Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут телекомунікаційних систем
(повна назва інституту/факультету)

Кафедра телекомунікацій
(повна назва кафедри)

До захисту допущено
Завідувач кафедри
_____ Явіся В.С.
(підпис) (ініціали, прізвище)
“___”_____20__ р.

Дипломна робота

на здобуття освітнього ступеня “бакалавр”

Спеціальність 172 Телекомунікації та радіотехніка
(код і назва)

на тему: Метод зменшення часу доставки повідомлень в IP мережі за рахунок модифікації протоколу UDP

Виконав : студент 4 курсу, групи ТЗ-61
(шифр групи)

Педько Андрій Дмитрович __ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник:асистент кафедри ТК, к.т.н., Маньківський В.Б. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент: к.т.н., доцент Созонник Г.Д. _____
(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

Національний технічний університет України
"Київський політехнічний інститут імені Ігоря Сікорського"

Інститут телекомунікаційних систем
(повна назва)

Кафедра телекомунікацій
(повна назва)

Освітній ступінь бакалавр
Спеціальність 172 Телекомунікації та радіотехніка
(шифр і назва)
Програма професійного спрямування Технології та засоби телекомунікацій
(назва)

ЗАТВЕРДЖУЮ
В.о.завідувача кафедри
Явіся В.С. _____
(прізвище ініціали) (підпис)
"___"_____2020 р.

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Педьку Андрію Дмитровичу

(прізвище, ім'я, по батькові)

1. Тема роботи: Метод зменшення часу доставки повідомлень в IP мережі за рахунок модифікації протоколу UDP
Керівник роботи: Маньківський Володимир Броніславович, асист. кафедри ТК, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від _03_ березня 2020р. №924-с

2. Термін подання студентом роботи 04.06.2020

3. Вихідні дані до роботи структура мережі на базу стеку UDP/IP – стандартний перелік елементів мережі, стандартний перелік елементів мережі TCP, модель одноканальної системи массового обслуговування, потік подій стохастичний,

модель обслуговування FIFO з єдиною фазою обслуговування, стандартні типи загроз, стандартні методи захисту.

4. Зміст роботи дослідження протоколів транспортного рівня та їх порівняння з протоколом UDP. Визначення основних параметрів протоколу UDP котрі впливають на втрату пакетів в мережі та швидкість передачі даних. Побудова мережі для тестування протоколу зі зміненими параметрами в GNS3. Реалізація додатків сервера та клієнта на віртуальних машинах. Вибір параметрів модифікованого протоколу UDP для пришвидшення відправки повідомлень з мінімальними втратами в мережі.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо):

Слайд №1 Назва, область дослідження, предмет дослідження

Слайд №2 Задачі дослідження

Слайд №3 Область дослідження

Слайд №4 Порівняння протоколів транспортного рівня

Слайд №5 Основні параметри модифікації протоколу UDP

Слайд №6 Packet pacing

Слайд №7 Maximum Transmission Unit

Слайд №8 Побудова мережі для тестування

Слайд №9 Вибір маршрутизації для тестового макету

Слайд №10 Написання додатків клієнту та сервера

Слайд №11Вплив модифікації буферу відправки на швидкість передачі

Слайд №12Вплив packet passing при передачі

Слайд №13Оптимальні показники модифікованого протоколу UDP

Слайд №14 Передача стандартним UDP

Слайд №15 Порівняння модифікованого UDP з стандартним TCP

Слайд №16 Висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7.Дата видачі завдання 22.01

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного роботи	Строк виконання етапів роботи	Примітка
1	Складання тестового середовища в GNS 3.0	25.01.2020 – 18.02.2020	
2	Скаладання досліджуваної мережі в тестовому середовищі	20.02.2020 – 30.02.2020	
3	Написання першого розділу дипломного проекту присвяченого області дослідження - протоколи транспортного рівня моделі OCI	02.03.2020 – 19.03.2020	
4	Складання порівняльной таблиці протоколів TCP та UDP	21.03.2020 – 28.03.2019	
5	Обгрунтування вибору параметрів протоколу UDP, підлягаючих до зміни	03.04.2020 – 20.04.2020	
6	Написання програмного забезпечення клієнта для передачі повідомлень по протоколу UDP	22.04.2020 – 13.05.2020	
7	Написання програмного забезпечення серверу для отримання повідомлень по протоколу UDP	14.05.2020 – 25.05.2020	
8	Дослідження впливу зміни параметрів протоколу UDP на швидкість доставки повідомлення та втрати пакетів в IP-мережі	26.05.2020 – 03.06.2020	

Студент _____ **Педько А.Д.**
(підпис) (прізвище та ініціали)

Керівник роботи _____ **Маньківський В.Б.**
(підпис) (прізвищетаініціали)

РЕФЕРАТ

Дипломна робота містить: 104 сторінки, 73 рисунків, 4 таблиці, 16 посилань.

У роботі порівняно найпоширеніші протоколи транспортного рівня і вказані їх недоліки. Не зважаючи на всі переваги UDP протоколу над TCP, в сучасних мережах протокол UDP не задовольняє вимогам по швидкості. Основною сферою де використовується протокол UDP є закриті корпоративні мережі. В теперішній час, в таких мережах компанії модифікують протокол UDP відповідно до своїх вимог. При цьому відсутні стандарти та рекомендації щодо модифікації протоколу UDP, тому розробники беруть ризики на власний рахунок.

TCP протокол гарантує доставку за рахунок того, що на рівні протоколу закладено підтвердження доставки приймаючою стороною і якщо підтвердження про доставку відсутнє, то відправник повторює відправку пакетів по тайм-ауту, якщо вчасно не отримав таке підтвердження.

Гарантія доставки не є основним критерієм вибору протоколу. Постає питання вартості забезпечення гарантій доставки. Протокол TCP вимагає встановлення з'єднання, цей процес складається з обміну трьома пакетами. Після того як з'єднання встановлено при передачі даних використовуються періодичні підтвердження прийому інформації та повторні відправки, в разі якщо вона не дійшла. Для цього ядру операційної системи необхідно пам'ятати стан всіх відкритих TCP з'єднань і підтримувати буфери для прийнятої / переданої інформації. А з додатком необхідно для спілкування з кожним клієнтом використовувати окреме з'єднання, при тому що в більшості ОС є серйозні обмеження на кількість одночасно відкритих процесом дескрипторів файлів / з'єднань (в більшості ОС 255). Наприклад, Windows XP за

замовчуванням одночасно обслуговує не більше 5 з'єднань, що знаходяться у відкритому стані.

При цьому UDP не гарантує доставку на рівні протоколу, весь контроль виконується на рівні додатку, у якого відсутні обмеження на контроль максимальної кількості відкритих з'єднань.

В роботі описано метод модифікації протоколу UDP для невеликих приватних мереж та визначено параметри котрі необхідно змінити для досягнення більшої швидкості передачі даних з мінімальною втратою датаграм в мережі. А також побудована масштабуєма мережа для майбутнього тестування протоколу UDP зі зміненими параметрами.

Основним критерієм при модифікації протоколу UDP є досягнення необхідних швидкостей передачі з низькими втратами в мережі. Проблематика полягає в визначенні таких параметрів протоколу UDP при яких буде забезпечена більша швидкість передачі повідомлень. При цьому такі параметри як гарантування доставки повідомлення, відновлення втрачених пакетів, повторна відправка повідомлень не розглядаються. Тому що дані параметри є неактуальними з урахуванням особливостей протоколу UDP та специфіку використання протоколу в замкнутих корпоративних мережах.

Ключові слова: IP - мережа, параметри, UDP, протокол, транспортний рівень, швидкість доставки, packet loss.

Abstract

Thesis contains: 104 pages, 73 figures, 4 tables, 16 references.

The thesis compares the most common protocols of the transport level and their shortcomings. Despite all the advantages of the UDP protocol over TCP, in modern networks, the UDP protocol does not meet the requirements for speed. The main area where the UDP protocol is used is closed corporate networks. Currently, in such networks, companies are modifying the UDP protocol according to their requirements. However, there are no standards and recommendations for modifying the UDP protocol, so developers take risks at their own expense.

TCP protocol guarantees delivery due to the fact that at the protocol level there is a confirmation of delivery by the receiving party and if there is no confirmation of delivery, the sender repeats the sending of packets on timeout, if not received in time.

Delivery guarantee is not the main criterion for selecting a protocol. The question arises as to the cost of providing delivery guarantees. TCP requires a connection, this process consists of exchanging three packets. Once the connection is established, periodic acknowledgments of receiving information and resending are used during data transmission. To do this, the kernel of the operating system must remember the state of all open TCP connections and maintain a buffer for received / transmitted information. And with the application it is necessary to use a separate connection to communicate with each client, although in most OS there are serious restrictions on the number of simultaneously open process descriptors of files / connections (in most OS 255). For example, Windows XP defaults to up to 5 open connections at a time.

However, UDP does not guarantee delivery at the protocol level, all control is performed at the application level, which has no restrictions on controlling the maximum number of open connections.

The paper describes the method of modification of the UDP protocol for small private networks and identifies the parameters that need to be changed to achieve higher data rates with minimal loss of datagrams in the network. And also the scalable network for the future testing of the UDP protocol with the changed parameters is constructed.

The main criterion when modifying the UDP protocol is to achieve the required transmission speeds with low network losses. The problem is to determine such parameters of the UDP protocol at which a higher speed of message transmission will be provided. In this case, such parameters as guaranteeing the delivery of the message, recovery of lost packets, re-sending messages are not considered. Because these parameters are irrelevant given the peculiarities of the UDP protocol and the specifics of the use of the protocol in closed corporate networks.

Key words: IP - network, parameters, UDP, protocol, transport layer, delivery speed, packet loss.

Пояснювальна записка
до дипломної роботи

на тему: Метод зменшення часу доставки повідомлень в IP мережі за рахунок модифікації протоколу UDP

Київ – 2020 року

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

IP - Міжмережевий протокол

HTTP - Протокол передачі гіпертексту

TCP - Протокол керування передачею

QoS - Якість обслуговування

MTU - Розмір даних котрі можна переправити по мережі

UDP - Протокол передачі датаграм

API - Прикладний програмний інтерфейс

BBR – Вид congestion control протоколу TCP

3G - Третє покоління технології мобільного зв'язку

LTE – Мобільний протокол передавання даних

SSL – Рівень захищених сокетів

OSI - Модель взаємозв’язку відкритих систем

RTT – Час затримки

ARP – Протокол визначення адрес

MAC - Управління доступом до носія

LLC - Підрівень керування логічним зв'язком у комп'ютерних мережах

FTP – Протокол передачі файлів

SMTP - Простий протокол пересилання пошти

RTO - Допустимий час простою сервісу в разі збою

ATM - Асинхронний спосіб передачі даних

DNS - Система доменних імен

PPP - Протокол "точка-точка"

ICMP - Протокол міжмережевих керуючих повідомлень

RIPv2 - Протокол маршрутної інформації

VoIP - Телефонний зв’язок через Інтернет-протокол

GNS3 - Графічний мережевий симулятор

ASIC - інтегральна схема спеціального призначення

ВСТУП

Актуальність теми

Завдяки виникненню і розвитку мереж передачі даних, з'явився новий високоефективний спосіб взаємодії між людьми. Спочатку мережі використовувалися, головним чином, для наукових досліджень, але потім вони стали проникати буквально в усі сфери людської діяльності. При цьому більшість мереж існувало абсолютно незалежно один від одного, вирішуючи конкретні завдання для конкретних груп користувачів. Відповідно до цих завдань вибиралися ті чи інші мережеві технології і апаратне забезпечення. Побудувати універсальну фізичну мережу з однотипної апаратури просто неможливо, оскільки така мережа не могла б задовольняти потреби потенційних користувачів. З одного боку, потрібна високошвидкісна мережа для з'єднання мережевих машин в межах будівлі, а з іншого - надійні комунікаційні мережі між комп'ютерами, рознесеними на сотні кілометрів. Попит на високошвидкісну передачу даних постійно зростає. Потреба в цьому може бути знайдена в самих різних галузях - від розваг до наукових досліджень. Однак є кілька проблем, які перешкоджають можливостям мережевих додатків. Одним з них є повільна обробка пакетів через значні накладні витрати на системні виклики для простих мережевих операцій. Існують апаратні рішення, але з економічної точки зору кращим є використання застарілого рівноправного обладнання через високу вартість поновлення мережевої інфраструктури. Таким чином, актуальність полягає у програмному вирішенні цих проблем. Одним з рішень є підхід до зміни параметрів протоколу транспортного рівня, при цьому, в пріоритеті визначати параметри спрямовані на високошвидкісну передачу даних по мережах з втратами і високою затримкою.

Протокол побудований на стандартних Linux-сокетах UDP. Таким чином, він сильно залежить від продуктивності мережевого стека. Метою даної роботи

є підвищення продуктивності передачі за допомогою зміни параметрів протоколу UDP в мережі і оцінка переваг такої реалізації. Використовуючи протокол UDP, можна знизити навантаження мережі. Оскільки цей протокол не потребує встановлення з'єднання з сервером, а також не використовує підтвердження про отримані дані, це може значно підвищити швидкість передачі повідомлень, тим самим знизити чергу на відправлення пакетів по мережі.

Областю дослідження являється транспортний рівень моделі OSI.

Найбільш використовуваним на сьогоднішній день є стек протоколів TCP/IP [1]. Дані породжуються на рівні додатків конкретної користувальницької програмою і передаються на транспортний рівень. Програма, як правило, використовує ір адресу (або доменне ім'я) цільового пристрою і передбачає, що на тому кінці має бути запущено додаток, що підтримує той же самий протокол рівня додатків, що і на передавальному пристрої. Отже транспортний рівень - це перший рівень, який зустрічає дані додатки і починає процедуру по їх підготовці до передачі.

На транспортному рівні широко розповсюджені два протоколи: TCP і UDP.

Протоколи транспортного рівня забезпечують прозору доставку даних між двома прикладними процесами. Процес, який одержує або відправляє дані, за допомогою транспортного рівня ідентифікується на цьому рівні номером, котрий називається номером порту. Таким чином, роль адреси відправника і одержувача на транспортному рівні виконується номером порту. Аналізуючи заголовок свого пакета, отриманого від мережевого рівня, транспортний модуль визначає за номером порту одержувача по якому з прикладних процесів спрямовані дані і передає ці дані відповідному прикладному процесу.

Номер порту одержувача і відправника є невід’ємним атрибутом при передачі даних мережею при використанні транспортного рывня TCP/IP, котрий відправляє дані. Заголовок транспортного рівня містить також і деяку іншу службову інформацію. А формат заголовка залежить від протоколу транспортного рівня, який використовується.

На сьогоднішній день в мережі Інтернет часто використовуються два протоколи транспортного рівня - UDP, що забезпечує негарантовану доставку даних між програмами, і TCP, що забезпечує гарантовану доставку з встановленням віртуального з'єднання.

TCP / IP (Transmission Control Protocol / Internet Protocol) - це набір протоколів зв'язку, які використовуються для з'єднання мережевих пристроїв в Інтернеті. TCP / IP також може використовуватися як протокол зв'язку в приватній мережі (інтранет або екстранет). [1]

UDP (User Datagram Protocol) - один з ключових елементів набору мережевих протоколів для мережі Інтернет. З UDP комп'ютерні програми можуть надсилати повідомлення (в даному випадку датаграми) іншим хостам по IP-мережі без необхідності попереднього повідомлення для установки спеціальних каналів передачі або шляхів даних. Протокол був розроблений Девідом П. Рідом в 1980 році і офіційно визначений в RFC 768. [2]

Проблематика

Обидва протоколи TCP і UDP–функціонують на транспортному рівні та вирішують завдання доставки повідомлень. При цьому дані протоколи використовують різні підходи при формуванні повідомлень.

Обмін даними орієнтований на встановлення з'єднання. Для забезпечення якого протокол рівня 4 посилає підтвердження про отримання даних і запитує повторну передачу, якщо дані не отримані або перемішані. Протокол TCP

використовує саме такий надійний зв'язок. TCP використовується в таких прикладних протоколах, як HTTP, FTP, SMTP і Telnet.[1]

Протокол TCP вимагає, щоб перед відправкою повідомлення було встановлено з'єднання. Серверний додаток має виконати так зване пасивне відкриття (passive open), щоб створити з'єднання з відомим номером порту, і, замість того щоб відправляти виклик в мережу, сервер переходить в очікування надходження вхідних запитів. Клієнтський додаток повинен виконати активне відкриття (active open), відправивши серверному додатку синхронізуючий порядковий номер (SYN), що ідентифікує з'єднання. [1]

На відміну від TCP, UDP - дуже швидкий протокол, оскільки в ньому використовується самий мінімальний механізм, необхідний для передачі даних. Звичайно, він має деякі недоліки. Доставка UDP не гарантує доставку даних, датаграма має шанс загубитися, і можуть бути отримані дві копії одного і того ж повідомлення. [3]

UDP не вимагає встановлення з'єднання, і дані можуть бути відправлені відразу ж, як тільки вони підготовлені. UDP не надсилає повідомлення підтвердження, тому дані можуть бути отримані або втрачені. Якщо при використанні UDP потрібна надійна передача даних, її слід реалізувати в протоколі вищого рівня.

Таким чином, використовуючи протокол UDP, можна знизити навантаження мережі. Оскільки цей протокол не потребує встановлення з’єднання з сервером, а також не використовує підтвердження про отримані дані, що може значно підвищити швидкість передачі повідомлень, тим самим знизити чергу на відправлення пакетів.

Незважаючи на всі переваги UDP протоколу над TCP, в сучасних мережах протокол UDP не задовольняє вимогам по швидкості. Основною сферою де використовується протокол UDP є закриті корпоративні мережі. В теперішній час, в таких мережах компанії модифікують протокол UDP відповідно до своїх

вимог. При цьому відсутні стандарти та рекомендації щодо модифікації протоколу UDP, тому розробники беруть ризики на власний рахунок.

Основним критерієм при модифікації протоколу UDP є досягнення необхідних швидкостей передачі. Вимоги по безпеці та гарантуванні доставки повідомлень виносяться на рівень інфраструктури підприємства, яка представляє собою закриту мережу.

Проблематика полягає в тому щоб визначити такі параметри протоколу UDP при яких буде забезпечена більша швидкість передачі повідомлень. При цьому такі параметри як гарантування доставки повідомлення, відновлення втрачених пакетів, повторна відправка повідомлень не розглядаються. Тому що дані параметри є неактуальними з урахуванням особливостей протоколу UDP та специфіку використання протоколу в замкнутих корпоративних мережах.

Ціль та задачі

Отже, основним завданням модифікації протоколу UDP є :

- досягнення приросту швидкості передачі даних, завдяки зміненню основних параметрів протоколу, наприклад, зменшення ваги датаграм;
- знизити можливість перестановки пакетів місцями (в реальному житті досить рідкісне явище, але трапляється в 0,2% випадків);
- оптимізувати роботу мережі для досягнення рівномірного відправлення та отриманні датаграм(jitter при голосовому типу трафіка);
- зменшити втрату пакетів при передачі точка-точка;

Крім того, в процесі експлуатації стандартного UDP, може виникнути ряд невизначеностей, які можуть привести до стандартних недоліків протоколу UDP, які мають різну ймовірність прояву при експлуатації. Ці помилки, як правило,виявляються після закінчення розрахунку параметрів налаштування UDP протоколу, на етапі експлуатації. Це вимагає повернення до початкового етапу розрахунку мережі в цілому.

Таким чином, вдосконалення математичного апарату, котрий дозволяє враховувати особливості функціонування мереж з конкретною технологією і усунення невизначеностей в процесі рішення завдань, дозволяє підвищити ефективність застосування вже існуючих методик.

Все вище сказане, визначило необхідність подальших наукових досліджень в цій предметній області, а саме, рішення актуальної наукової задачі по розробці нової імплементації протоколу UDP.

РОЗДІЛ 1. ІСНУЮЧА ПРОБЛЕМА ВИБОРУ ПРОТОКОЛУ ТРАНСПОРТНОГО РІВНЯ В СУЧАСНИХ МЕРЕЖАХ

1.1. Область дослідження

Модель взаємозв’язку відкритих систем OSI (Рис. 1.1) визначає мережеві рамки для реалізації протоколів у рівнях, при цьому управління передається від одного рівня до іншого. В основному модель сьогодні використовується як навчальний інструмент. Модель концептуально розділяє архітектуру комп'ютерної мережі на 7 логічних рівнів.

Дані	7 прикладний application	Доступ до мережевих служб
	6 представлень presentation	Представлення і кодування даних
	5 сеансовий session	Управління сеансом зв’язку
Сегменти	4 транспортний transport	Прямий зв’язок між кінцевими пунктами і надійність
Пакети	3 мережевий network	Визначення маршруту і логічна адресація
Кадри	2 канальний data link	Фізична адресація
Біти	1 фізичний physical	Робота з середовищем передачі , сигналами і двійковими даними

Рисунок 1.1.Коротке описання моделі OSI.

Нижні рівні моделі обробляють електричні сигнали, фрагменти двійкових даних та маршрутизацію цих даних по мережах. Вищі рівні моделі охоплюють мережеві запити та відповіді, представлення даних та мережеві протоколи, як

видно з точки зору користувача. Модель OSI спочатку була задумана як стандартна архітектура для побудови мережевих систем.

Фізичний рівень

Перший рівень моделі OSI відповідає за передачу цифрових бітів даних від фізичного рівня пристрою, що надсилає (джерело), через мережеві носії зв'язку, до фізичного рівня пристрою прийому (призначення). Приклади технологій першого рівня включають кабелі та концентратори Ethernet. Крім того, концентратори та інші ретранслятори - це стандартні мережеві пристрої, які функціонують на фізичному рівні, як і кабельні з'єднувачі.

Канальний рівень

Отримуючи дані з фізичного рівня, канальний рівень перевіряє наявність фізичних помилок передачі та пакує біти в "кадри" даних. Цей рівень також керує схемами фізичної адреси, такими як MAC-адреси для мереж Ethernet, контролюючи доступ будь-яких різних мережевих пристроїв до фізичного середовища. Оскільки рівень Data Link є єдиним найскладнішим у моделі OSI, його часто ділять на дві частини, підрівень Media Access Control і підрівень Logical Link Control.

