240(493.345)	0(0.000)	0(0.000)	0	0
TD-Random	240	22.99	19.99	56(132.571)	57(134.315)	127(226.458)	0	0
TD-CPU-High-First	240	22.99	19.99	56(132.571)	57(134.315)	127(226.458)	0	0
TD-CPU-Low-First	240	22.99	19.99	56(132.571)	57(134.315)	127(226.458)	0	0
TD-Delay-High-First	240	22.99	19.99	56(132.571)	57(134.315)	127(226.458)	0	0
TD-Delay-Low-First	240	22.99	19.99	56(132.571)	57(134.315)	127(226.458)	0	0
Iman’s TD_PS	240	22.958	19.989	56(132.57)	57(134.32)	127(226.46)		
Traditional CGE	240	30	12.968					

Figure 5.67: Comparison with strategy for Low-rate users in Simulation 1.



Algorithm Name	Allowed Players	Average Delay (MS)	Average Delay Slack(MS)	L0 User Num and CPU (GHz)	L1 User Num and CPU (GHz)	L2 User Num and CPU (GHz)	Failed Users)	Failed probability
BU-Random	254	10.000	29.48	254(482.043)	0(0.000)	0(0.000)	0	0
BU-CPU-High-First	254	10.000	29.48	254(482.043)	0(0.000)	0(0.000)	0	0
BU-CPU-Low-First	254	10.000	29.48	254(482.043)	0(0.000)	0(0.000)	0	0
BU-Delay-High-First	254	10.000	29.48	254(482.043)	0(0.000)	0(0.000)	0	0
BU-Delay-Low-First	254	10.000	29.48	254(482.043)	0(0.000)	0(0.000)	0	0
TD-Random	254	22.01	17.47	70(151.558)	63(144.806)	121(185.678)	0	0
TD-CPU-High-First	254	22.01	17.47	70(151.558)	63(144.806)	121(185.678)	0	0
TD-CPU-Low-First	254	22.01	17.47	70(151.558)	63(144.806)	121(185.678)	0	0
TD-Delay-High-First	254	22.01	17.47	70(151.558)	63(144.806)	121(185.678)	0	0
TD-Delay-Low-First	254	22.01	17.47	70(151.558)	63(144.806)	121(185.678)	0	0
Iman’s TD_PS	254	22.007	15.331	70(151.558)	63(144.806)	121(185.678)	0	0
Traditional CGE		30	9.48					

Figure 5.68: Comparison with strategy for Low-rate users in Simulation 2.

The above analysis shows that the new distributed cloud game engine combining edge computing and cloud computing has a significant performance advantage over the traditional cloud game engine, with the Bottom-Up-delay-first and Top-Down strategies both having a much higher performance than the traditional cloud game engine when serving the same number of users. The other BU strategies have low average user latency, but some users are unable to access them successfully, which severely reduces the average user QoE. The performance advantage of the Bottom-Up strategy is further reduced in high-load environments, and as the resources of the edge and intermediate nodes decrease, the performance converges to that of Top-Down, and as the load decreases, the performance of the Top-Down strategy decreases. The advantage of the Top-Down strategy is that it is very stable, serving all users regardless of resource changes in the service nodes and load changes in the entire system, and achieving a stable QoE that is much better than that of traditional cloud gaming engines. There is also the advantage of increasing the edge resources will also significantly improve the performance and average user QoE of the distributed cloud gaming engine. There is also the advantage of increasing the edge The Top-Down strategy also has the advantage that it is easier to achieve load balancing and does not cause congestion on any of the Layer service nodes.