Мережевий рівень

Коли дані надходять на мережевий рівень, адреси джерела та призначення, що містяться всередині кожного кадру, перевіряються, щоб визначити, чи досягли дані кінцевого пункту призначення. Якщо дані досягли кінцевого пункту призначення, цей рівень формує дані в пакети, доставлені до транспортного рівня. В іншому випадку мережевий рівень оновлює адресу призначення та відкидає кадр назад до низів.

Для підтримки маршрутизації мережевий рівень підтримує логічні адреси, такі як IP-адреси для пристроїв у мережі. Мережевий рівень також управляє відображенням між цими логічними адресами та фізичними адресами. В

мережах IP, це відображення здійснюється за допомогою протоколу адресної розв’язки (ARP).

Транспортний рівень

Транспортний рівень - це рівень у моделі відкритого системного взаємозв’язку (OSI), відповідальний за зв'язок між кінцевими пунктами мережі. Він забезпечує логічну комунікацію між прикладними процесами, що працюють на різних хостах.

Сеансовий рівень

Сеансовий рівень керує послідовністю і потоком подій, які ініціюють і розривають мережеві з'єднання. Рівень 5 побудований для підтримки декількох типів з'єднань, які можна створювати динамічно і працювати через окремі мережі.

Рівень представлення

Цей рівень є найпростішим у функціонуванні будь-якого фрагмента моделі OSI. Він здвйснює обробку синтаксису даних повідомлень, таку як перетворення формату та шифрування / дешифрування, необхідні для підтримки рівня додатків над ним.

Прикладний рівень

Прикладний рівень постачає мережеві послуги кінцевим користувачам. Мережеві послуги, як правило, є протоколами, які працюють з даними користувача. Наприклад, у програмі веб-браузера протокол HTTP рівня додатків пакує дані, необхідні для надсилання та отримання вмісту веб-сторінки. Цей рівень 7 надає дані (і отримує дані з) рівня презентації.

Областю дослідження цієї роботи є транспортний рівень, отже, розглянемо його детальніше.

Транспортний рівень відповідає за встановлення тимчасової сесії зв'язку між двома додатками та доставку даних між ними. Додаток генерує дані, що надсилаються з програми на вихідному хості до програми на хості призначення.

Це не стосується типу хоста, типу носія, по якому мають проходити дані, шлях, який проходять дані, перевантаженість або розмір мережі.

Функції транспортного рівня:

До основних функцій транспортного рівня можна виділити в порядку критичності реалізації:

1. Відстеження окремих сеансів зв’зку

На транспортному рівні кожен певний набір даних, що передається між додатком джерела і додатком призначення, називається сеансом зв'язку. Вузол може мати кілька додатків, які одночасно обмінюються даними по мережі. Кожен з цих додатків взаємодіє з одним або декількома іншими додатками на одному або декількох віддалених вузлах. Транспортний рівень відповідає за підтримку та відстеження цих сеансів зв'язку.

2. Сегментація даних і подальша збірка сегментів

Дані необхідно підготувати для пересилки в середовищі, розділивши їх на відповідні для цього частини. У більшості мереж існують обмеження на обсяг даних, які можна включити в один пакет. Протоколи транспортного рівня включають сервіси, які поділяють дані додатків на окремі блоки необхідного розміру. Такий сервіс забезпечує інкапсуляцію, необхідну для кожної частини даних. До кожного блоку даних додається заголовок, який надалі використовується для повторного складання. Цей заголовок дозволяє відстежувати потік даних.

На вузлі призначення транспортний рівень повинен забезпечити відновлення окремих частин в один повний потік даних, придатний для обробки на рівні додатків. Протоколи на транспортному рівні описують, як використовувати інформацію в заголовку транспортного рівня для повторного складання частин даних в потоки з метою їх подальшої передачі на рівні додатків.

3. Визначення додатків

Щоб переслати потоки даних відповідних додатків, транспортному рівню необхідно визначити цільовий додаток. Для цього транспортний рівень привласнює кожному з додатком окремий ідентифікатор - номер порту. Кожному програмному процесу, якому потрібен доступ до мережі, призначається номер порту, унікальний для цього вузла.

На транспортному рівні функціонує велика кількість протоколів, але оскільки найбільш широкого поширення набули протоколи TCP та UDP. Ці протоколи підтримують більшість пристроїв з котрих складається будь-яка мережа. Отже, саме на цих протоколах і буде порівняно модифікований UDP протокол в рамках даної дипломної роботи.

1.2. Майбутнє мережевих протоколів: TCP або UDP

Перед кожним сервісом, котрий генерує хоча б 1 Мбіт / сек трафіку в інтернеті виникає питання: «Яким протоколом передавати? по TCP або по UDP?». У прикладних областях, а також і платформах доставки вже склалися методи і традиції прийняття подібних рішень. На початку 2000-х років [4] всі були абсолютно впевнені, що UDP - це протокол про негарантовану доставку. Якщо потрібен надійний протокол то слід обирати TCP.

TCP	UDP
Надійний	Ненадійний
Орієнтований на встановлення зв’язку з сервером	Без встановлення зв’язку
Ретрансляція сегментів та контроль потоку	Без ретрансляції та контролю потоку
Послідовність сегментів	Без послідовності
Підтвердження при отриманні пакетів	Немає підтвердження

Таблиця1.1. Порівняння протоколів TCP та UDP.

TCP забезпечує надійну і впорядковану доставку потоку байтів від користувача на сервер або навпаки. UDP не призначений для встановлення з’єднання та не перевіряє готовність приймача .

Відповідно до характеристик порівняння (див. Табл. 1.1), наведемо основні агрументи, розкриваючи характеристики:

1. Надійність.

TCP є більш надійним, оскільки використовує підтвердження повідомлень та повторну передачу у разі втрачених пакетів. Таким чином, втрачених даних майже немає. UDP не гарантує, що датаграма досягне приймача, оскільки поняття підтвердження, тайм-ауту та повторної передачі відсутні.

2. Встановлення з'єднання.

TCP - це протокол котрий вимагає встановлення з'єднання. UDP - це легкий транспортний протокол, розроблений на вершині IP котрий не відстежує та не впорядковує повідомлення.

3. Виявлення помилок.

UDP працює на основі "найкращих зусиль". Протокол підтримує виявлення помилок через контрольну суму, але коли виявлена помилка, пакет відкидається. Повторна передача пакета для відновлення після цієї помилки не робиться. Це пояснюється тим, що UDP зазвичай використовується для програм, залежних від часу, таких як ігри або передача голосу. Відновлення помилки було б безглуздим, оскільки до моменту отримання повторно відправленого пакету він не принесе користі.

TCP використовує як виявлення помилок, так і відновлення помилок. Помилки виявляються за допомогою контрольної суми, і якщо пакет помилковий, одержувач не підтверджує, що ініціює повторну передачу відправника.

Потік даних, який ви відправляєте, розбивається на пакети, якийсь чорний ящик доставляє ці пакети до клієнта. Клієнт збирає пакети і отримує потік

даних. Зазвичай це все прозоро і немає необхідності думати, що там на нижніх рівнях.

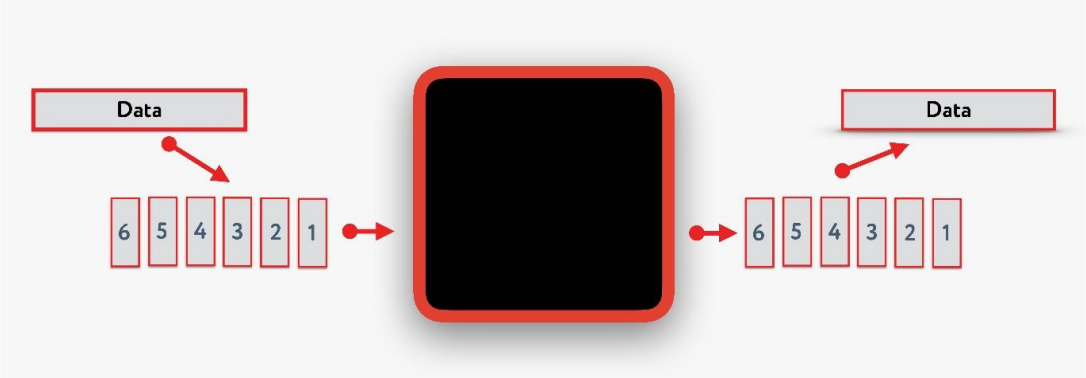


Рисунок 1.2 Дані в IP мережах передаються пакетами

І там, і там є порти. Але в UDP є тільки контрольна сума - довжина пакета, цей протокол максимально простий. А в TCP - дуже багато даних, які явно вказують вікно, acknowledgement, sequence, пакети і так далі. Очевидно, що TCP складніший. Якщо говорити дуже грубо, то TCP - це протокол надійної доставки, а UDP - ненадійної.

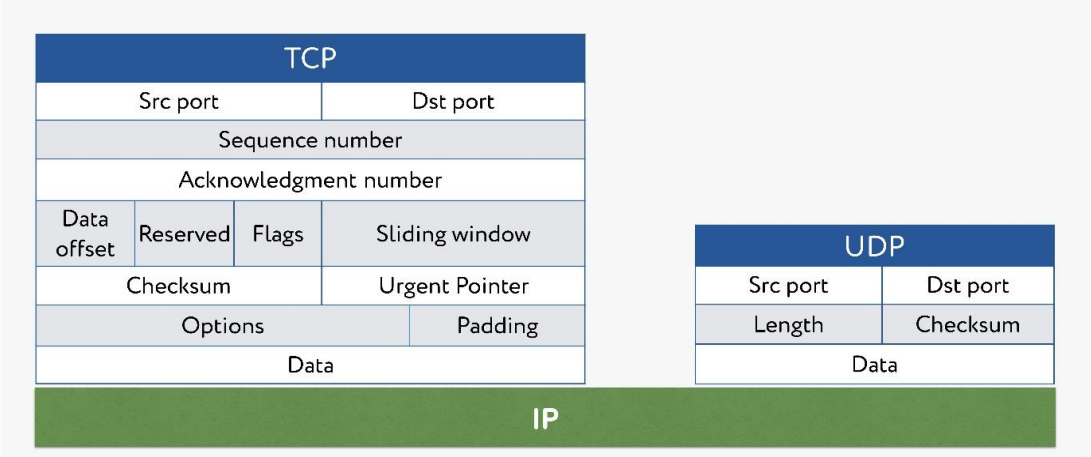


Рисунок 1.3 Порівняння пакета TCP та датаграми UDP.

І все ж, незважаючи на заявлену ненадійність UDP, в ході виконання роботи побачимо, чи можливо доставити дані швидше і надійніше ніж з використанням TCP.

1.3. TCP: переваги та недоліки

Мобільний світ переміг, з'явилися бездротові протоколи, а TCP був як і раніше залишається незмінною. На початок 2020 року 80% користувачів використовують Wi-Fi або бездротову 3G-4G мережу.

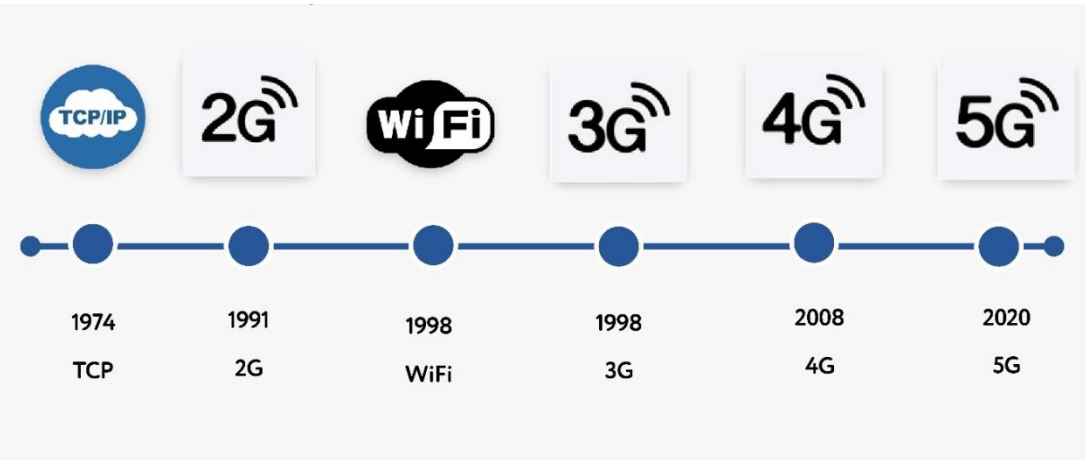


Рисунок 1.4. TCP та бездротові технології

У бездротових мережах існують особливості, які притаманні саме такому фізичному рівні моделі OCI:

• packet loss - приблизно 0,6% пакетів, які ми відправляємо, втрачаються по дорозі;

• reordering - перестановка пакетів місцями, в реальному житті досить рідкісне явище, але трапляється в 0,2% випадків;

• jitter - коли пакети відправляються рівномірно, а приходять чергами з затримкою приблизно в 50 мс.

Всі ці особливості передачі даних в гетерогенних мережах TCP успішно приховує. Тобто в середньому у користувачів (якщо виключити західну частину країн СНД): пропускна здатність 1,1 Мбіт / сек, 0,6% packetloss, RTT (round-triptime) близько 200 мс.

Це можливо перевірити на прикладі споживання контенту в залежності від швидкості інтернету - є багато статистичних даних. Згідно статистиці[5],

яка говорить, що чим вище швидкість інтернету в країні, тим більше користувачі дивляться відео.

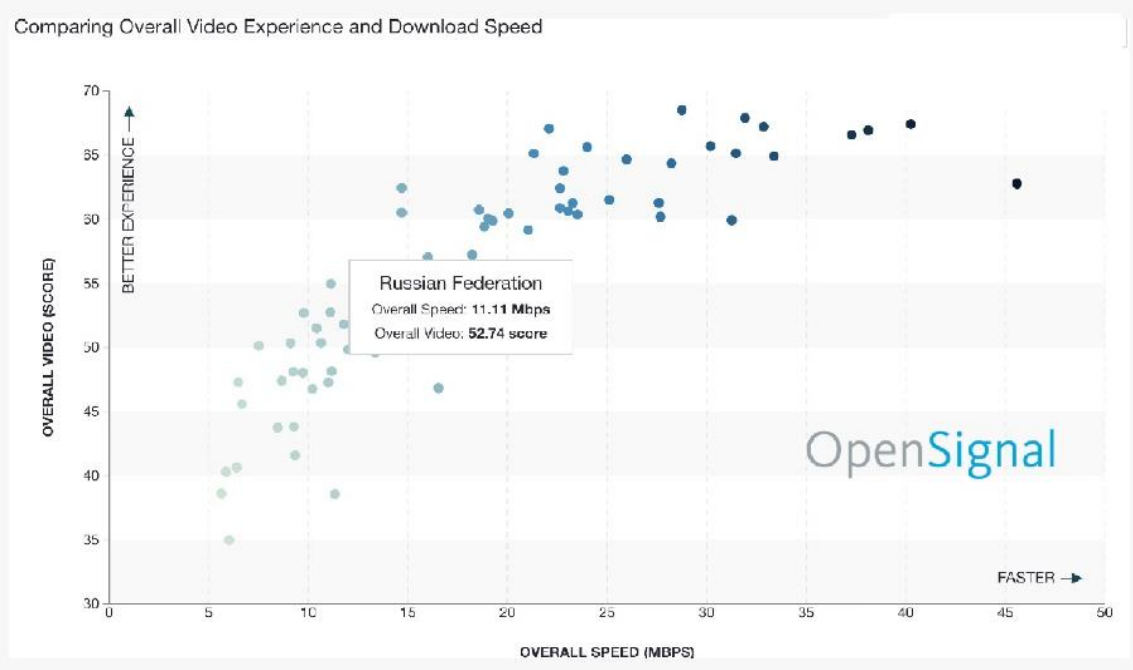


Рисунок 1.5. Використання мобільного відео в залежності від пропускної здатності

Згідно з цією статистикою в країнах СНД досить швидкий Інтернет, однак за іншими даними середня швидкість трохи нижче.

На користь того, що швидкість інтернету в цілому недостатня, говорить те, що всі творці популярних додатків, соціальних мереж, відеосервісів і так далі оптимізують свої сервіси для роботи в поганій мережі. Вже після 10 Кбайт отриманих даних можна побачити мінімум інформації в стрічці, а на швидкості 500 Кбіт/с можна дивитися відео.

Основне питання полягає в тому, як прискорити завантаження. Розробники в процесі розробки платформи для трансляції відео, зрозуміли, що TCP не дуже ефективний в бездротових мережах, саме через особливості

фізичного рівня – бездротового [6]. Було вирішино прискорити завантаження і зробити наступний крок в оптимізації передачі.

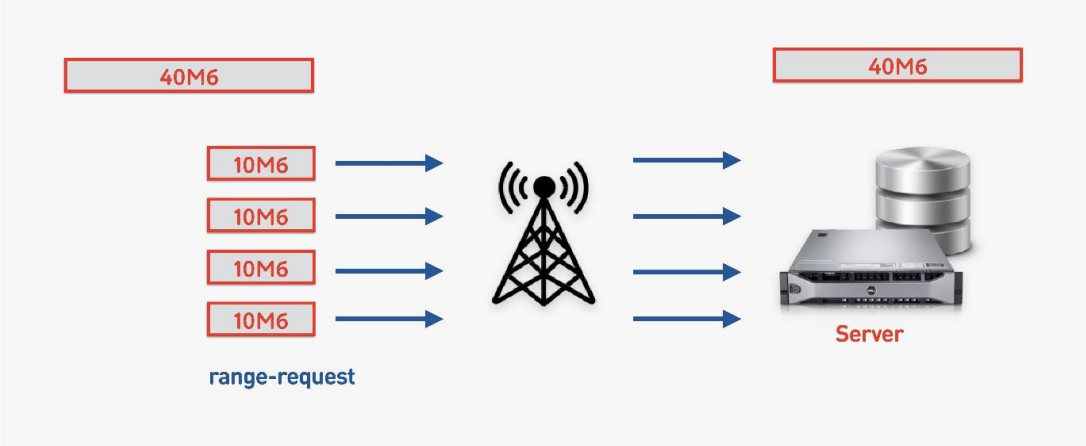


Рисунок 1.6. Паралельне завантаження

Вантаживши відео з клієнта на сервер, в декілька потоків, тобто 40 Мбайт ділимо на 4 частини по 10 Мбайт(Рис 1.6) і завантажуємо їх паралельно. Запустили це на Android і отримали, що паралельно завантажується швидше, ніж в одне з'єднання. Найцікавіше, що тестуючи паралельне завантаження , було зрозуміло, що в деяких регіонах швидкість завантаження виросла в 3 рази .

По чотирьом TCP-з'єднаннях можна завантажити дані на сервер в 3 рази швидше. Таким чином, автори роботи [6] прийшли до висновку, що краще використовувати паралельне з’єднання для відвищення швидкості передачі відео потоку .

1.4. TCP в нестабільних мережах

Неймовірний ефект з паралельним завантаженням можна перевірити наступним чином. Досить взяти вимірювач швидкості отримання / відправки даних (наприклад, SpeedTest) і трафік шейпер (наприклад network link Conditioner). Обмежуємо мережу параметрами 1 Мбіт/сек на upload і download і починаємо нарощувати втрату пакетів.

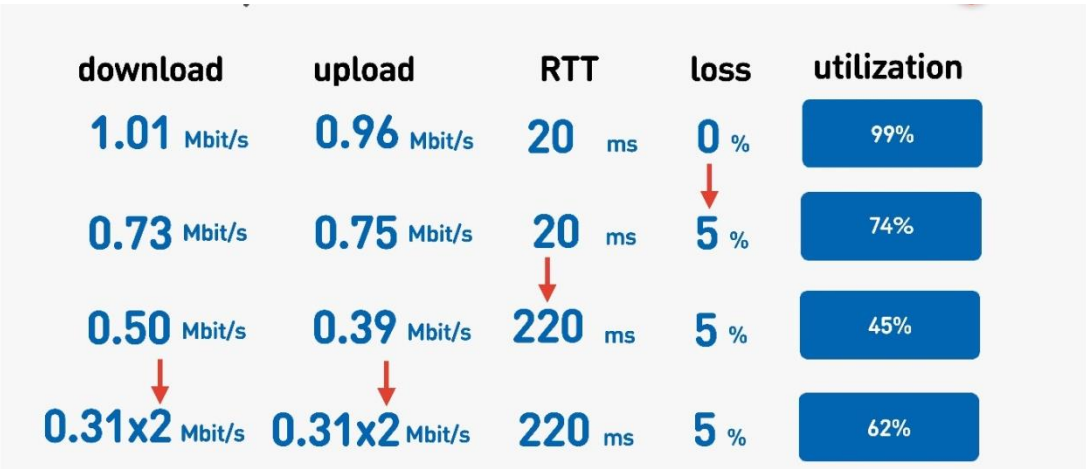


Рисунок 1.7. Тестування мережі побудованої на протоколі TCP

На рисунку вказані RTT і втрати(див. Рис.1.7). Видно, що в нашому випадку 0% втрат, мережа утилізована на 100%. Наступною дією збільшуємо packet loss на 5%, і бачимо, що мережа утилізується всього на 74%. Нібито нічого страшного - при packet loss в 5% втрачається 26% мережі. Але якщо збільшити ще й ping, то залишиться менше половини каналу.

Якщо канал з високим RTT і великим packet loss, то одне TCP з'єднання не повністю утилізує мережу. Подальший експеримент показує, що якщо почати використовувати паралельні TCP-з'єднання (можна просто запустити декілька Speed Test-ів одночасно), видно обернене зростання утилізації каналу.

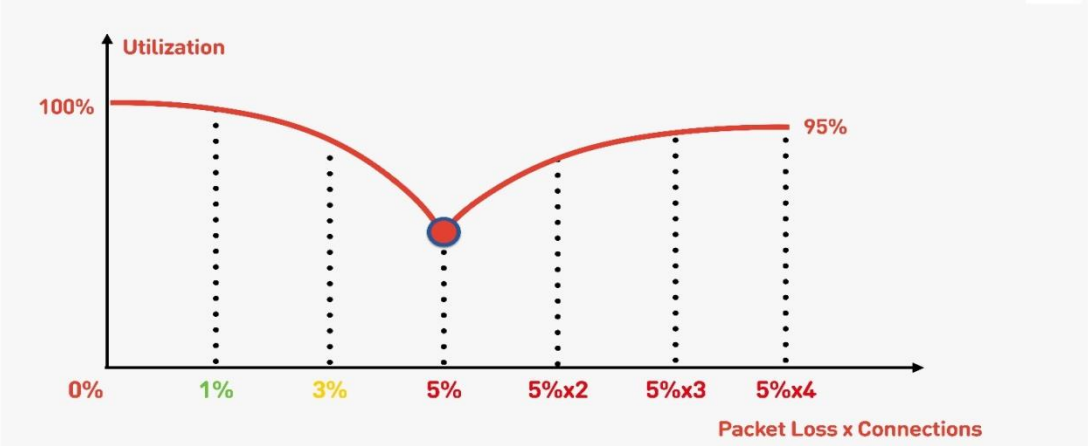


Рисунок 1.8. Крива неефективності протоколу TCP

Зі збільшенням числа паралельних TCP-з'єднань утилізація мережі стає майже рівною пропускній здатності(див. Рис.1.8), за вирахуванням відсотка втрат.

Таким чином, можна зробити наступні висновки:

1. Бездротові мобільні мережі мають більший відсоток утилізації каналу.
2. TCP не до кінця утилізує канал в нестабільних мережах.
3. Споживання контенту залежить від швидкості інтернету: чим вище швидкість інтернету, тим більше користувачі використовують трафіку.

1.5. Критерії порівняння протоколів транспортного рівня

Отже, яку мережу використовувати для перевірки протоколу.

Мережі бувають різні:

• З перевантаженнями, коли пакетів дуже багато і деякі з них втрачаються через перевантаження каналів або обладнання.

• Високошвидкісні з великими round-trip (наприклад коли сервер розташовується відносно далеко).

• Дивні - коли в мережі нібито нічого не відбувається, але пакети все одно пропадають, просто тому-що Wi-Fi точка доступу знаходиться за стінкою.

Профілі мережі завжди можна побачити(Рис 1.9): досить вибрати на своєму телефоні той чи інший профіль і запустити Speed Test.

Рисунок 1.9. Профілі мережі

Крім профілів мережі, потрібно ще визначитися з профілем споживання трафіку. Ось ті, які було використано:

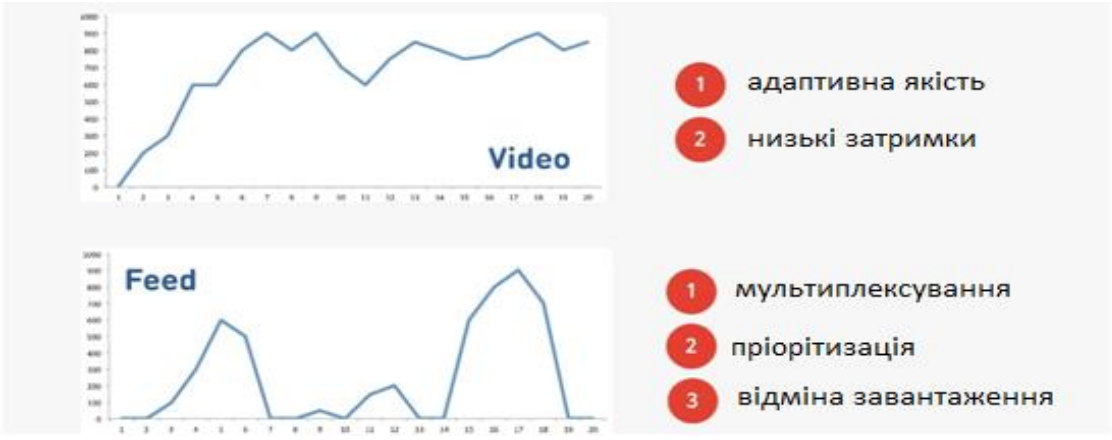


Рисунок 1.10. Використані профілі споживання трафіку.

Можна виділити наступні профілі(див. Рис. 1.10):

1. Профіль Відео, коли ви підключаєтеся до мережі і транслюєте той чи інший контент. Швидкість з'єднання збільшується, як на верхньому

графіку. Вимоги до цього протоколу: низькі затримки і адаптація бітрейта.

2. Варіант перегляду стрічки: імпульсне завантаження даних, фонові запити, проміжки простою. Вимоги до цього протоколу: ці дані мультиплексируются і пріорітізіруются, пріоритет користувацького контенту вище фонових процесів, є скасування завантаження.

Звичайно, порівнювати протоколи потрібно на найпопулярніших протоколах верхнього рівня. Відповідно до моделі OCI – це протоколи 7-го рівня, серед яких виділяють HTTP. В моделі TCP/IP – це також протоколи HTTP.

Стандартний стек 2000-х виглядав як HTTP 1.1 поверх SSL. Сучасний стек - це HTTP2.0, TLS 1.3, і все це поверх TCP(див. Рис. 1.11)[7].

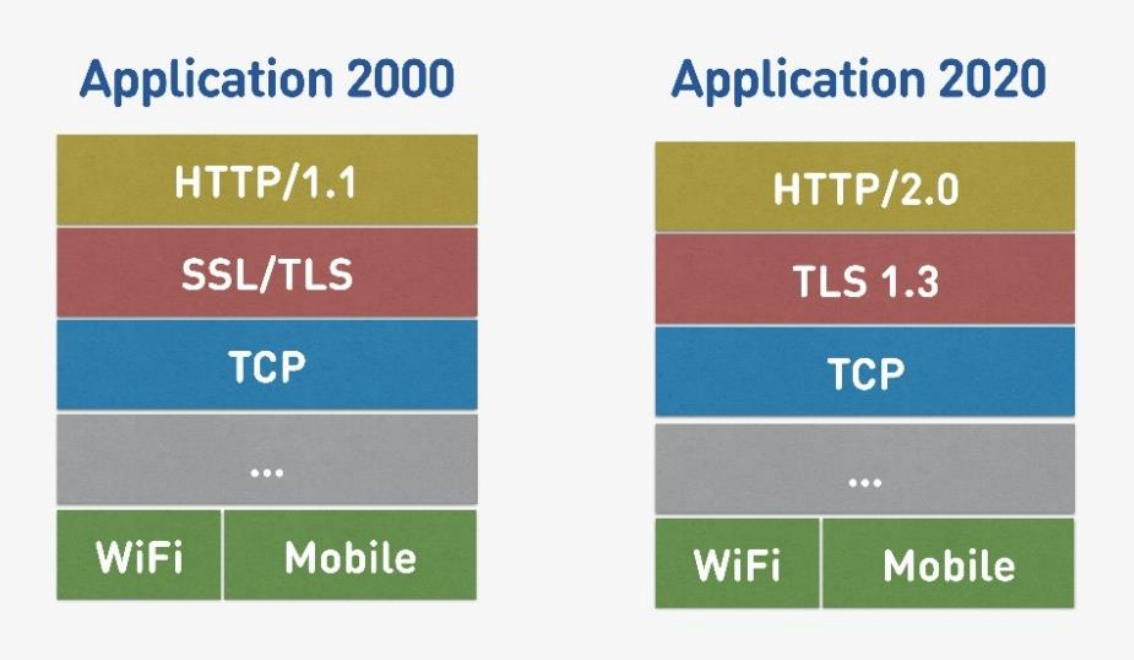


Рисунок 1.11. Порівняння стеків HTTP 1.1 і HTTP 2.0.

Основна відмінність в тому, що HTTP 1.1 використовує обмежений пул з'єднань в браузері до одного домену, який створює окремий домен для

картинок, для даних і так далі. HTTP 2.0 пропонує одне мультиплексироване з'єднання, в якому передаються всі ці дані(див. Рис. 1.12).

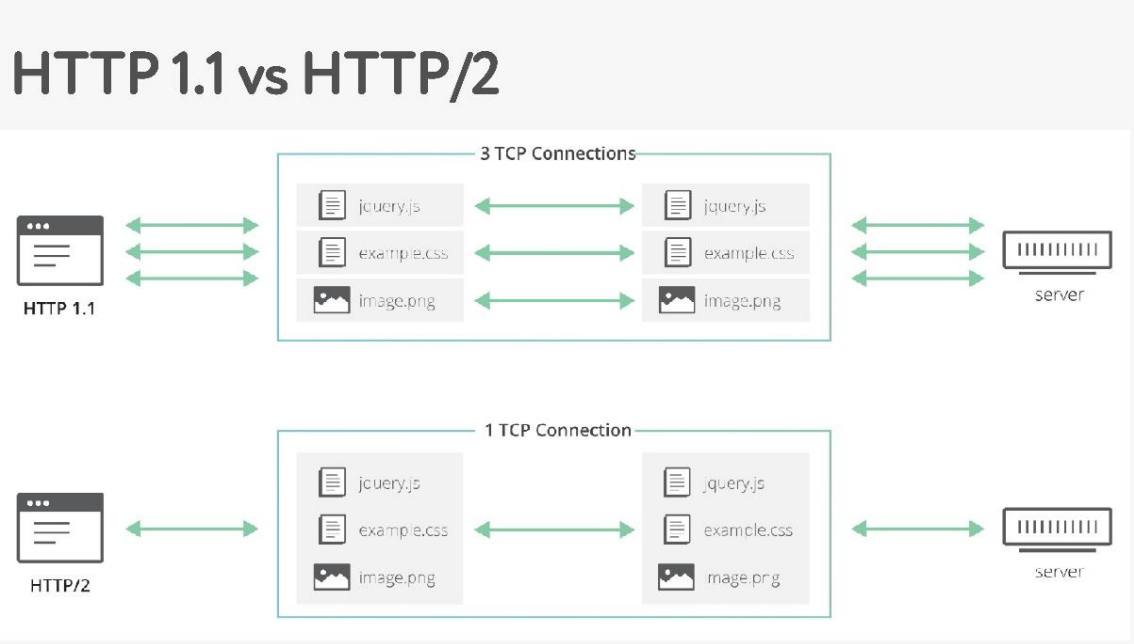


Рисунок 1.12. Порівняння HTTP 1.1 та HTTP 2.0

HTTP 1.1 в основі своєї роботи виконує послідовне з’єднання: робите запит, отримуєте дані, робите запит, отримуєте дані (див. Рис. 1.13).

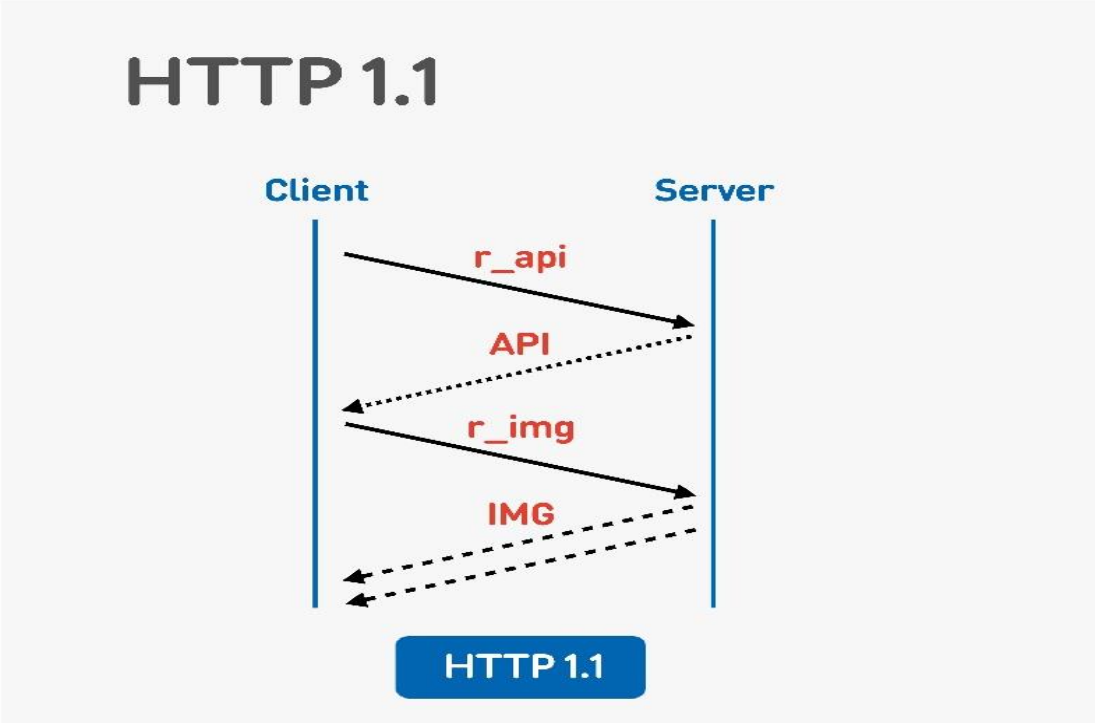


Рис. 1.13. Схема встановлення з’єднання з сервером протокола HTTP 1.1.

Зазвичай браузер або мобільний додаток встановлює з'єднання на отримання картинок, даних по API, і ви паралельно виконуєте запит за картинкою, за API, за відео і так далі. Основна проблема - конкуренція. Ви ніяк не керуєте відправленими запитами. Ви розумієте, що користувачеві вже не потрібна картинка, яку він перегорнув, але нічого не можете зробити.

З HTTP 1.1 ви все одно отримуєте те, що запросили, скасувати завантаження важко. Єдиний шанс - socketclose - це закрити з'єднання.

HTTP 2.0 вирішує ці проблеми:

• бінарний, стиснення заголовків;

• мультиплексування даних;

• пріоритизація;

• можлива відміна завантаження;

• server push

Розглянемо більш детально важливі моменти.

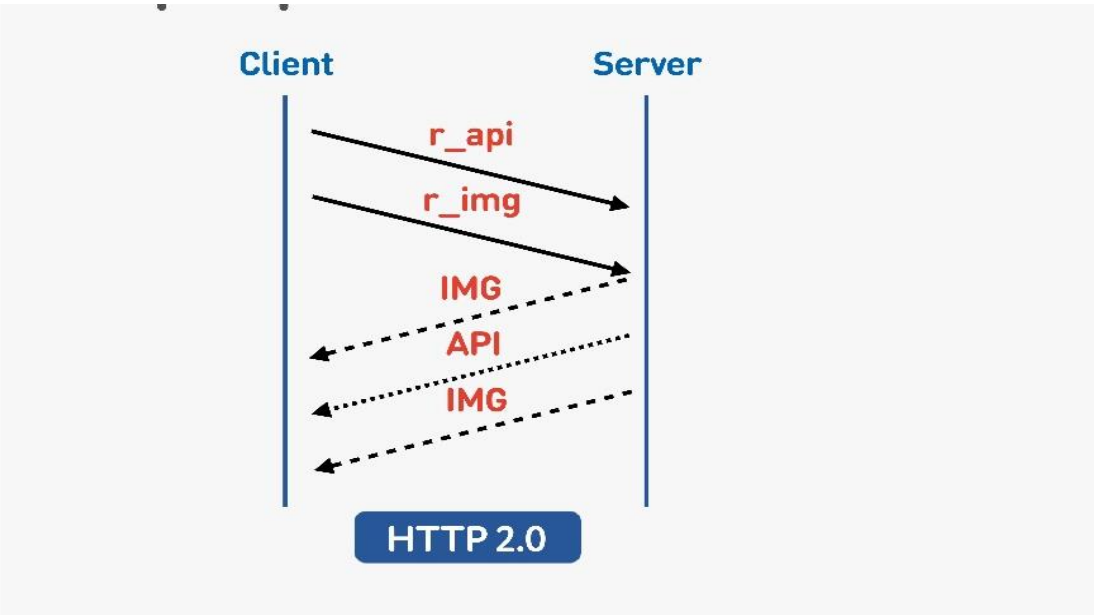


Рисунок 1.14. Принцип роботи протокола HTTP 2.0

Робимо запит на картинку і API. Картинка відразу надсилається, API підготувався через деякий час. Надіслався API - надіслалась до кінця картинка(Рис 1.14). Все це відбувається прозоро. Високопріоритетний контент завантажується раніше.

Server push - це коли ви зробили запит на щось конкретне типу API, але ще в добавок на клієнті закешувались зображення, які точно знадобляться для перегляду, наприклад, стрічки.

Ще є команда Reset stream, яку браузер виконує сам, якщо перейти між сторінками. Для мобільного клієнта з її допомогою можна відмовитися від отримання даних, при цьому не розриваючи з'єднання**.**

1.6 Висновки до розділу 1.

В цьому розділі розглянуто модель OSI та порівняно протоколи транспортного рівня. Описано переваги та недоліки таких протоколів як TCP та UDP, а також порівняно пакет TCP з датаграмою UDP. Розкрито проблему вибору протоколу транспортного рівня в залежності від потреб користувачів.

Було визначено, що протокол TCP не дуже ефективний в бездротових мережах, саме через особливості фізичного рівня – бездротового та визначено, що UDP краще використовувати при передачі даних для котрих важливі затримки у часі. А також розглянуто роботу протоколу 7 рівня HTTP 1.0 та HTTP2.0 моделі OSI поверх протоколу транспортного рівня TCP.

РОЗДІЛ 2. ВИБІР ПАРАМЕТРІВ ДЛЯ МОДИФІКАЦІЇ ПРОТОКОЛУ UDP

2.1 Параметри моделі без втрат при використанні UDP

Можливі два варіінти розвитку протоколу транспортного рівня.

Перший варіант - на рівні IP є TCP і UDP. Очевидно, що якщо паралельно з TCP і UDP запустити свій протокол, то про нього не знатимуть Firewall, Brandmauer, маршрутизатори і всі інші пристрої, які беруть участь в доставці пакетів. У підсумку доведеться роками чекати, коли все обладнання оновиться і почне працювати з новим протоколом.

Другий варіант - зробити свій надійний протокол доставки даних поверх ненадійного UDP. Очевидно, що чекати, поки Linux, Android і iOS додадуть новий протокол до свого ядра можна чекати довго, тому треба робити протокол власноруч.

Таке рішення є цільовим та виступає предметом дослідження роботи, яке можна називати self-made UDP-протокол. Щоб почати його розробляти, не потрібно нічого особливого: просто відкриваємо UDP socket і відправляємо дані(Рис 2.1).

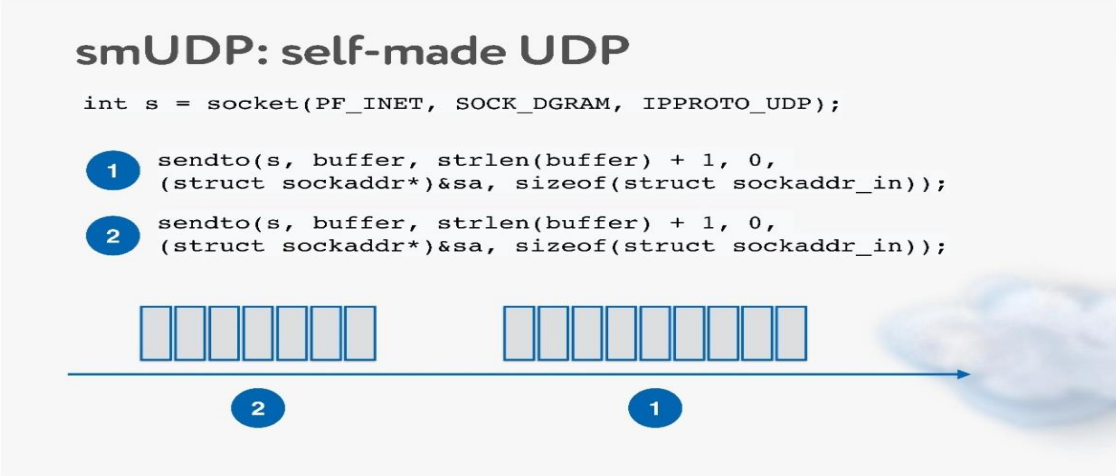


Рисунок 2.1. Приклад відкритого UDP сокета

Основною моделлю роботи мережі є модель без втрат**,** дана умова буде виступати головним обмеженням.

Почнемо порівняння з простою мережею, в якій існує тільки два параметри (критерії порівняння): **round-triptime і bandwidth**. RTT - це пінг, час обороту пакету, тобто час отримання підтвердження на свій запит.

Щоб виміряти bandwidth - пропускну здатність мережі - відправляємо пачку пакетів і рахуємо кількість минулих пакетів на якомусь тимчасовому інтервалі.

Дослідникам відомо, що чим менше ping, тим краще. Цей показник вимірюється в мілісекундах (мс) і позначає час, за який один комп'ютер або сервер отримує інформацію від іншого пристрою.

Ping перевіряє, чи здатний сервер, до якого направлено запит, взагалі його обробити. Якщо з'єднання може бути встановлено, то вимірюється швидкість обміну даними. Ідеальна затримка ping менше 40 мс. Нормальне або терпиме значення - від 40 і до 110 мс.

Поки що не придумали іншого більш швидкого способу перевірки працездатності серверів. Як відомо, Інтернет являє собою мережу з комп'ютерів і серверів, деякі з них обробляють величезну кількість інформації. Іноді запитів від користувачів до конкретного сервера настільки багато, що він не встигає реагувати. Якщо той чи інший популярний сайт «зберігається» на проблемному сервері, то всі користувачі будуть мати труднощі з його завантаженням. Ping в таких випадках буде досягати критичних 200 мс і більше, як було раніше в мережі 2G.

Таким чином, нормальний пінг - це підтвердження того, що конкретний елемент мережі працює справно. Значення цього показника залежить як від

якості покриття вашого оператора і навантаження на його мережу, так і від самих сайтів, до яких ви звертаєтеся.