Chapter 6

Conclusion

Due to the growing popularity of cloud gaming, the number of supported game genres is increasing. With the development of cloud gaming engines and the varying services currently offered by cloud gaming providers, it is difficult for users to have a good experience, especially in eSports games such as First Person Shooters (FPS), which require a higher Quality of Experience (QoE). In order to achieve better QoE, this thesis investigates various metrics related to player QoE, and concludes that the user's screen latency is the most important experience metric under the premise of ensuring network quality, and proposes the idea of improving user QoE by combining cloud computing technology and edge computing technology with distributed cloud gaming engine and different allocation allocation algorithms. Based on the proposed approach, a distributed cloud game engine is programmatically built, and multiple sets of user data are used to simulate different load types, and two sets of engine resource data are introduced to simulate resources under different budgets. The simulation results are analyzed in detail. The test results show that the proposed new distributed cloud gaming engine has higher user QoS and QoE than the traditional cloud gaming engine. The Bottoms-Up policy has a significant performance advantage in the light load case, and the user QoE is also higher. In other cases the Top-Down allocation algorithms is more stable, does not lag too far behind in performance, and is easier to load balance and serve more users with fewer resources available, which means it can significantly reduce the equipment budget of the cloud gaming platform and guarantee a decent user QoE. One question that cannot be answered in this thesis is that this allocation method is performed in a static environment, whereas a realistic cloud gaming engine needs to allocate users to manage access in a dynamic environment. This question will remain open until further research on the dynamic allocation algorithms is conducted.

Bibliography

[1] Stefan Hall. How covid-19 is taking gaming and esports to the next level. *2020-05-15*, pages 1–1, 2020.

[2] Gabe Gurwin. Cloud gaming vs. console gaming: The pros and cons of each. *2019-10-28*, pages 1–1, 2019.

[3] Stefan Hall. Microsoft announces global game streaming service, project xcloud, beta next year. *2018-10-08*, pages 1–1, 2018.

[4] Micah Singleton. Google announces project stream, will let testers play assassin's creed odyssey for free. *2018-10-01*, pages 1–1, 2018.

[5] Anandtech. Nvidia's geforce now - grid cloud gaming service goes the subscription way. *2017-05-04*, pages 1–1, 2017.

[6] Jay Peters. How amazon's luna cloud gaming service compares to stadia, xcloud, and geforce now. *2020-09-24*, pages 1–1, 2020.

[7] Sebastian Möller Patrick Le Callet and eds. Andrew Perkis. Qualinet white paper on definitions of quality of experience. *2013-03-12*, pages 1–1, 2012.

[8] Ben Gilbert. Getting to know microsoft's new xbox lead, phil spencer. *2014-04-10*, pages 1–1, 2014.

[9] wiki. x86. https://en.wikipedia.org/wiki/X86#cite_note-4, 2022.

[10] wiki. Arm architecture family. https://en.wikipedia.org/wiki/ARM_architecture_family, 2022.

[11] Springer London. *Distributed Programs*, pages 373–406. Springer London, London, 2009.

[12] Eric Hamilton. "what is edge computing: The network edge explained". *cloudwards.net, 2019-05-14*, page 1, 12 2018.

[13] Asif Laghari, Hui He, Kamran Ali, Rashid Laghari, Imtiaz Halepoto, and Asiya Khan. Quality of experience (qoe) in cloud gaming models: A review. *Multiagent and Grid Systems*, 15:289–304, 10 2019.

[14] Mirko Suznjevic, Lea Skorin-Kapov, and Maja Matijasevic. The impact of user, system, and context factors on gaming qoe: A case study involving mmorpgs. In *2013 12th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6, 2013.

[15] Tom Beigbeder, Rory Coughlan, Corey Lusher, John Plunkett, Emmanuel Agu, and Mark Claypool. The effects of packet loss and latency on player performance in unreal tournament 2003. *NETGAMES*, pages 144–151, 01 2004.

[16] Phu Lai, Qiang He, Guangming Cui, Feifei Chen, Mohamed Abdelrazek, John Grundy, John Hosking, and Yun Yang. Quality of experience-aware user allocation in edge

computing systems: A potential game. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 223–233, 2020.

[17] L. De Giovanni, D. Gadia, P. Giaccone, D. Maggiorini, C. Palazzi, L. Ripamonti, and G. Sviridov. Revamping cloud gaming with distributed engines. *IEEE Internet Computing*, (01):1–1, may 5555.

[18] Kajal Claypool and Mark Claypool. On frame rate and player performance in first person shooter games. *Multimedia Syst.*, 13:3–17, 07 2007.

[19] Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. Measuring the latency of cloud gaming systems. pages 1269–1272, 11 2011.