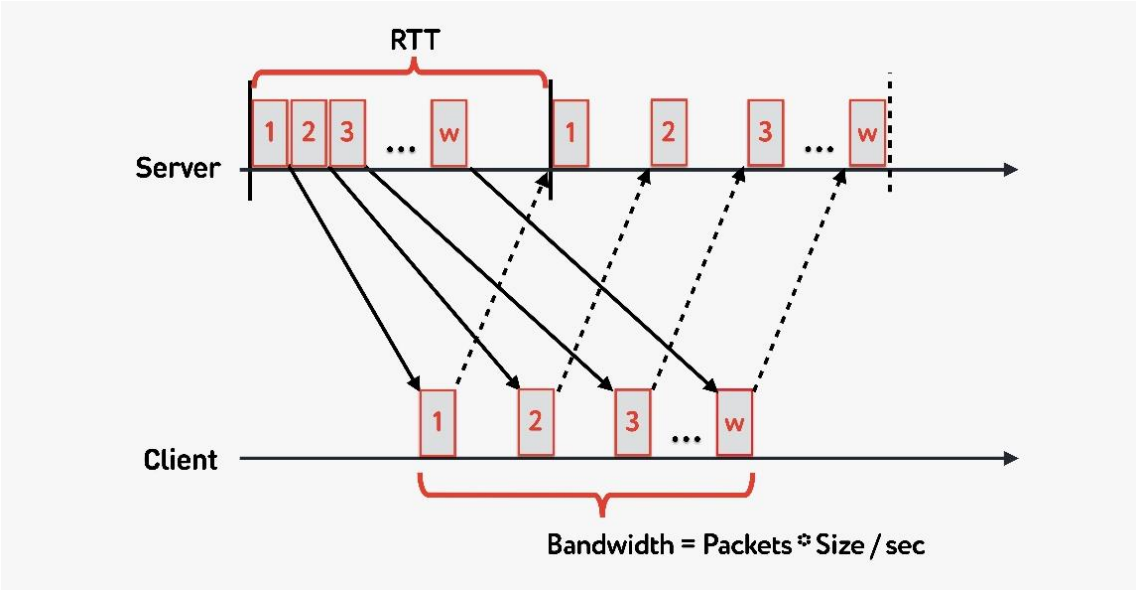


Рисунок 2.2. Робота надійного протоколу в простій мережі

Так як ми працюємо з надійними протоколами, то, звичайно, є acknowledgement - відправляємо пакети і отримуємо підтвердження про отримання.

2.2. Рішення задачі про низьку швидкість передачі в мережі

Якщо ви проживаєте в Україні, але захочете подивитися улюблений серіал в FullHD якості, наприклад, з Канади. То навіть при ідеальному Wi-Fi 400 Мбіт / с ви будете мати RTT приблизно в 250 мс.

Як ви вважаєте, чи зможете подивитися відео? Відповідь залежить від настройки send / recv buffer на серверах.

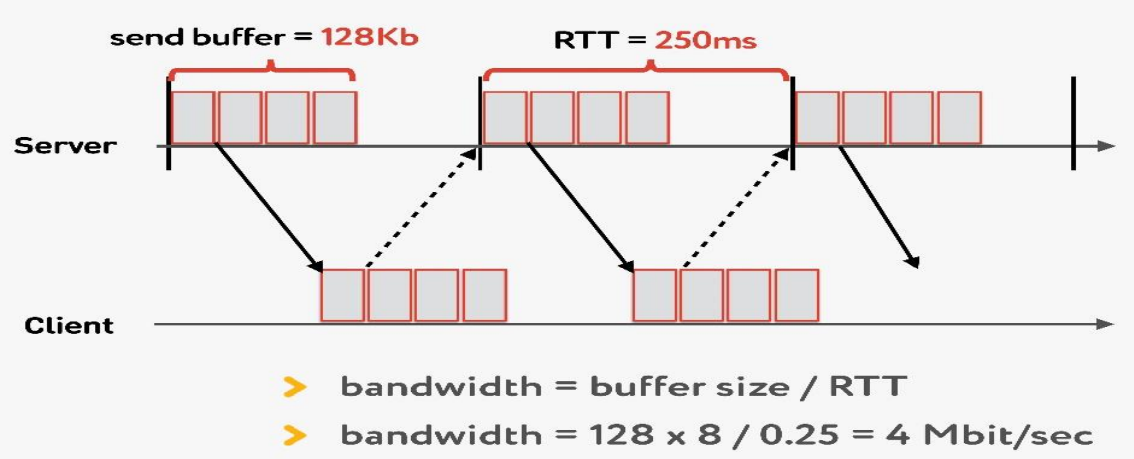


Рисунок 2.3. Значення send / recv buffer на сервері та клієнті

Так як у нас протокол з acknowledgement, то всі дані, які не отримали підтвердження про доставку, зберігаються в буфері. Якщо send buffer обмежений 128 Кб, то ці 128 Кб менше, ніж за час RTT, ми відправити не можемо. Таким чином, від нашої мережі в 400 Мбіт / с залишилося 4 Мбіт / с. Цього недостатньо, щоб онлайн дивитися відео в FullHD.

Тоді змінивши розмір буфера можна побачити, як дійсно змінюється швидкість віддачі одного сегмента відео в залежності від зміни розміру буфера. Recv buffer підлаштовувався автоматично, тобто те, що відправляв сервер, клієнт завжди міг прийняти.

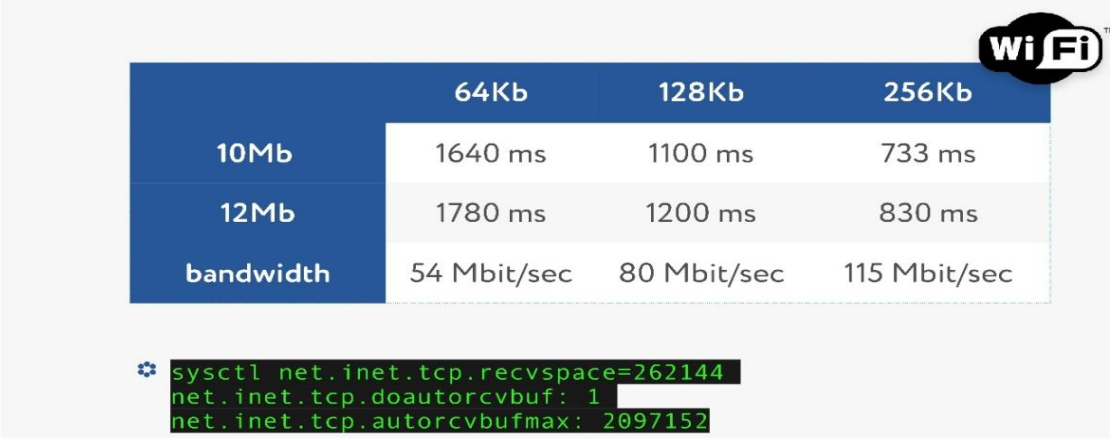


Рисунок 2.4 Приклад збільшення розміру буфера

За результатами досліджень протоколу TCP, очевидно, що якщо передаєте дані на великі відстані по високошвидкісній мережі, потрібно збільшити буфер відправки(див. Рис. 2.4).

Для того щоб розібратися з оптимальним розміром буфера, потрібно зрозуміти, що таке On-the-flypackets (див. Рис.2.5).

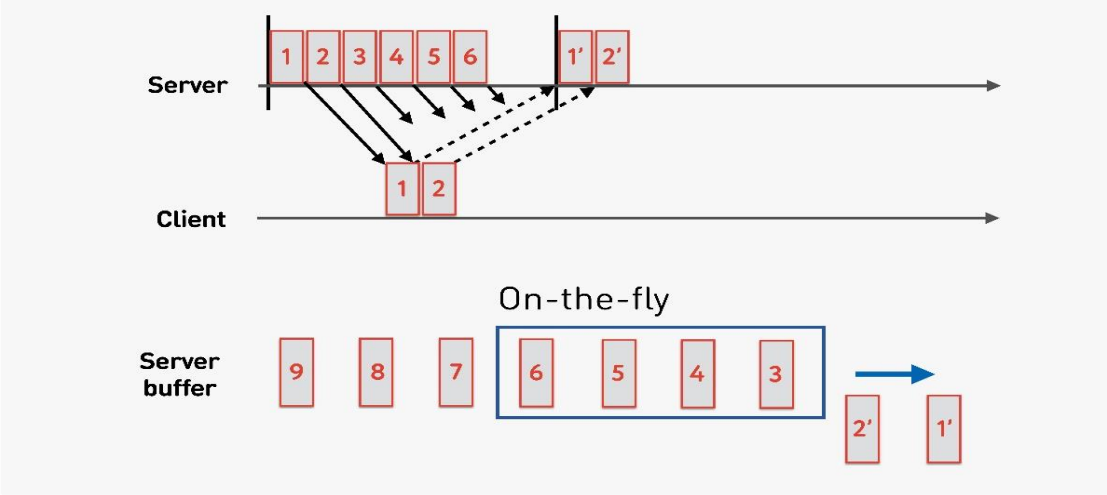


Рисунок2.5. On-the-fly packets в мережі

Відповідно в мережі існують наступні стани:

- пакети 1 і 2 вже відправлені, для них отримано підтвердження;
- пакети 3, 4, 5, 6 відправлені, але результат доставки невідомий (on-the-fly packets)(див. Рис. 2.5);
- інші пакети знаходяться в черзі.

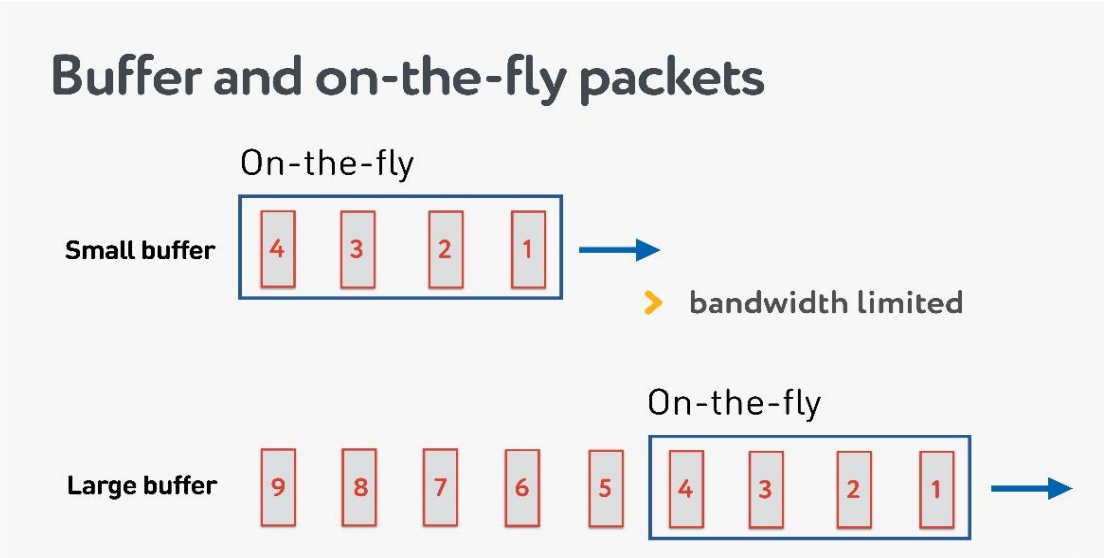


Рис.2.6. Розмір буферу в залежності від кількості on-the-fly пакетів

Якщо кількість пакетів on-the-fly дорівнює розміру буфера, то він недостатнього розміру(див .Рис. 2.6).У цьому випадку мережа використовується не повністю – рівень утилізації нижчий від 80%. Можлива зворотна ситуація - занадто великий розмір буфера.

З точки зору мультиплексування даних і якщо відправити кілька запитів одночасно, наприклад, картинки в це ж з'єднання та API, то коли вся величезна мегабайтна картинка помістилась в буфер, а ми намагаємося передати ще й високопріоритетний API, то буфер переповнюється.

Простим рішенням є автоматична налаштування розміру буфера. Зараз це використовується на багатьох клієнтах і працює приблизно так.

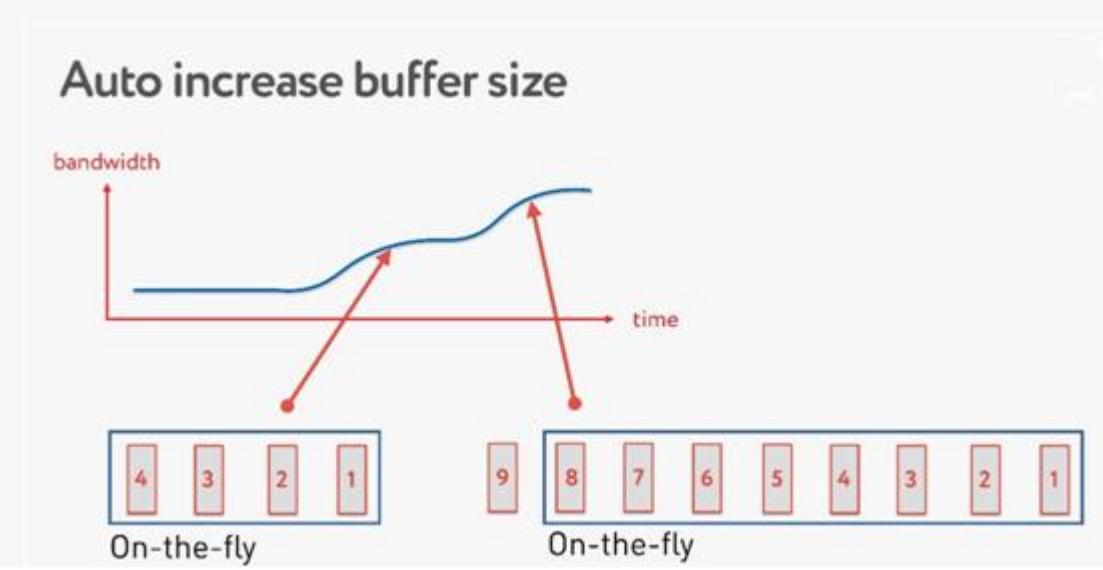


Рис.2.7. Автоматичне збільшення буферу в залежності від кількості on-the-fly пакетів

Якщо зараз може бути відправлено багато пакетів, буфер збільшується, передача даних прискорюється, розмір буфера зростає (див. Рис. 2.7).

При такому підході існує проблема:якщо буфер збільшився, його не можна так просто зменшити. Це більш складна задача. Якщо швидкість просідає, то відбувається те саме розпухання буфера. Буфер досить великий і весь заповнений, нам потрібно чекати, поки всі дані відправляться на клієнт.

При реалізації свого UDP-протоколу, це все дуже просто - у нас є доступ до буферу, відповідно можливо його змінювати. Залишається питання в оптимальному налаштуванню параметрів розміру буфера.

Рис.2.8. Порівняння буферу UDP та TCP

Якщо TCP в таких ситуаціях просто додає дані в кінець, і нічого з цим не можна зробити, то в self-made протоколі можна поміщати дані, наприклад, вперед, відразу ж за On-the-fly packets(див. Рис. 2.8.).

А якщо прийде cancel, і клієнт скаже, що ця картинка більше не потрібна, йому потрібні API дані, він перегорнув контент далі, можна все це викинути з буфера і відправити потрібне.

Відомо, що щоб відновлювати пакети, управляти доставкою, отримувати acknowledgements, потрібен якийсь sequence_id пакетів. Sequence_id ми виписується тільки для on-the-fly packets, тобто видаємо його тільки тоді, коли відправляємо пакети. Все інше в буфері можна пересувати як хочемо до тих пір, поки пакети не відправились[1].

Висновок: в TCP буфер треба правильно налаштувати, знайти баланс, щоб не впиратися в мережу і незробити буфер занадто великим. Для власного UDP-протоколу все просто – можна використовувати автоматичне налаштування буферу.

2.3. Рішення задачі в моделі мережі з втратами

Пересуваємося на рівень вище, мережа стає трохи складніше, в ній з'являється packetloss. Для мобільних мереж це звичайна ситуація. Частина з відправлених пакетів не доходить до клієнта. Стандартний алгоритм відновлення retransmit працює приблизно так:

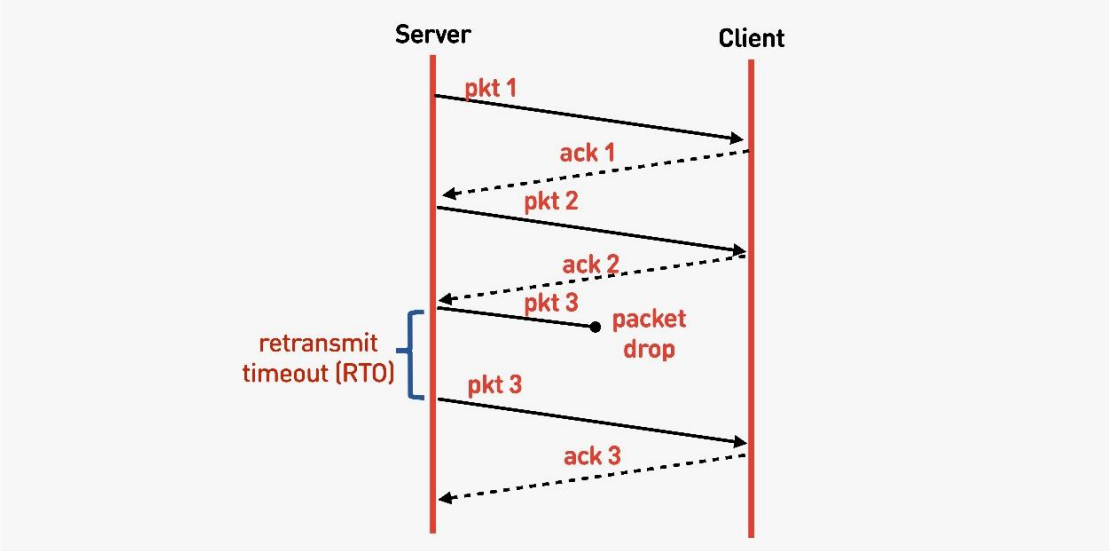


Рис.2.9. Алгоритм пересилання пакетів

Відправляємо пакети, на кожен пакет отримуємо підтвердження (acknowledgement)(див. Рис. 2.9.). Якщо через Retransmit time out (RTO) рівному RTT плюс деякі константи підтвердження немає, то перевідправляємо пакет.

Повернемося до кривої неефективності TCP, коли втрачається всього 5% пакетів, а утилізація мережі дорівнює 50%(Рис 2.10).

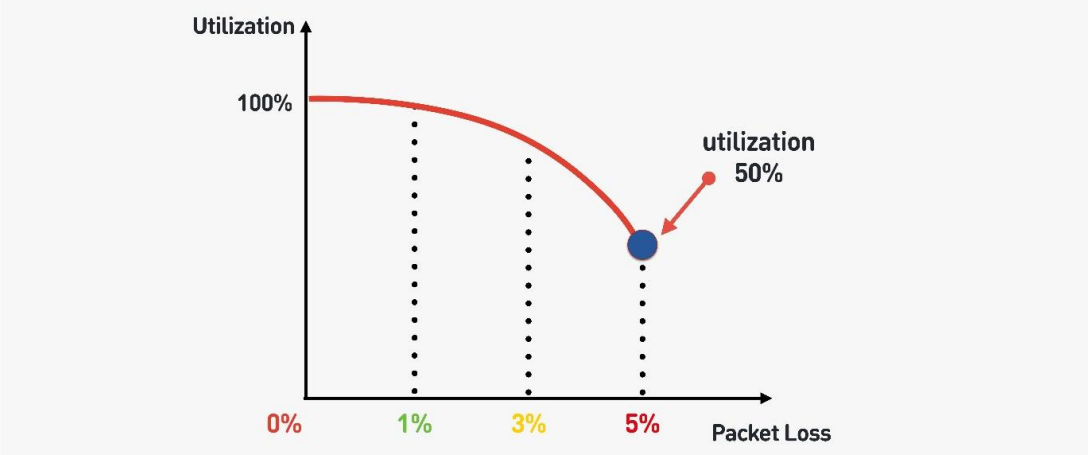


Рис.2.10. Утилізація мережі в залежності від кількості втрачених пакетів

При retransmit, який просто досилає пакети, ми не повинні спостерігати таку проблему. Щоб розібратися в причинах, потрібно зрозуміти, що таке Congestion control.

Розглянемо особливості flowcontrol та congestion control:

• **Flow control** - це певний механізм захисту від перевантаження. Одержувач повідомляє, на скільки даних у нього є місце в буфері, щоб він був готовий їх прийняти. Якщо передати понад flow control або recv window, то ці пакети просто будуть викинуті.

• У **congestion control** абсолютно інше завдання. Механізми схожі, але завдання - врятувати мережу від перевантаження.

Якщо перевантажити мережу, то цілком імовірна така ситуація: посилаєте дані, частина пакетів не доходить, посилаєте ще більше даних, і всі ці дані знову пропадають. За те щоб лімітувати видачу даних деякими порціями, як раз і відповідає congestion control.

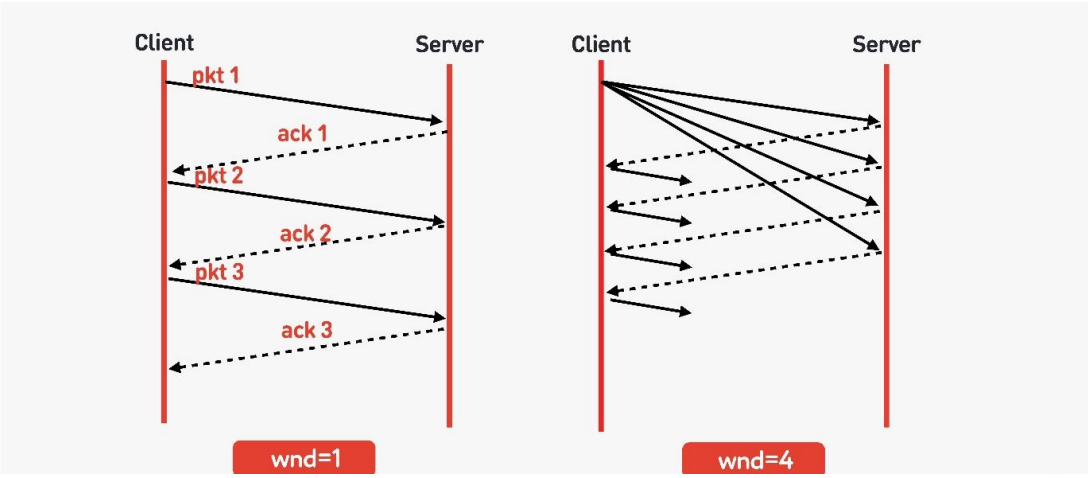


Рис. 2.11. Спосіб відправлення пакетів в залежності від розміру TCP window

Відповідно наведемо два приклади (див. Рис. 2.11.):

• Якщо TCP window = 1, то дані передаються як на схемі зліва: чекаємо acknowledgement, відправляємо наступний пакет і т.д.

• Якщо TCP window = 4, то відправляємо відразу пачку з чотирьох пакетів, чекаємо acknowledgement і далі працюємо.

Коли з'єднання тільки стартує, розмір вікна поступово збільшується. Розмір initial window в TCP = 10.

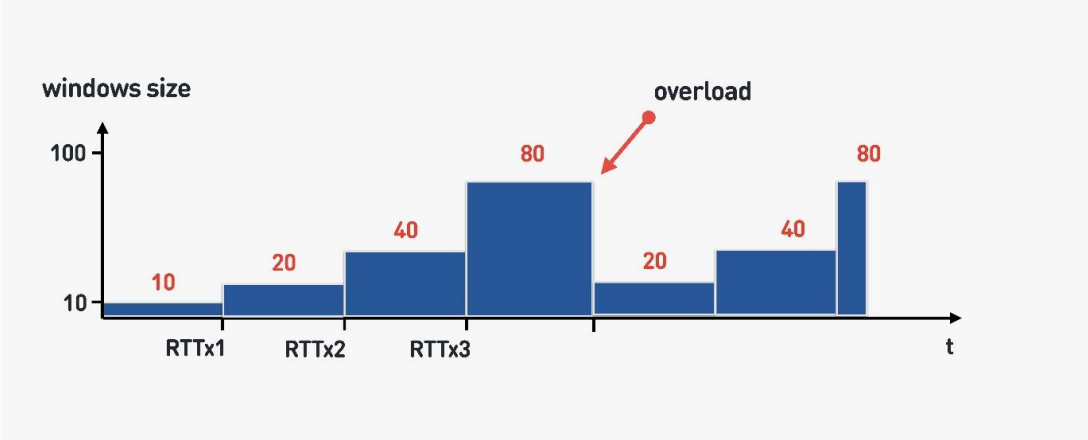


Рис. 2.12. Зменшення TCP window в залежності від навантаження на мережу

Якщо відбувається перевантаження мережі, пропадають пакети, то вікно назад звужується і починає розганятися заново(див. Рис 2.12.).

При цьому мережа має наступний вигляд(див. Рис 2.13):

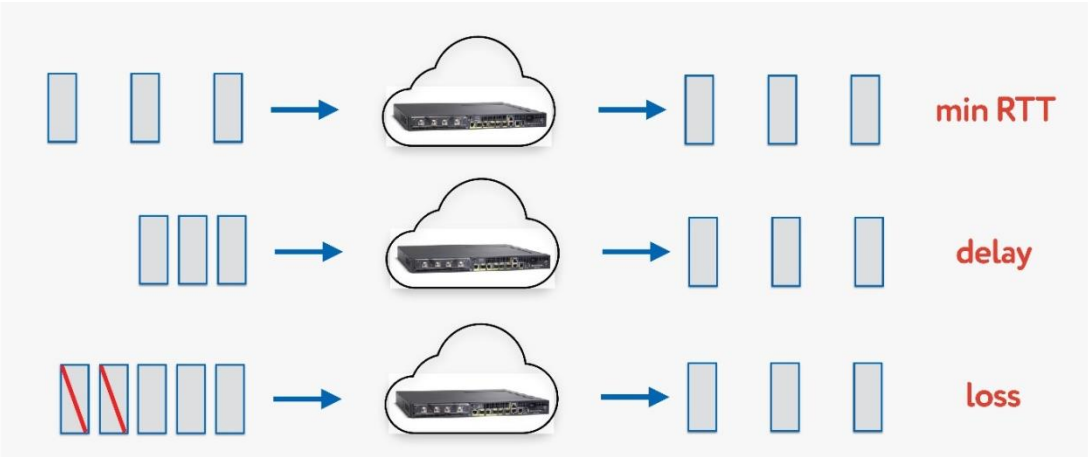


Рис.2.13. Поведінка мережі при різному навантаженні

• На верхній схемі мережу, в якій все добре. Пакети відправляються із заданою частотою, з такою ж частотою повертаються підтвердження.

• У другого рядка починається перевантаження мережі: пакети йдуть частіше, acknowledgements приходять з затримкою.

• Дані накопичуються в буферах на маршрутизаторах і інших пристроях і в якийсь момент починають пропускати пакети, acknowledgements на ці пакети не приходять (нижня схема).

З точки зору маршрутизатора це виглядає так як на Рис 2.14.

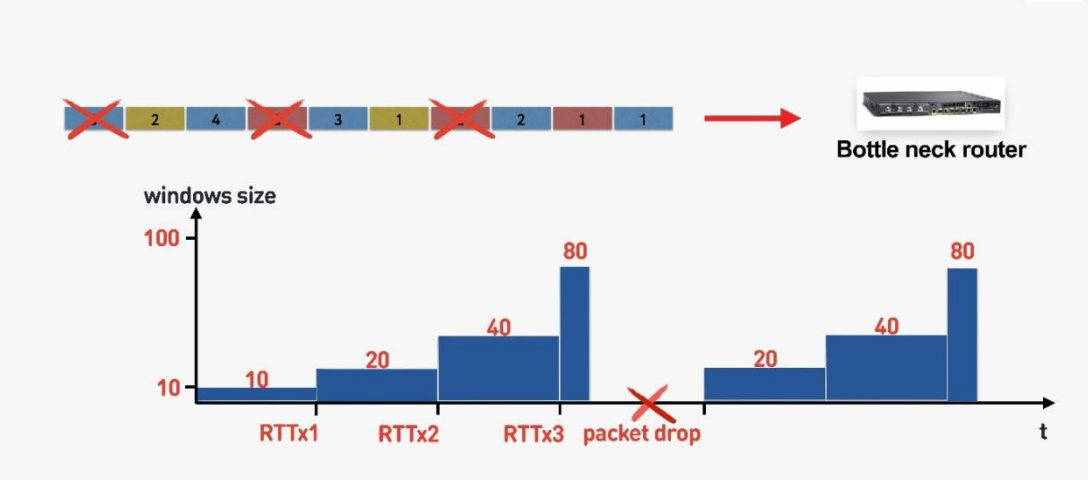


Рис.2.14. Модель втрати пакетів на роутері при навантаженій мережі

Маршрутизатор не чекає перевантаження, і відразу дропає(відкидає). У маршрутизатора є механізм тікетів: він видає тікет на відправку, якщо канал звільниться і т.д. Суть механізму в тому, що він дропає пакети трохи раніше (Рис2.14). Тоді спрацьовує congestion control, зменшується розмір TCP window, навантаження на маршрутизатор падає, і все продовжує працювати.

Зрозуміло, що TCP розвивався, адаптувався, і перший congestion control оперував тільки loss-функцією. Після цього з'явилися congestion control на loss delay, тобто і на втрати, і на затримки.

Розглянемо:

• Cubic – стандартний Congestion Control з Linux 2.6. Саме він використовується найчастіше і працює примітивно: втратив пакет - зменшив вікно.

• BBR - більш складний Congestion Control, який придумали в Google в 2016 році. Враховує розмір буфера.

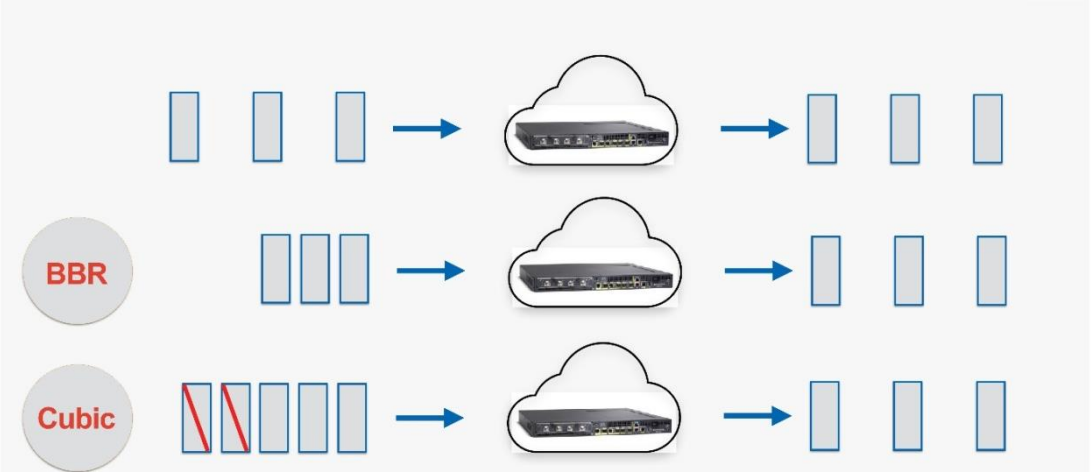


Рис.2.15 Мережі з використанням таких Congestion Control як Cubic і BBR

На схемі зверху нормальний маршрутизатор і маршрутизатор, у якого чергу починає збиратися - кожен наступний acknowledgement приходить все довше і довше щодо відправки. В цьому випадку:

• BBR розуміє, що йде переповнення буфера, і намагається зменшити вікно, зменшити навантаження на маршрутизатор.

• Cubic чекає втрати пакета і після цього змінює вікно.

Нижче графік залежності затримки від часу з'єднання(Рис 2.16.), з якого видно, що відбувається на різних Congestion Control.

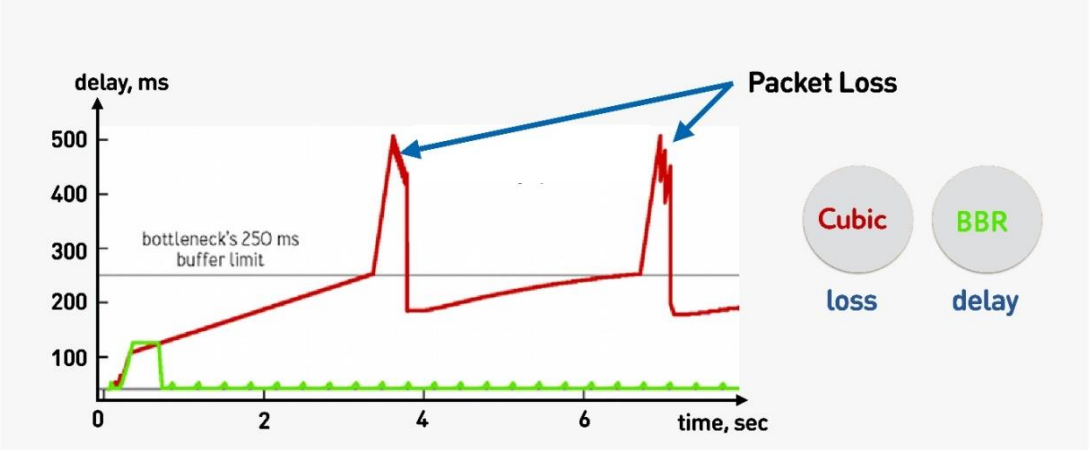


Рис.2.16 Графік залежності затримки від часу з’єднання при різних Congestion Control

BBR спочатку дивиться на час round-trip, відправляє більше і більше пакетів, потім розуміє, що буфер забивається, і виходить на режим роботи з мінімальною затримкою.

Cubic працює агресивно - він переповнює цілком буфер, і, коли буфер переповнюється і трапляється packet loss, то cubic зменшує вікно.

Здається, що за допомогою BBR можна було б вирішити всі проблеми, але в мережах існує jitter - пакети іноді затримуються, іноді групуються пачками. Ви їх відправляєте з певною частотою, а вони приходять групами. Ще гірше, коли ви отримуєте acknowledgements назад на ці пакети, і вони теж затримуються.

Для того щоб все побачити, то пінгуєм, наприклад, сайт HighLoad ++, дивимося ping і рахуємо jitter між пакетами.

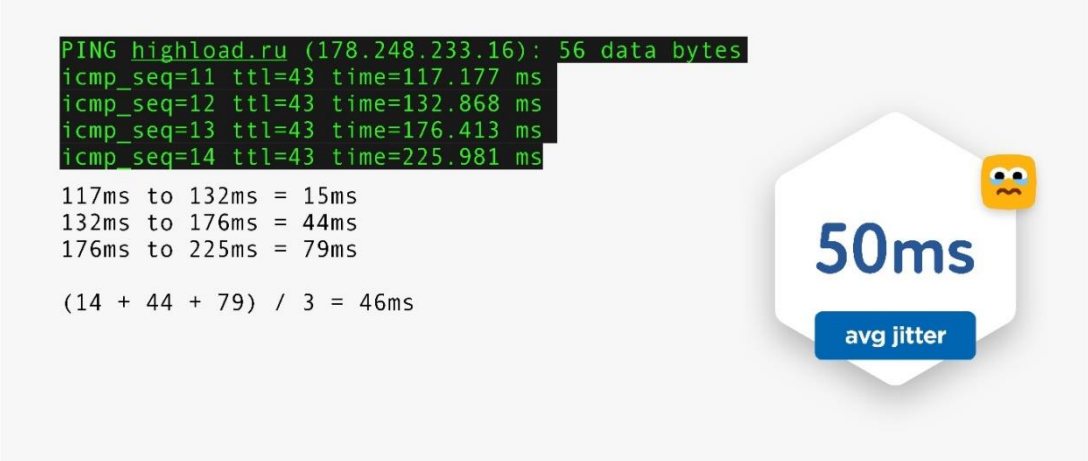


Рис.2.17 Вимірювання параметру jitter в мережі

Видно, що пакети приходять нерівномірно, середній jitter близько 50 мс. Звичайно, BBR може при цьому помилитися(Рис 2.17.).

BBR хороший тим, що розрізняє: реальний congestion loss, від втрати пакетів через переповнення буферів пристроїв, і random loss через погану бездротову мережу. Але погано працює в разі високого jitter.

2.4. Підходи для покращення Congestion control

Насправді у TCP в acknowledgement досить мало інформації, в ній є тільки те, які пакети він бачив. Є ще selective acknowledgement, в якому говориться, які пакети підтверджені, а які ще не дійшли. Але і цієї інформації недостатньо.

Приклад того, як можна за допомогою acknowledgement поділити jitter на відправку і jitter на прийом(Рис 2.17), і відслідковувати їх окремо. Тоді ми станемо більш гнучкими, і розумітимемо, коли стався congestion loss, а коли random loss. Наприклад, можна зрозуміти, скільки jitter в кожну сторону, і більш точно налаштувати вікно.

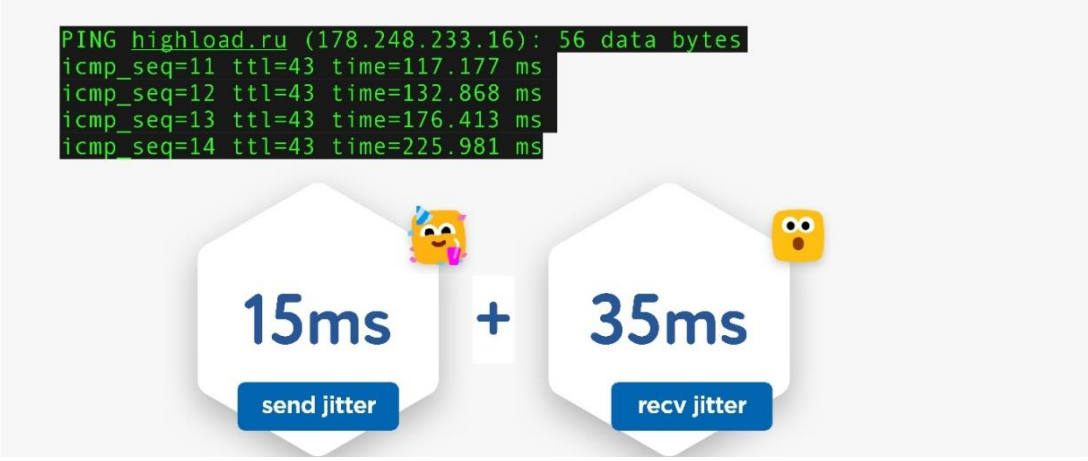


Рис.2.18. Вимірювання параметру jitter при відправці та прийомі даних в несиметричній мережі

Залишається питання, який Congestion control вибрати. BBR завжди ефективний для відео, тому що зменшує затримки. В інших випадках зазвичай використовується Cubic - він хороший для фотографій(Рис 2.19).

Algorithm	Network	AppProfile	Server
westwood+	high latency networks	speed	API
cubic	high bandwith/long RTT	stability	Photo
BBR	high random loss	streaming	Video

Рис.2.19. Переваги та недоліки різних Congestion Control

Є десятки різних варіантів congestion control. Для того щоб вибрати кращий, можна зібрати статистику по клієнту і для різного типу профілю навантаження спробувати той чи інший congestion control.

Вдалося серйозно збільшити глибину перегляду. Google повідомляє, що у них приблизно на 10% зменшується кількість буферизації в плеєрі при використанні BBR [8].

	Default	Check
Linux	Cubic	ipv4.tcp_congestion_control
Android	Cubic	kernel auditor
iOS	Cubic	но это не точно
macOS	Cubic	net.inet.tcp.use_newreno

Рис.2.20. Тип congestion control наклієнтах.

Висновки про congestion control:

• Для відео завжди хороший BBR.

• В інших випадках, якщо ми використовуємо модифікований UDP-протокол, то можна додати congestion control до процесу обробки.

• З точки зору TCP можна використовувати тільки congestion control, який є в ядрі. Якщо хочете реалізувати свій congestion control в ядро, потрібно обов'язково відповідати специфікації TCP. Неможливо роздути acknowledgement, зробити зміни, тому що просто їх немає на клієнті.

При розробці свого UDP-протоколу, набагато більше свободи з точки зору congestion control.

2.5. Мультиплексування і пріоритизація

Є кілька запитів, які мультиплексируются через одне TCP-з'єднання. Ми їх відправили в мережу, але якийсь пакет пропав. TCP-з'єднання буде цей пакет пересилати, він перешлеться за час, близький до RTT або більше. У цей час ми нічого отримати не зможемо, хоча в TCP-буфері знаходяться дані від іншого запиту, повністю готові до того, щоб їх можна було забрати.

Виходить, що мультиплексування поверх TCP, якщо ви використовуєте HTTP 2.0, не завжди ефективно в поганих мережах.

Наступна проблема - це розпухання буфера.

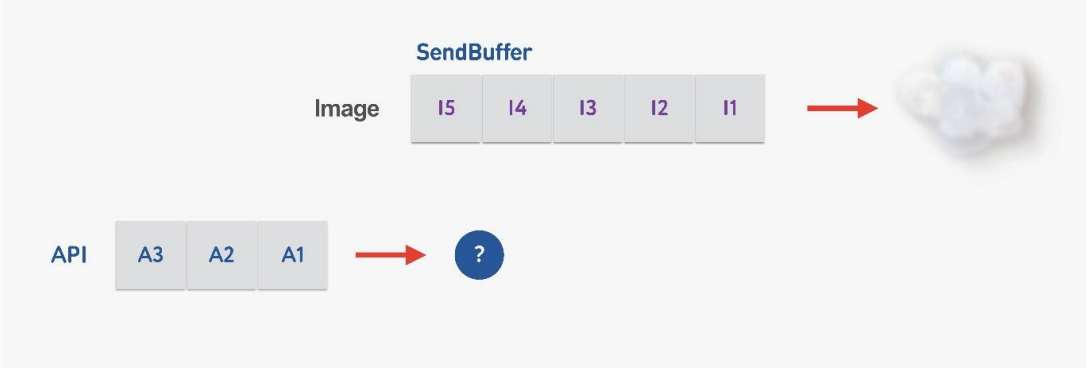


Рис.2.21. Збільшення буферу та відсутність пріорітизації

Коли картинка відправляється клієнту, збільшується буфер. Ми його довго відправляємо, а потім з'являється API-запит, і він ніяк не може бути пріоритезованим. У таких випадках не працює TCP-пріоритизація.(Рис 2.21)

Отже, якщо трапляється втрата пакетів, є head-of-Line blocking, а коли у клієнта змінний бітрейт (а у мобільних клієнтів це буває часто), то з'являється ефект bufferbloat. В результаті не працює ні мультиплексування, ні пріоритизація, ні server push, ні все інше, тому що у нас або забиті буфери, або клієнт щось очікує.

При модифікації UDP протоколу необхідно реалізувати своє мультиплексування, то можемо помістити туди різні дані. Це неважко, просто складаємо в буфер пакети з номерами. On-the-fly - те, що вже було відправлено, не чіпаємо, а те, що ще не відправлено, можна переставляти (див Рис. 2.22).

Алгоритм буде наступним:

1. Відправили дані.
2. Розбили на пакети
3. При отриманні пріоритетного API-запит:
    3.1. Вставили та відправили. Навіть якщо пропав пакет, ми з буфера можемо дістати готовий API-запит, він високопріоритетний і швидко дістанеться до клієнта. В TCP по визначенню при стрімінговій передачі даних таке неможливо.

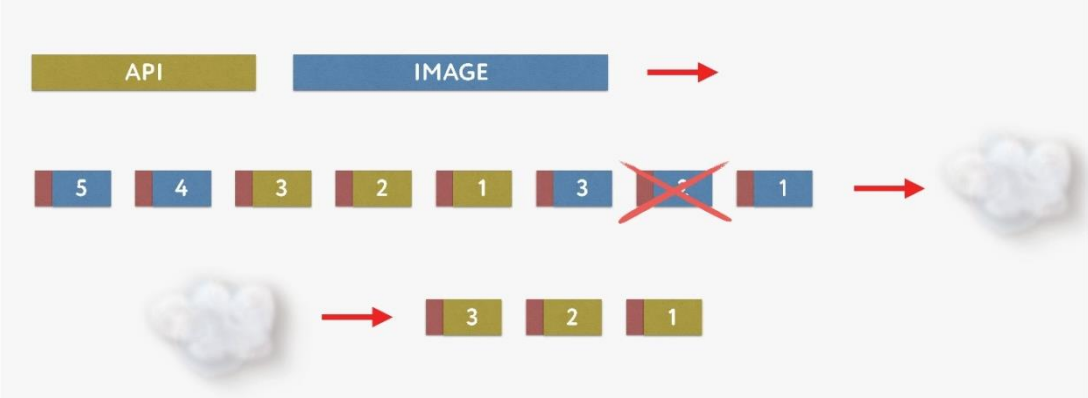


Рис.2.22 Можливість перестановки пакетів місцями в буфері протокола UDP

Також одна з важливих речей при використанні протоколу UDP є packet pacing. Це дуже важливо, якщо ви робите свій UDP-протокол. Іншими словами, чим довше ви посилаєте пакети в мережу, тим більше вірогідність втрати пакетів. Якщо пакети надсилати з певним інтервалом часу, то втрата пакетів зменшиться.

Отже, при умові задання рівномірного темпу відправки пакетів, то швидше за все, імовірність переповнення буферів стане нижче.

Також, при модифікації протоколу UDP, важливо звернути увагу на параметр MTU.

MTU – розмір даних котрі можна переправити по мережі.

Відправляючи пакети з сервера на клієнт, наприклад, розміром 1500 біт. Якщо на шляху зустрічається маршрутизатор, котрий не підтримує цей розмір MTU, то він фрагментує ці пакети. Отже, важливо правильно обрати MTU оскільки при фрагментації якщо губиться один пакет то доведеться відкинути і інший що може значно підвищити втрату пакетів в мережі.

Таким чином, виконавши модифікацію параметрів описаних вище, можна значно підвищити швидкість роботи мережі та при цьому знизити втрату пакетів.

1.6 Висновки до розділу 2

В цьому розділі було розглянуто недоліки протоколу TCP в мережах з втратами. Визначено, що протокол TCP не може відправляти нові дані по мережі до тих пір, поки не отримає підтвердження на старі, що значно знижує утилізацію мережі. Також було розглянуто принципи роботи таких congestion control як Cubic та BBR. З чого зроблено висновок, що BBR краще використовувати при передачі відео, а Cubic при передачі зображень.

Особливо слід зазначити, що було розглянуто параметри котрі впливають на швидкість передачі даних по мережі. Визначено, як впливає буфер відправки на передачу даних та встановлено, що параметр packet pacing впливає на втрати пакетів в мережі та швидкість передачі. Також було розглянуто вплив MTU на втрати пакетів в мережі та описано чому краще не перевищувати допустимий розмір MTU для будь-якої мережі.

РОЗДІЛ 3. СТВОРЕННЯ ДОСЛІДНОГО ОТОЧЕННЯ.

3.1 Топологія досліджуваного середовища: переваги і недоліки.

Після визначення параметрів модифікації протоколу UDP, можна приступати до побудови мережі на котрій буде тестуватися модифікований протокол UDP.

Необхідно визначитися з топологією мережі. В нашому випадку буде використовуватися змішана топологія мережі з використанням топології кільце та топології зірка. Так як в сучасних мережах дуже важлива надійність та масштабованість, приступимо до визначення переваг та недоліків використаних топологій.

Топологія кільце.

Кільцева мережа являє собою конфігурацію, в якій кожен вузол з'єднується рівно з двома іншими вузлами, утворюючи єдиний безперервний шлях для сигналів через кожен комп'ютер - кільце.

Кільцеві топології з'єднують всі пристрої мережі в послідовний ланцюг. Дані переміщаються з одного пристрою на інший, поки не досягають місця призначення і, нарешті, не повертаються в операційний центр. Ця конфігурація вимагає меншої кількості кабелів і траншей, ніж альтернативні топології типу "зірка", і, отже, вона простіше і економічніше в реалізації.

Переваги топології:

1. Легке налаштування. Для підключення комп'ютерів один до одного не потрібен сервер або центральна робоча станція. Вони можуть бути легко пов'язані між собою, поєднуючи один пристрій з іншим. Вона дешевше топології типу "зірка" або "дерево", обидві з яких вимагають центрального або головного пристрою для керування вузлами.

2. Передавання даних. Кільцева топологія може обробляти великий об’єм даних, оскільки дані передаються в одну сторону. Це спрощує потік даних і

запобігає перевантаження мережі. Це також знижує ймовірність пошкодження даних.

3. Усунення несправностей. Коли відбувається помилка, легко визначити, де вона сталася, оскільки послідовна передача даних, дає зрозуміти на якому з вузлів зв'язок був розірваний.

4. Якість роботи при навантаженні. Не всі системи можуть витримувати більшу потокову навантаження на мережу. Наприклад, якщо порівнювати кільце з шиною, то перша буде працювати значно краще. Топологія кільце може досить спокійно працювати в умовах підвищеного навантаження.

5. Пропускна здатність. Підключення додаткових вузлів майже або зовсім не впливає на пропускну здатність мережі.

Недоліки:

1. Збої мережі. Не дивлячись на те що їх легко усунути, при збої одного пристрою відбувається збій всієї мережі через обрив лінії зв'язку. Поки вузол не буде полагоджений або замінений, мережа працювати не буде.

2. Розширення. Інший недолік такої конфігурації виявляється, при розширенні мережі. Якщо у вихідній конфігурації є п'ять комп'ютерів, а потім потрібно додати ще п'ять, то доведеться відключити всю мережу, перш ніж приступати до її розширення. Щоб розмістити додаткові комп'ютери в таку систему, необхідно відключити кожне з'єднання і під'єднати нові пристрої в установку зі зворотним зв'язком, перш ніж знову перенастроювати всю мережу.

3. Одне з'єднання. В даному типі підключення використовується кабель однієї довжини, що з'єднує всі комп'ютери і утворює петлю. У разі обриву кабелю все системи в мережі не зможуть отримати доступ до мережі. Тому виникає повна залежність від одного кабелю.

4. Швидкість роботи. Пакети даних повинні проходити через кожен комп'ютер між відправником і отримувачем, тому це може призводити до уповільнення передачі.

Використання топології кільце для маршрутизаторів може усунути більшість недоліків цієї топології. Отже, було прийнято рішення використовувати цю топологію в основі мережі. Оскільки будуть використовуватися маршрутизатори, то при збої на одному з маршрутизаторів мережа не перестане функціонувати повністю. Вийде з ладу лише одна підмережа котра підключена до цього маршрутизатора.

Топологія зірка

У цій топології кожен пристрій у мережі має власний кабель, який підключається до комутатора або концентратора. Хаб надсилає кожен пакет даних на кожен пристрій, тоді як комутатор надсилає пакет даних лише на цільовий пристрій.

Перевагами топології зірка є:

1.Легко керувати та підтримувати мережу, оскільки кожен вузол потребує окремого кабелю.

2.Простіше знайти проблеми, оскільки несправність кабелю стосується лише одного користувача.

3.Розширення мережі не впливає на її роботу.

4.Завдяки концентратору або комутатору управління мережею набагато простіше.

5.Легко знайти неполадки в мережі.

Недоліки топології зірка:

1.Продуктивність мережі залежить від центру концентратора або комутатора.

2.Якщо комутатор або концентратор вийде з ладу, мережа перестане функціонувати.

3.Для цієї топології необхідно використовувати більше проводів порівняно з топологією кільця та шини.

Отже, топологія зірка добре підходить для підключення кінцевих пристроїв тестового стенду. Приступимо до побудови.

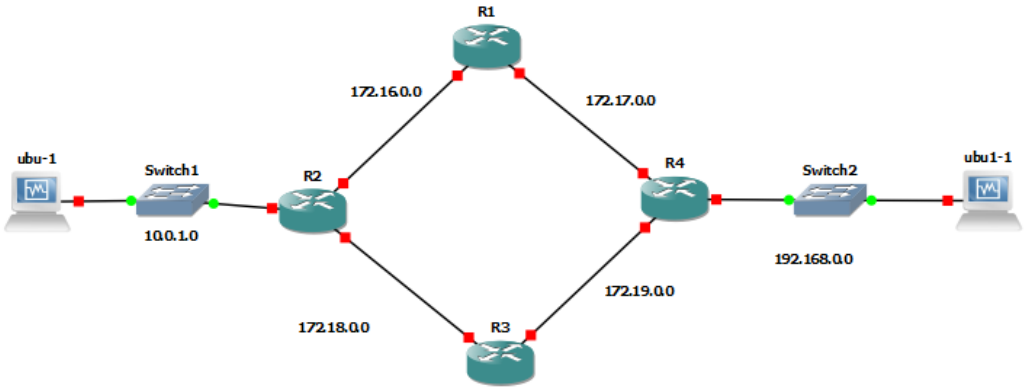


Рисунок 3.1 Стенд тестування модифікованого протоколу UDP

В даному випадку(Рис 3.1) було побудовано мережу з 2 персональними комп’ютерами з ОС Ubuntu 16.04, 4 образа маршрутизатора Cisco та 2 комутатора взятих з GNS3.

3.2 Протоколи функціонуючі на всіх рівнях.

Після побудови мережі необхідно визначити які протоколи будуть функціонувати в мережі на проміжних пристроях (маршрутизатори) так і на кінцевих (комп'ютери). Для цього розглянемо чотирирівневу (Рис 3.1) модель стека TCP/IP та порівняємо її з моделлю OSI.

TCP / IP - Transmission Control Protocol / Internet Protocol (Протокол Управління Передачею Даних / Міжмережевий Протокол). Стек TCP / IP - сукупність протоколів організації взаємодії між структурами і програмними

компонентами мережі; являє собою програмно реалізований набір протоколів міжмережевого взаємодії.[1]

Почнемо з розгляду подібностей, які мають ці моделі. Нижче наведено схожість між базовою моделлю OSI та еталонною моделлю TCP / IP.

1. Обидві моделі мають багатошарову архітектуру.
2. Шари (рівні) забезпечують подібні функціональні можливості.
3. Обидві моделі є стеком протоколів.
4. Обидві є еталонними моделями.

Наступним кроком розглянемо відмінності між моделлю OSI та моделлю TCP/IP. Нижче наведено основні відмінності між базовою моделлю OSI та еталонною моделлю TCP / IP, наведеним нижче схематичним порівнянням.(Табл. 3.1)

OSI (відкрите системне з'єднання)	**TCP / IP (протокол управління передачею / протокол Інтернету)**
1. OSI - це загальний, незалежний від протоколу стандарт, який виступає шлюзом зв'язку між мережею та кінцевим користувачем.	1. TCP / IP-модель базується на стандартних протоколах, навколо яких розвивався Інтернет. Це протокол зв'язку, який дозволяє підключати хости через мережу.
2. У моделі OSI транспортний рівень гарантує доставку пакетів.	2. У моделі TCP / IP транспортний рівень гарантує доставку пакетів. Проте модель TCP / IP є більш надійною.
4. Модель OSI має окремий рівень презентації та рівень сесії.	4. TCP / IP не має окремого рівня презентації або рівня сесії.

5. Транспортний рівень орієнтований на з'єднання.	5. Транспортний шар орієнтований як на з'єднання, так і на роботу без встановлення з’єднання.
7. OSI - еталонна модель, навколо якої будуються мережі. Зазвичай вона використовується як навчальний посібник.	7. Модель TCP / IP певним чином є реалізацією моделі OSI.
8. Мережевий рівень моделі OSI забезпечує сервіс, орієнтований на з'єднання, і без встановлення зв'язку.	8. Мережевий рівень у моделі TCP / IP не встановлює з’єднання
9. Модель OSI має проблему з додаванням протоколів до моделі.	9. Модель TCP / IP використовує стандартні протоколи
10. Протоколи приховані в моделі OSI і легко замінюються в міру зміни технології.	10. У TCP / IP заміна протоколу складний процес.
11. Модель OSI дуже чітко визначає послуги, інтерфейси та протоколи та чітко розрізняє їх. Це не залежить від протоколу.	11. У TCP / IP послуги, інтерфейси та протоколи чітко не розділені. Це також залежить від протоколу.
12. Модель має 7 рівнів	12. Модель має 4 рівні

Таблиця 3.1 Порівняння моделі OSI з моделлю TCP/IP

З огляду на значення двох моделей, модель OSI є концептуальною моделлю. Вона в основному використовується для опису, обговорення і розуміння окремих мережевих функцій. Однак, TCP / IP в першу чергу сконструйована для побудови мереж, а не для того щоб діяти, як опис покоління для всіх мережевих взаємодій, як модель OSI. Модель OSI є спільною,

незалежної від протоколу, але більшість протоколів і систем дотримуються її, в той час як модель TCP / IP заснована на стандартних протоколах мережі Інтернет. Інший момент, який слід відзначити в моделі OSI полягає в тому, що не всі рівні використовуються в більш простих додатках. У той час як рівні 1, 2, 3 є обов'язковими для будь-якої передачі даних, додаток може використовувати якийсь унікальний інтерфейс рівня замість звичайних верхніх рівнів в моделі.

Отже, далі буде розглядатися саме модель TCP/IP(Табл. 3.2). Будуть підібрані протоколи для кожного з чотирьох рівнів цієї моделі. Та описано призначення кожного рівня моделі TCP/IP.

Рівень стеку	**Назва протоколу**
4.Прикладний	HTTP, FTP, SMTP, Telnet, DNS
3.Транспортний	TCP, UDP
2.Міжмережевий	IP, ARP
1.Канальний	Ethernet, Frame Relay,PPP,ATM, Token Ring

Таблиця 3.2 Стек протоколів TCP/IP, TCP/IP-модель

1-й рівень повинен забезпечити інтеграцію в складену мережу будь-якої іншої мережі, незалежно від технології передачі даних цієї мережі.

2-й рівень повинен забезпечити можливість передачі пакетів через складену мережу, використовуючи оптимальний маршрут.

3-й рівень вирішує завдання забезпечення надійної передачі даних між джерелом і адресатом.

4-й рівень об'єднує всі мережеві служби і послуги, що надаються мережею користувачу.

В TCP / IP досить добре розвинений перший рівень, відповідний до 1 і 2 рівнів моделі OSI. Другий рівень TCP / IP - IP. Також присутній ICMP -

протокол керуючих повідомлень мережі. IP не гарантує надійної передачі даних. Основним завданням цього рівня є вибір найкращого маршруту. Вирішення цього завдання IP перекладає на RIP і OSPF протоколи. В мережі для тестування модифікованого протоколу UDP буде використовуватися протокол RIPv2 на маршрутизаторах. Третій рівень - TCP, основна функція - надійність і правильність доставки даних. Також використовується UDP, в ньому кожен пакет передається незалежно. Надійність доставки даних не гарантується, тому що не встановлюється зв'язок заздалегідь. Зазвичай по UDP передаються дані, не критичні до надійності. 4 рівень - набір служб і послуг, пропонованих користувачу.

Чому існують два транспортних протоколу TCP і UDP, а не один з них?

Справа в тому, що вони надають різні послуги прикладним процесам. Більшість прикладних програм користуються тільки одним з них. Необхідно обирати той протокол, який найкращим чином відповідає вашим потребам. Якщо вам потрібна надійна доставка, то кращим може бути TCP. Якщо вам потрібна доставка датаграмм, то краще може бути UDP. Якщо вам потрібна ефективна доставка по довгому і ненадійному каналу передачі даних, то краще може підійти протокол TCP. Якщо потрібна ефективність на швидких мережах, то кращим може бути протокол UDP. Якщо ваші потреби не потрапляють ні в одну з цих категорій, то вибір транспортного протоколу не ясний. Однак прикладні програми можуть усувати недоліки обраного протоколу. Наприклад, якщо ви вибрали UDP, а вам необхідна надійність, то прикладна програма повинна забезпечити надійність. Якщо ви вибрали TCP, а вам потрібно передавати записи, то прикладна програма повинна вставляти маркери в потік байтів так, щоб можна було розрізнити записи.

Повернемося до тестового стенду. Мережа побудована, але на маршрутизаторах не налаштована маршрутизація, це означає, що данні

відправлені по мережі не дійдуть до отримувача. Отже, необхідно розглянути переваги існуючих типів маршрутизації та налаштувати її на маршрутизаторах.

В TCP / IP передбачено два типи маршрутизації: статична і динамічна.

Статична маршрутизація означає, що таблиці маршрутизації обслуговуються вручну. Цей тип маршрутизації рекомендується застосовувати тоді, коли ваша мережа взаємодіє з однією або двома іншими мережами. Однак якщо мережа з'єднана з великим числом мереж, то число шлюзів різко зростає, і для обслуговування таблиць маршрутизації вручну потрібно чимало часу.

При динамічній маршрутизації таблиці маршрутизації автоматично оновлюються. Пристрої на котрих налаштована маршрутизація безперервно отримують інформацію та оповіщують інші пристрої, що вони є в мережі, тому вони постійно оновлюють таблиці маршрутизації.

	Статична маршрутизація	**Динамічна маршрутизація**
Складність налаштування	Збільшується з ростом розміру мережі	Зазвичай не залежить від розміру мережі
Зміни в топології	Необхідна участь системного адміністратора	Змінюється автоматично в залежності зі змінами топології
Масштабованість	Підходить для простих топологій	Підходить для будь-якої топології
Безпечність	Високий рівень	Низький рівень

Використання ресурсів маршрутизатора	Не потребує додаткових ресурсів	Використовує ЦП, пам’ять, смугу пропускання каналу
Передбачуваність маршрута	Маршрут до місця призначення завжди однаковий	Маршрут змінюється в залежності від навантаження мережі

Таблиця 3.3 Порівняння статичної та динамічної маршрутизації

Отже, розглянувши всі переваги та недоліки типів маршрутизації (Табл. 3.3), можна зробити висновок, що в нашому випадку краще використовувати динамічну маршрутизацію. Оскільки було обрано динамічну маршрутизацію,на маршрутизаторах буде налаштовано протокол RIPv2.

Таким чином, в мережі для тестування модифікованого протоколу UDP на канальному рівні буде використовуватися протокол Ethernet, на міжмережевому рівні будуть використовуватися протоколи IP та ARP, на транспортному рівні буде використовуватися протокол UDP, а роль прикладного рівня візьме на себе додаток котрий буде розроблено та встановлено на кінцевих пристроях.

3.3 Кінцеві пристрої: основне завдання та специфікація.

Мережеві пристрої, з якими користувачі знайомі найкраще, називаються кінцевими пристроями або вузлами. Ці пристрої утворюють інтерфейс між користувачами і комунікаційною мережею, яка надає зв'язок.

Кінцевими пристроями називать такі технічні засоби як:

1. Комп'ютери (ноутбуки, файлові сервери або веб-сервери)
2. VoIP-телефони
3. Мережеві принтери

4. Термінальне обладнання TelePresence

5. IP-камери

6. Пересувні кишенькові пристрої (наприклад, смартфони, планшетні ПК)

Кінцевий пристрій є або джерелом, або отримувачем повідомлення, переданого по мережі. Щоб відрізняти один пристрій від інших, кожному пристрою в мережі призначена адреса. Коли вузол ініціює взаємодію, він використовує адресу пристрою призначення, щоб визначити, куди повинно бути направлено повідомлення.

Для того щоб додати кінцеві пристрої до віртуальної мережі, необхідно використовувати технології віртуалізації. Розглянемо що таке віртуалізація та навіщо вона потрібна нижче.

Віртуалізація - це технологія, що симулює функції апаратного забезпечення для створення таких програмних IT-сервісів, як сервери додатків, сховище і мережі.

В процесі віртуалізації створюється кілька віртуальних машин на базі однієї фізичної машини з використанням програмного забезпечення. Оскільки такі віртуальні машини поводяться так само, як фізичні, при цьому використовуючи обчислювальні ресурси тільки однієї машини, віртуалізація дозволяє IT-організаціям запускати кілька операційних систем на одному сервері (який також називається хостом). Під час цих операцій гіпервізор виділяє обчислювальні ресурси кожної віртуальної машини в міру необхідності.

Оскільки, для тестування модифікованого протоколу UDP необхідно не менше ніж 2 комп’ютери (джерело повідомлення та отримувач) будемо використовувати серверний тип віртуалізації. При такому типі віртуалізації можна отримати 2 повноцінних віртуальних комп’ютери, на котрих можна встановлювати додатки та виконувати будь-які операції. В якості сервера буде комп’ютер на котрому встановлено Oracle VM VirtualBox.

VirtualBox - це програмне забезпечення, яке імітує справжній комп'ютер, що дає можливість користувачеві встановлювати, запускати і використовувати інші операційні системи, як звичайні додатки.

Віртуальна машина створює якесь ізольоване оточення на комп'ютері, яке складається з віртуальних компонентів реального ПК: жорсткого диска, відеокарти, оперативної пам'яті, різних контролерів пристроїв. Таким чином, встановлена в VirtualBox операційна система буде повністю впевнена в тому, що вона працює на реальному пристрої.

В мережі побудованій для тестування модифікованого протоколу UDP, використовується 2 кінцевих прострої, кожен з яких може виступати як джерелом повідомлення так і отримувачем. В нашому випадку використовується два комп’ютери з операційною системою Ubuntu 16.04, котрі були додані до мережі, за допомогою VirtualBox. На кожному з пристроїв встановлено інтерпретатор Python 3.5, оскільки наш додаток, котрий буде використовувати модифікований протокол UDP, розроблений саме на мові програмування Python 3.

3.4 Пояснення вибору графічного стимулятору мережі GNS3.

Для побудови стенду тестування модифікованого протоколу UDP використовувався графічний стимулятор мережі. Розглянемо цю програму детальніше.

GNS3 - симулятор графічної мережі. За допомогою цього інструменту ви зможете створювати і моделювати дуже складні і передові Cisco мережі.

Проектування мережі в GNS3, робить її більш ефективним навчальним матеріалом. Немає реальної необхідності представляти деякі абстрактні сценарії за текстовою консоллю. Можна змінювати розташування мережевих пристроїв і все це візуально.

Слід зазначити і те, що багато системних адміністраторів і інженери мереж знайдуть цей інструмент дуже корисним. Оскільки, можна моделювати

нову конфігурацію, різні образи ISO, або, можливо, повністю зробити реконструкцію деяких частин складної мережі, що набагато простіше з цією програмою, ніж її застосування в реальному світі, і, найголовніше, це більш економічно ефективний процес. Прогрес може бути досягнутий набагато швидше.

Крім того, краща частина полягає в тому, що продукт обробляє установку і настройку необхідних утиліт автоматично. Всі емулятори попередньо включені в інсталяційний пакет GNS3 .У випадку з установкою GNS3 на операційну систему Microsoft Windows, також необхідно встановлювати WinPcap, це необхідно для перехоплення пакетів і бібліотек мережевого монітора, на якому безліч інших додатків.

Переваги використання GNS3:

1. Перша і найголовніша причина - повний функціонал емульованих пристроїв. Тобто запустивши той же маршрутизатор Cisco, нам будуть доступні практично всі функції, які працюють на реальному маршрутизаторі. Якщо згадати Cisco Packet Tracer, то там значна частина функціоналу недоступна.
2. Можлівість побудови гетерогенних мереж. Мається на увазі, що ми можемо зібраті схему де будуть не только пристрої Cisco, а й Juniper, Mikrotik, CheckPoint и т.д.
3. Додавання в мережу повноцінних робочих станцій і серверів. Знову ж таки, якщо згадати Cisco Packet Tracer, то там в якості кінцевих пристроїв були доступні клієнтські комп'ютери або сервери з дуже обмеженим функціоналом. У GNS3 ми можемо додати повноцінний комп'ютер з Windows 7 або Ubuntu. Що й було зроблено в тестовому стенді.
4. І ще одна, четверта, багато важлива причина - Безкоштовність! GNS3 знаходиться у вільному доступі і не має будь-яких обмежень щодо використання.

Але цей симулятор має декілька незначних недоліків:

1. Головний недолік - відсутність можливості емулювати комутатори. Але в GNS3 є стандартний пристрій котрий виконує роботу комутатора. Справа в тому, що в реальних комутаторах велику кількість ASIC мікросхем, які поки що неможливо емулювати на звичайному комп'ютері. Саме ці ASIC мікросхеми забезпечують величезну швидкість обробки пакетів. А ось маршрутизатори працюють на основі процесора, який дуже схожий на процесор звичайного комп'ютера, а іноді і точно такий же. Тому проблем з емуляцією маршрутизатора не виникає. Однак процесор значно повільніше ASIC мікросхем.
2. Ще один важливий недолік - дуже високі вимоги до системних ресурсів. Хоча це скоріше не проблема GNS3, а проблема пристроїв котрі емулюються. GNS3 на відміну від Cisco Packet Tracer працює з реальними прошивками пристроїв. Для прикладу, щоб запустити 1 образ Cisco маршрутизатора необхідно 512мб оперативної пам’яті. В топології мережі котра була побудована використовується 4 роутера, а також 2 віртуальні машини. Кожна з яких використовує приблизно 2048 Мб. Тобто для запуску мережі необхідно 512*4 + 2048*2 =6144 Мб оперативної пам’яті. Це при тому що на комп’ютері котрий симулює мережу ще працює Windows 10.
3. Третя вада - баги. У GNS3 їх досить багато. Для того щоб їх прибрати випускають релізи нових версій GNS3. Майже кожен з яких створює нові баги.

Отже, незважаючи на недоліки, можна зробити висновок, що GNS3 дуже потужний інструмент при побудові віртуальної мережі. Можна емулювати образи пристроїв та працювати з ними як в реальному житті. GNS3 має свої

недоліки, але переваги набагато значніші, тим самим недоліки майже не помітні, а також він є безкоштовним та дуже зручним інструментом.

3.5 Висновки до розділу 3

Даний розділ присвячено побудові мережі для тестування модифікованого протоколу UDP. Було обрано GNS3 в якості інструменту для побудови віртуальної мережі. Також обрано VirtualBox для додавання віртуальних кінцевих пристроїв до мережі.

Розглянуто переваги та недоліки статичної та динамічної маршрутизації. Було прийнято рішення використовувати протокол RIPv2 для налаштування маршрутизації на маршрутизаторах. Також детально розглянуто модель TCP/IP мережі та визначено які протоколи будуть функціонувати в нашій мережі відповідно до моделі TCP/IP. Таким чином, було обрані такі протоколи та технології:

1. Рівень доступу до мережі (Ethernet)
2. Мережевий рівень (IP, RIPv2)
3. Транспортний рівень(UDP)
4. Прикладний рівень (додаток на віртуальних машинах)

РОЗДІЛ 4. ПРОТОТИП МОДИФІКАЦІЇ ПРОТОКОЛУ UDP

4.1 Концепція модифікації протоколу UDP.

Після побудови мережі для тестування модифікованого протоколу UDP, можна приступати до написання програми для передачі повідомлення від джерела до отримувача. Як було сказано в розділі 3, джерелом повідомлення є сервер, а отримувачем повідомлення є клієнт.

Перед написанням програмного забезпечення на клієнті та сервері необхідно визначитися з мовою програмування. В нашому випадку буде використовуватись мова програмування Python 3, оскільки вона має багато переваг над іншими мовами. Такими перевагами є:

1. Динамічна типізація. У python не треба заздалегідь оголошувати тип змінної, що дуже зручно при розробці.
2. Підтримка модульності. Можна легко написати свій модуль і використовувати його в інших програмах.
3. Вбудована підтримка Unicode в рядках. В Python необов'язково писати все англійською мовою.
4. Інтеграція з C / C ++, якщо можливостей python недостатньо.
5. Зрозумілий синтаксис, що сприяє ясному відображенню коду. Зручна система функцій дозволяє при грамотному підході створювати код, в якому буде легко розібратися іншій людині в разі необхідності.
6. Величезна кількість стандартних модулів, які встановлюються на комп’ютер разом з Python 3, також є можливість завантажувати інші модулі з Інтернет. У деяких випадках для написання програми досить лише знайти підходящі модулі і правильно їх скомбінувати. Таким чином, ви можете думати про складання програми на більш високому рівні, працюючи з уже готовими елементами, які виконують різні дії.

7. Кросплатформеність. Програма, написана на Python, буде функціонувати абсолютно однаково незалежно від того, в якій операційній системі вона запущена.

Після вибору мови програмування для написання додатків клієнта та серверу. Додатки котрі будуть створені матимуть можливість передавати такі типи повідомлень як: відео потік, зображення, текстовий документ, текст та голос. Після успішної реалізації додатків будемо змінювати основні параметри модифікованого UDP [9] котрі впливають на швидкість передачі та втрату пакетів в IP-мережі.

Отже, можна зробити висновок, що використання мови Python 3 значно полегшить написання програми. Оскільки в мережі Інтернет є безліч готових модулів, а це означає що немає необхідності написання програми з нуля. Досить лише правильно скомбінувати вже готові функції з інших модулів. Після написання програмного коду для клієнта та сервера будемо змінювати основні параметри UDP протоколу.

4.2 Концепція програмного забезпечення клієнта UDP.

Після вибору мови програмування, можна приступати до написання програмного коду для клієнта. Роль клієнта буде виконувати віртуальна машина з ОС (Ubuntu 16.04) яка має назву “ubu1”(Рис 4.1).

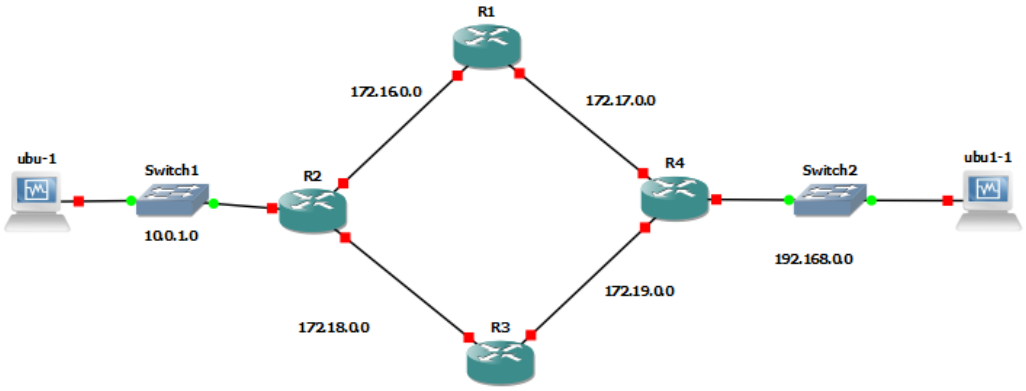


Рисунок 4.1 Мережа для тестування модифікованого протоколу UDP

Спочатку напишемо код, котрий дасть можливість клієнту відправляти текстові повідомлення до серверу. Перед написанням клієнтського додатку необхідно визначитися з модулями котрі будуть використовуватись. В програмах клієнта та сервера будуть використовуватися такі модулі як: socket, sys, time.

Модуль socket використовуємо для того щоб реалізувати відправку та прийом датаграм, від джерела до отримувача повідомлення. Взагалі сокет - це програмний інтерфейс для забезпечення інформаційного обміну між процесами. Для того щоб відправити датаграми та успішно їх прийняти, необхідно використовувати порт та IP адресу отримувача.

Модуль sys буде використовуватись для того щоб при запуску додатку була можливість обрати номер порта та адресу отримувача, а не змінювати програмний код кожен раз коли є необхідність використання іншого номеру порта або відправлення повідомлення іншому користувачу.

Модуль time на сервері дасть можливість простою для створення документу формату .txt якщо ми будемо передавати великі повідомлення. А

також на клієнті дасть можливість змінювати параметр packet pacing, тобто буде можливість рівномірної відправки датаграм з певним інтервалом часу.

Спочатку реалізуємо відправку простих повідомлень з віртуальної машини ubu1 на ubu. Програмний код такого клієнта матиме наступний вигляд (Рис 4.2).

```
import socket
import sys
import time

what = input('Enter wich data you want to send (txt, message or file: ')
if what == 'message':
    host = '192.168.0.2'
    port = 7777
    client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    msg = input('Enter message to send: ')
    client.sendto(msg.encode('utf-8'), ('10.0.1.2', 7777))
    d = client.recvfrom(1024)
    reply = d[0]
    addr = d[1]
    print ('Server reply: ' + reply.decode('utf-8'))
    client.close()
```

Рисунок 4.2 Реалізація відправки текстового повідомлення з віртуальної машини “ubu1”.

Цей код дає можливість відправлення повідомлень котрі користувач сам вписує до програми. Якщо це повідомлення має розмір менше ніж 1024 байт воно буде інкапсульовано лише в 1 датаграму.

Після написання коду для відправки простих повідомлень, можна приступати до написання коду для відправки більшої кількості даних, наприклад, текстового документу. Код клієнтської програми матиме наступний вигляд (Рис 4.3).

```
elif what == 'txt':
    UDP_IP = '192.168.0.2'
    UDP_PORT = 7777
    buf = 1024
    file_name = str(input(" Name of the documend: "))


    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(file_name.encode(), ('10.0.1.2', 7777))
    print('Sending %s' % file_name)

    f = open(file_name, 'r')
    data = f.read(buf)
    while(data):
        if(sock.sendto(data.encode(), ('10.0.1.2', 7777))):
            data = f.read(buf)
            time.sleep(0.02)
    sock.close()
    f.close()
elif what == 'file':
    def checkArg():
```

Рисунок 4.3 Реалізація відправки текстового документу.

Цей додаток працює наступним чином, спочатку відкривається текстовий документ, після чого з нього зчитується по 1024 байт , кодуються та інкапсулюються в датаграми для відправки по мережі. Серверний додаток створює текстовий документ, декодує повідомлення та записує його в створений документ.

Після написання коду для відправки текстового документу можна приступати до відправки більшої кількості даних, наприклад, таких як: зображення форматів (.png, .jpg, .jpeg) або відео потік (.mp4). Програмний код клієнту для відправки файлів матиме наступний вигляд (Рис4.4).

```
elif what == 'file':
    def checkArg():

        if len(sys.argv) != 3:
            print(
            "ERROR. Wrong number of arguments passed. System will exit. Next time please supply 2 arguments!")
            sys.exit()
        else:
            print("2 Arguments exist. We can proceed further")


    def checkPort():
        if int(sys.argv[2]) <= 5000:
            print(
            "Port number invalid. Port number should be greater than 5000 else it will not match with Server port. Next time enter valid port.")
            sys.exit()
        else:
            print("Port number accepted!")

    checkArg()
    try:
        socket.gethostbyname(sys.argv[1])
    except socket.error:
        print("Invalid host name. Exiting. Next time enter in proper format.")
        sys.exit()

    host = sys.argv[1]
    try:
        port = int(sys.argv[2])
    except ValueError:
        print("Error. Exiting. Please enter a valid port number.")
        sys.exit()
    except IndexError:
        print("Error. Exiting. Please enter a valid port number next time.")
        sys.exit()

    checkPort()


        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
            print("Client socket initialized")
            s.setblocking(0)
            s.settimeout(15)
        except socket.error:
            print("Failed to create socket")
            sys.exit()



        while True:
            command = input(
                "Please enter a command: \n1. get [file_name]\n2. put [file_name] ")


            CommClient = command.encode('utf-8')

            try:
                s.sendto(CommClient, (host, port))
            except ConnectionResetError:
                print(
                "Error. Port numbers are not matching. Exiting. Next time please enter same port numbers.")
                sys.exit()

            CL = command.split()
            print(
            "We shall proceed, but you may want to check Server command prompt for messages, if any.")

            if CL[0] == "get":
                print("Checking for acknowledgement")
                try:
                    ClientData, clientAddr = s.recvfrom(51200)
                except ConnectionResetError:
                    print(
                    "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                    sys.exit()
                except:
                    print("Timeout or some other error")
                    sys.exit()
                text = ClientData.decode('utf8')
```

```
            print(text)
            print("Inside Client Get")

            try:
                ClientData2, clientAddr2 = s.recvfrom(51200)
            except ConnectionResetError:
                print(
                "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                sys.exit()
            except:
                print("Timeout or some other error")
                sys.exit()

            text2 = ClientData2.decode('utf8')
            print(text2)

            if len(text2) < 30:
                if CL[0] == "get":
                    BigC = open("Received-" + CL[1], "wb")
                    d = 0
                    try:
                        # number of paclets
                        CountC, countaddress = s.recvfrom(1024)
                    except ConnectionResetError:
                        print(
                        "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                        sys.exit()
                    except:
                        print("Timeout or some other error")
                        sys.exit()

                    tillC = CountC.decode('utf8')
                    tillCC = int(tillC)
                    print("Receiving packets will start now if file exists.")

                    while tillCC != 0:
                        ClientBData, clientbAddr = s.recvfrom(1024)
                        dataS = BigC.write(ClientBData)
                        d += 1
                        print("Received packet number:" + str(d))
                        tillCC = tillCC - 1

                    BigC.close()
                    print(
                    "New Received file closed. Check contents in your directory.")

        elif CL[0] == "put":
            print("Checking for acknowledgement")
            try:
                ClientData, clientAddr = s.recvfrom(1024)
            except ConnectionResetError:
                print(
                "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                sys.exit()
            except:
                print("Timeout or some other error")
                sys.exit()

            text = ClientData.decode('utf8')
            print(text)
            print("We shall start sending data.")

            if text == "Valid Put command. Let's go ahead ":
                if os.path.isfile(CL[1]):

                    c = 0


                    size = os.stat(CL[1])
                    sizeS = size.st_size

                    print("File size in bytes: " + str(sizeS))
                    Num = int(sizeS / 1024)
                    Num = Num + 1
                    print("Number of packets to be sent: " + str(Num))
                    till = str(Num)
                    tillC = till.encode('utf8')
                    s.sendto(tillC, clientAddr)
                    tillIC = int(Num)
                    GetRun = open(CL[1], "rb")

                    while tillIC != 0:

```

```
                        Run = GetRun.read(1024)


                        s.sendto(Run, clientAddr)
                        c += 1
                        tillC -= 1

                        print("Packet number:" + str(c))
                        print("Data sending in process:")



                    GetRun.close()

                    print("Sent from Client - Put function")
                else:
                    print("File does not exist.")
            else:
                print("Invalid.")


        else:
            try:
                ClientData, clientAddr = s.recvfrom(51200)
            except ConnectionResetError:
                print(
                "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                sys.exit()
            except:
                print("Timeout or some other error")
                sys.exit()
            text = ClientData.decode('utf8')
            print(text)

    print("Program will end now. ")
    quit()

else:

    print('Inncorect input')
```

Рисунок 4.4 Код для відправки файлів по протоколу UDP

Це більш складна реалізація клієнтської частини. Для того щоб запустити цей код необхідно вказати вільний номер порту, котрий сервер використовує саме для цього клієнта та IP адресу сервера. Клієнт визначає скільки датаграм необхідно відправити серверу для повної передачі файлу. Після чого кодує дані та відправляє серверу, котрий їх розкодовує та збирає файл без перевірки на цілісність. Якщо буде втрачено багато датаграм то забраження буде виглядати гірше та матиме менший розмір.

Загальний вигляд програмного забезпечення матиме вигляд(Додаток 1):

Отже, з написання додатку клієнта завершено. Додаток клієнта дає можливість відправляти по протоколу UDP такі дані як текстове повідомлення, текстовий документ, зображення та відео будь-якого формату, але перед

відправкою необхідно вказати який тип даних буде передаватись. Можна приступати до написання серверної частини (отримувача даних).

4.3 Концепція програмного забезпечення сервера UDP.

Для написання серверної частини буде імпортовано ще один модуль, а саме select. Модуль select дає можливість серверу працювати в режимі очікування. Тоб-то не буде необхідності кожен раз заново запускати додаток після того як сервер отримає повідомлення.

Приступимо до написання простого серверного додатка для прийому текстових повідомлень від клієнта. Програмний код матиме наступний вигляд (Рис 4.5)

```
import socket
import sys
import select
import time
what = input('Which type of data do you want receive (txt, message or file)?: ')
if what == 'message':
    timeout = 60
    host = '10.0.1.2'
    port = 7777
    addr = (host, port)
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind(addr)

    while True:
        print('Waiting for data ({0} seconds)...'.format(timeout))
        server.settimeout(timeout)
        try:
            d = server.recvfrom(1024)
        except socket.timeout:
            print('Time is out. {0} seconds have passed'.format(timeout))
            break
        received = d[0]
        addr = d[1]
        print('From: ', addr )
        print('Received data: ', received.decode('utf-8'))

        msg = input('Enter message to send: ')
        server.sendto(msg.encode('utf-8'), addr)
    server.close()
```

Рисунок 4.5 Програмний код сервера для отримання текстового повідомлення.

Після реалізації відправки текстових повідомлень, можна починати написання серверної частини для отримання тестових документів формату .txt. Програмний код матиме наступний вигляд (Рис 4.6).

```
elif what == 'txt' :
    UDP_IP = '10.0.1.2'
    IN_PORT = 7777
    timeout = 3


    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((UDP_IP, IN_PORT))

    while True:
        data,addr = sock.recvfrom(1024)
        if data:
            print('File name: ', data.decode('utf-8'))
            file_name = data.strip()

        f = open(file_name, 'wb')

        while True:
            ready = select.select([sock], [], [],timeout)
            if ready[0]:
                data, addr = sock.recvfrom(1024)
                f.write(data)
            else:
                print('%s Finish!' % file_name.decode('utf-8'))
                f.close()
                break
```

Рисунок 4.6 Програмний код сервера для отримання текстових документів формату “.txt”

Після успішної передачі текстового документу, приступимо до реалізації більш складного серверного додатку для отримання файлів (зображення або відео потоку) відправлених по протоколу UDP. Програмний код серверного додатку матиме наступний вигляд (Рис 4.7)

```
elif what == 'file':
    def checkArg():

        if len(sys.argv) != 2:
            print(
            "ERROR. Wrong number of arguments passed. System will exit. Next time please supply 1 argument!")
            sys.exit()
        else:
            print("1 Argument exists. We can proceed further")


    def checkPort():
        if int(sys.argv[1]) <= 5000:
            print(
            "Port number invalid. Port number should be greater than 5000. Next time enter valid port.")
            sys.exit()
        else:
            print("Port number accepted!")




    def ServerGet(g):
        print("Sending Acknowledgment of command.")
        msg = "Valid Get command. Let's go ahead "
        msgEn = msg.encode('utf-8')
        s.sendto(msgEn, clientAddr)
        print("Message Sent to Client.")

        print("In Server, Get function")

        if os.path.isfile(g):
            msg = "File exists. Let's go ahead "
            msgEn = msg.encode('utf-8')
            s.sendto(msgEn, clientAddr)
            print("Message about file existence sent.")

            c = 0
            sizeS = os.stat(g)
            sizeSS = sizeS.st_size  # number of packets
            print("File size in bytes:" + str(sizeSS))
          NumS = int(sizeSS / 1024)
          NumS = NumS + 1
          tillSS = str(NumS)
          tillSSS = tillSS.encode('utf8')
          s.sendto(tillSSS, clientAddr)

          check = int(NumS)
          GetRunS = open(g, "rb")
          while check != 0:
              RunS = GetRunS.read(1024)
              s.sendto(RunS, clientAddr)
              c += 1
              check -= 1
              print("Packet number:" + str(c))
              print("Data sending in process:")
          GetRunS.close()
          print("Sent from Server - Get function")

        else:
            msg = "Error: File does not exist in Server directory."
            msgEn = msg.encode('utf-8')
            s.sendto(msgEn, clientAddr)
            print("Message Sent.")


    def ServerPut():
        print("Sending Acknowledgment of command.")
        msg = "Valid Put command. Let's go ahead "
        msgEn = msg.encode('utf-8')
        s.sendto(msgEn, clientAddr)
        print("Message Sent to Client.")

        print("In Server, Put function")
        if t2[0] == "put":

            BigSAgain = open(t2[1], "wb")
            d = 0
            print("Receiving packets will start now if file exists.")

            try:
                Count, countaddress = s.recvfrom(1024)
            except ConnectionResetError:
```

```
                print(
                "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                sys.exit()
            except:
                print("Timeout or some other error")
                sys.exit()

            tillI = Count.decode('utf8')
            tillI = int(tillI)


            while tillI != 0:
                ServerData, serverAddr = s.recvfrom(1024)

                dataS = BigSAgain.write(ServerData)

                d += 1
                tillI = tillI - 1
                print("Received packet number:" + str(d))


            BigSAgain.close()
            print("New file closed. Check contents in your directory.")



    def ServerElse():
        msg = "Error: You asked for: " + \
            t2[0] + " which is not understood by the server."
        msgEn = msg.encode('utf-8')
        s.sendto(msgEn, clientAddr)
        print("Message Sent.")


    host = ""
    checkArg()
    try:
        port = int(sys.argv[1])
    except ValueError:
        print("Error. Exiting. Please enter a valid port number.")
        sys.exit()
    except IndexError:
        print("Error. Exiting. Please enter a valid port number next time.")
        sys.exit()
    checkPort()

    #port = 7777
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        print("Server socket initialized")
        s.bind((host, port))
        print("Successful binding. Waiting for Client now.")
        # s.setblocking(0)
        # s.settimeout(15)
    except socket.error:
        print("Failed to create socket")
        sys.exit()

    # time.sleep(1)
    while True:
        try:
            data, clientAddr = s.recvfrom(1024)
        except ConnectionResetError:
            print(
            "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
            sys.exit()
        text = data.decode('utf8')
        t2 = text.split()
        #print("data print: " + t2[0] + t2[1] + t2[2])
        if t2[0] == "get":
            print("Go to get func")
            ServerGet(t2[1])
        elif t2[0] == "put":
            print("Go to put func")
            ServerPut()
        else:

            ServerElse()

    print("Program will end now. ")
    quit()

else:
    print('Incorrect input')
```

Рисунок 4.7 Програмний код сервера для отримання файлів (забраження, відео)

Остаточний серверний додаток має можливість отримання текстових повідомлень, текстових документів (.txt), зображень та відео. Все це було

реалізовано в одній програмі з можливістю вибору які данні буде отримувати сервер. Остаточний код програмного забезпечення серверу матиме вигляд (Додаток 2).

Отже, написання серверного додатку для UDP датаграм завершено. Було успішно відправлено та отримано всі типи даних. Можна приступати до модифікації основних параметрів протоколу UDP, таких як: buffer, MTU, packet pacing. Модифікація котрих знизить втрату пакетів в мережі та прискорить передачу повідомлень. Тоб-то шляхом експеременту визначимо оптимальне значення головних параметрів протоколу UDP.

4.4 Висновки до розділу 4

Цей розділ присвячений обгрунтуванню вибору мови програмування та написанню додатків клієнта та сервера. Було визначено, що використання мови програмування python3 значно полегшить написання додатків. А також обрано модулі котрі будуть використовуватись на додатках серверу та клієнту. Таким чином на клієнті будуть використовуватися такі бібліотеки як (socket, sys, time), а на сервері (socket, sys, time, select).

Модуль socket використовуємо для того щоб реалізувати відправку та прийом датаграм, від джерела до отримувача повідомлення.

Модуль sys буде використовуватись для того щоб при запуску додатку була можливість обрати номер порта та адресу отримувача.

Модуль time на сервері дасть можливість простою для створення документу формату .txt якщо ми будемо передавати великі повідомлення. А також на клієнті дасть можливість змінювати параметр packet pacing.

Модуль select дає можливість серверу працювати в режимі очікування.

РОЗДІЛ 5. ДОСЛІДЖЕННЯ ВПЛИВУ ПАРАМЕТРІВ ПЕРЕДАЧІ ПРОТОКОЛУ UDP

5.1 Вплив зміни буферу передачі

Почнемо знімати показники передачі даних стандартним UDP протоколом в IP мережі (Рис 5.1) створеній за допомогою GNS3. Для цього будемо використовувати WireShark.

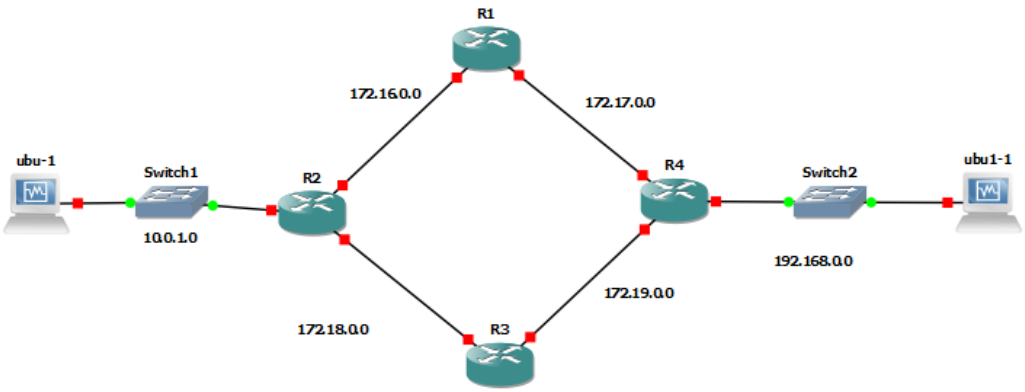


Рисунок 5.1 Мережа для тестування стандартного та модифікованого UDP протоколу

Стандартний буфер відправки має розмір 4096 Байт [10] . Спробуємо передати текстове повідомлення “Hello” по стандартному UDP протоколу з віртуальної машини ubu1 на ubu та вирахувати час за котрий приходить повідомлення.

4	36.042074050	192.168.0.2	10.0.1.2	UDP
6	36.104130860	192.168.0.2	10.0.1.2	UDP

Рисунок 5.2 Показники зняті за допомогою Wireshark стандартного UDP

Отже, час доставки повідомлення 0,06205681сек. Спробуємо зробити те ж саме з розміром буферу в 1024 байт.

3	19.568768849	192.168.0.2	10.0.1.2	UDP
5	19.639093417	192.168.0.2	10.0.1.2	UDP

Рисунок 5.3 Показники модифікованого UDP

Видно що зміна буферу відправки не впливає на час доставки повідомлення малого розміру. В обох випадках packet loss = 0%. Спробуємо передати повідомлення у вигляді текстового документу “files.txt” розміром 28297 Байт. Спочатку передамо стандартним протоколом UDP та визначимо час за який файл буде передано від ubu1 на ubu.

Повідомлення було надіслано без втрат за час 3.47 сек з моменту відправки джерелом першої датаграми та отриманням останньої адресатом. Але повідомлення було фрагментовано маршрутизатором (Рис 5.4).

No.	Time	Source	Destination	Proto	Length	Info
117	479.408443181	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
118	479.408561152	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
119	479.408666668	192.168.0.2	10.0.1.2	UDP	1178	56723 → 7777 Len=4096
120	479.472711321	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
121	479.472726012	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
122	479.472927154	192.168.0.2	10.0.1.2	UDP	1178	56723 → 7777 Len=4096
123	479.473019037	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
124	479.473128290	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
125	479.473251878	192.168.0.2	10.0.1.2	UDP	1178	56723 → 7777 Len=4096
126	479.473347371	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
127	479.473448614	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
128	479.473543707	192.168.0.2	10.0.1.2	UDP	1178	56723 → 7777 Len=4096
129	479.504953713	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
130	479.504966263	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
131	479.504967741	192.168.0.2	10.0.1.2	UDP	1178	56723 → 7777 Len=4096
132	479.536954488	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17
133	479.536967525	192.168.0.2	10.0.1.2	IPv4	1514	Fragmented IP protocol (proto=UDP 17

Рисунок 5.4 Фрагментація повідомлення стандартного UDP протоколу

Зробимо те ж саме з буфером відправки 1450 Байт. Повідомлення було відправлено без втрат за 3.27сек.

No.	Time	Source	Destination	Protoc	Length	Info
359	1334.9952142…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
360	1334.9952678…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
361	1334.9952898…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
362	1334.9953076…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
363	1334.9953257…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
364	1334.9954768…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
365	1334.9956154…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
366	1334.9956392…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
367	1334.9956601…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
368	1334.9957867…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
369	1334.9958202…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
370	1334.9958828…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
371	1334.9959582…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
372	1334.9959877…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
373	1334.9960125…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
374	1334.9960364…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450
375	1334.9960859…	192.168.0.2	10.0.1.2	UDP	1492	60453 → 7777 Len=1450

Рисунок 5.5 Відправлення files.txt модифікованим UDP з буфером відправки 1450 Байт

Видно, що жодна датаграма не була фрагментована(Рис 5.5) а передача повідомлення була пришвидшена на 0.2сек або 5.76% у порівнянні з стандартним протоколом UDP.

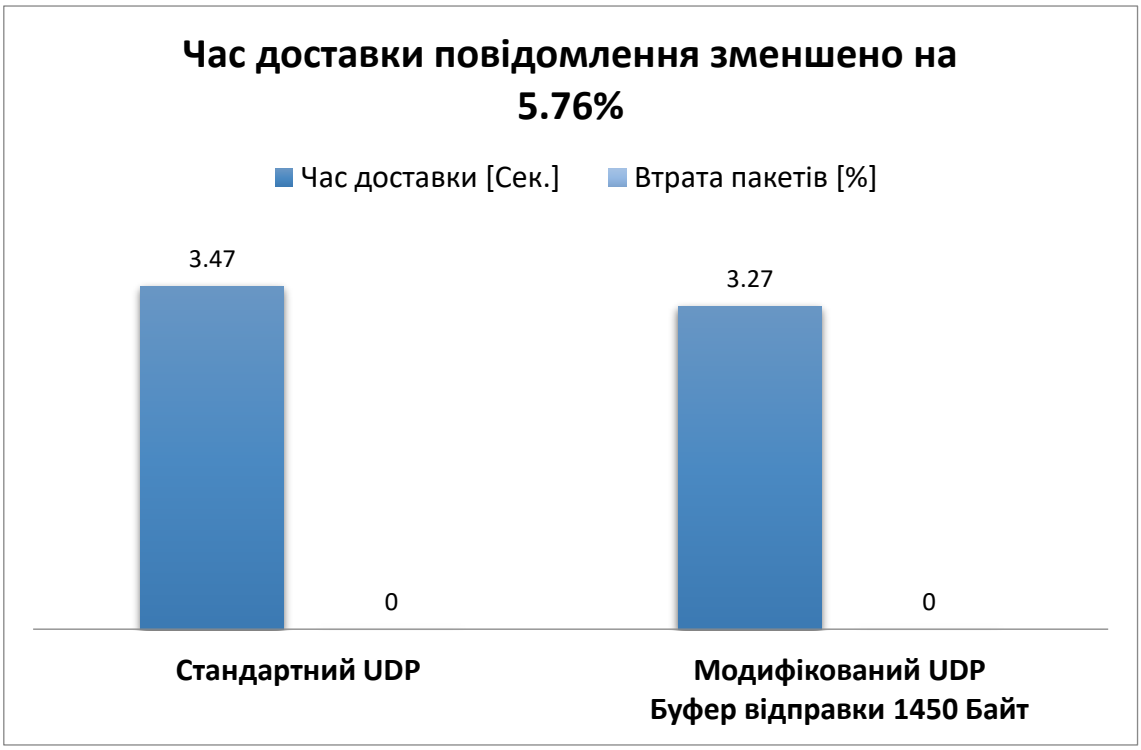


Рисунок 5.6 Передача текстового документу стандартним та модифікованим UDP протоколами

В обох випадках packet loss = 0%. Можна приступати до відправки ще більших за розміром повідомлень по мережі , наприклад, картинки. Будемо відправляти по мережі зображення розміром 469 283 Байт. Спочатку відправимо його стандартним протоколом UDP. Повідомлення було надіслано за 5.47сек., але при цьому packet loss = 60.8%. Було надіслано 115 датаграм, з них до адресата надійшло лише 45.

Виконаємо цей самий дослід з розміром буферу відправки 1450 Байт. Зображення було надіслано за 4.86 сек, при цьому packet loss = 55.55%. Було відправлено 324 датаграми з яких до адресата дійшло 144. Таким чином, вдалося зменшити час доставки повідомлення на 11.15% при цьому з меншими втратами пакетів в мережі, оскільки не було фрагментації. Але такі втрати недопустимі для передачі забражень або відео, приступимо до зміни такого параметра як packet pacing.

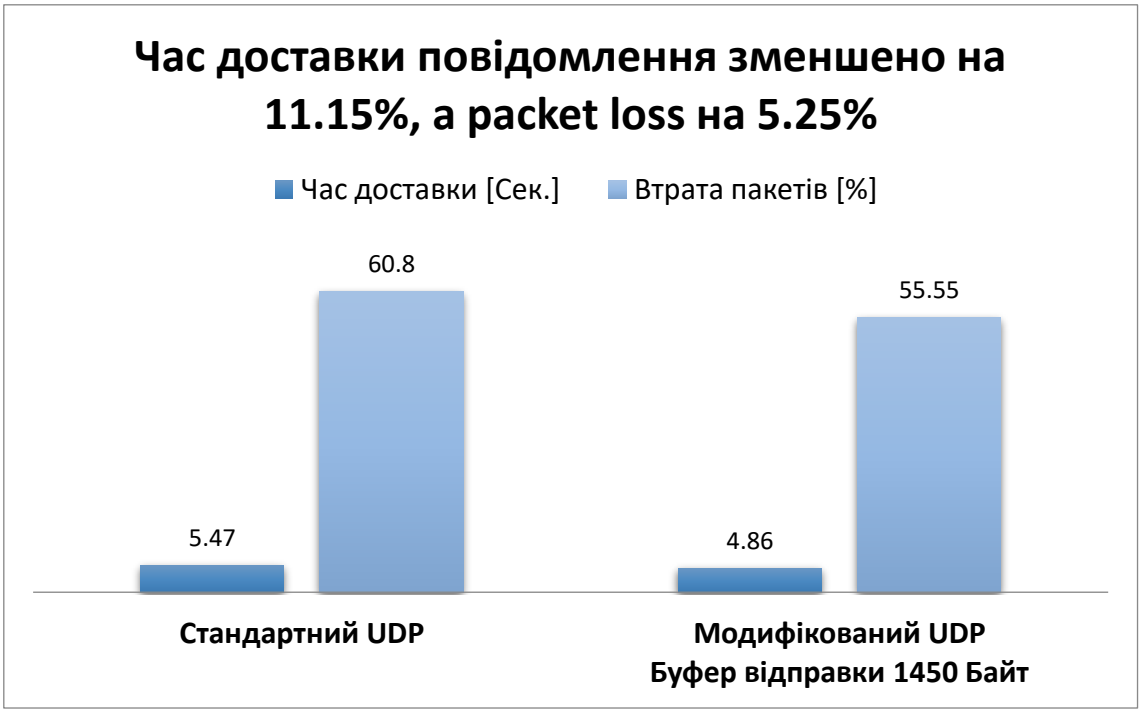


Рисунок 5.7 Передача зображення стандартним та модифікованим UDP протоколами

Отже, завдяки результатам досліду з текстовим повідомленням, можна зробити висновок, що розмір буферу відправки не впливає на швидкість передачі даних, розмір яких не перевищує 1400 Байт. В цьому досліді швидкість передачі була майже однаковою, а packet loss = 0%. При відправці даних більшого розміру(картинка), завдяки зміні буферу відправки, модифікований протокол UDP швидше передавав дані на 11.15% при цьому packet loss був на 5.25% нижче у порівнянні з стандартним UDP протоколом.

5.2 Вплив packet passing при передачі

З попередніх дослідів видно, що модифікований протокол UDP має виграш у швидкості (при цьому з меншими втратами в мережі) над стандартним протоколом. Стандартний протокол UDP не враховує такий параметр як packet pacing, тому порівнювати його з модифікованим протоколом вже не має сенсу, оскільки модифікований протокол показав кращий результат в дослідах зі зміненим буфером.

Не зважаючи на те, що лише за рахунок зміни буферу модифікований протокол показав себе краще, packet loss мав дуже високий відсоток. Отже, необхідно визначити оптимальний параметр packet pacing при якому швидкість передачі буде високою, а packet loss буде низьким.

Packet pacing це час очікування між відправкою кожної з датаграм. Іншими словами, взявши цей параметр за 0.01сек., відправивши першу датаграму по мережі, кожна наступна датаграма буде відправлена через 0.01сек..

Проведемо дослід роботи модифікованого протоколу UDP з packet pacing = 0.01сек. Будемо передавати те ж саме зображення розміром 469 283 Байт. Повідомлення було прийнято адресатом за 4.9сек. було відправлено 324

датаграм (Рис. 5.6), а прийнято сервером 173 (Рис5.7). Таки чином, майже при однаковій швидкості передачі packet loss було знижено до 46.6%.

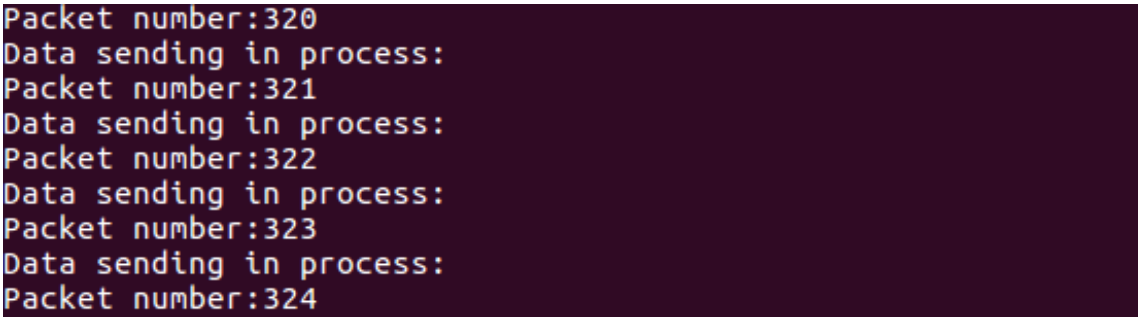


Рисунок 5.8 Кількість відправлених датаграм

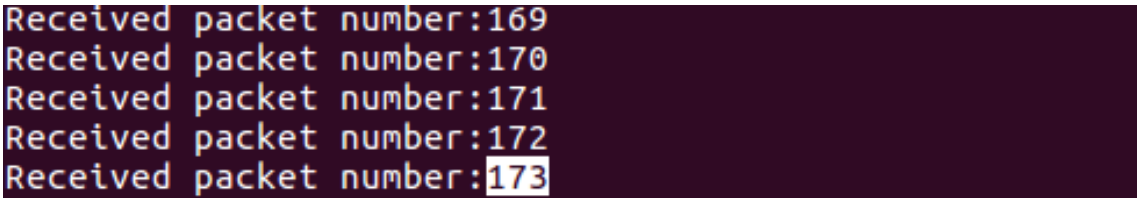


Рисунок 5.9 Кількість датаграм отриманих сервером

Якщо збільшити packet pacing до 0.02сек. це означає що це ж саме зображення з 324 датаграм буде відправлено не раніше ніж за 324*0.02 = 6.48сек., що значно сповільнить роботу модифікованого протоколу UDP. Таким чином, було прийнято рішення спробувати відправляти пакети по мережі пачками тоб-то для першого досліду при відправці кожних 50 датаграм робити паузу на 0.2сек.. Проведемо дослід з тим самим зображенням. Було відправлено 324 датаграми з них 139 надійшло до адресата (Рис. 5.8) за час 4.38 сек..

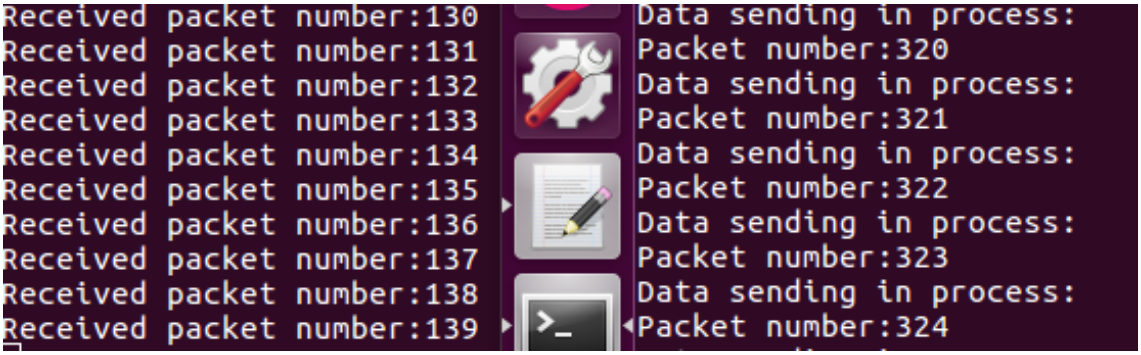


Рисунок 5.10 Отримані та відправлені датаграми модифікованим протоколом UDP

При цьому packet loss 57.09%. Спробуємо виконати цей самий дослід з іншим значенням параметру packet pacing. Будемо робити паузу в 0.25с між кожними 50 датаграмами та 0.01сек між кожною датаграмою.

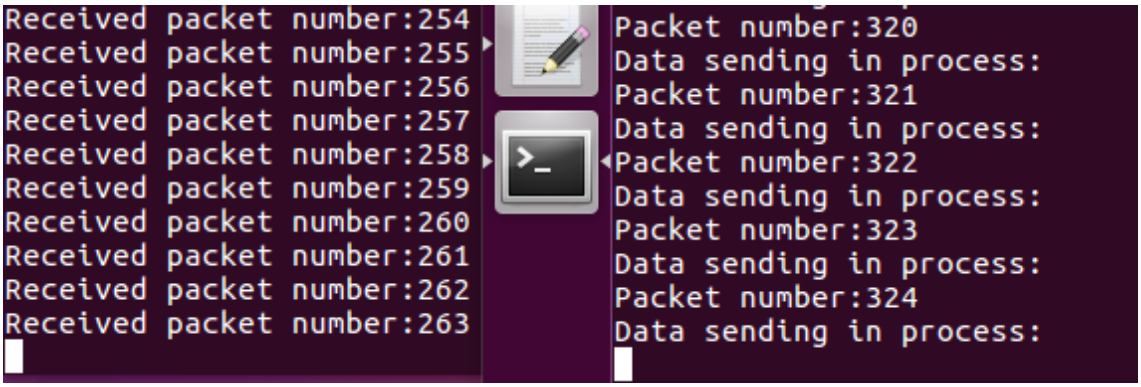


Рисунок 5.11 Отримані та відправлені датаграми модифікованим протоколом UDP з параметром packet pacing =0.25 сек. на кожні 50 датаграм та 0.01 сек. між кожною датаграмою

Повідомлення було надіслано за 5.12с при цьому packet loss = 18.8%. Видно що параметр packet pacing значно впливає на втрату пакетів по мережі. Можна приступати до вибору оптимальних параметрів передачі.

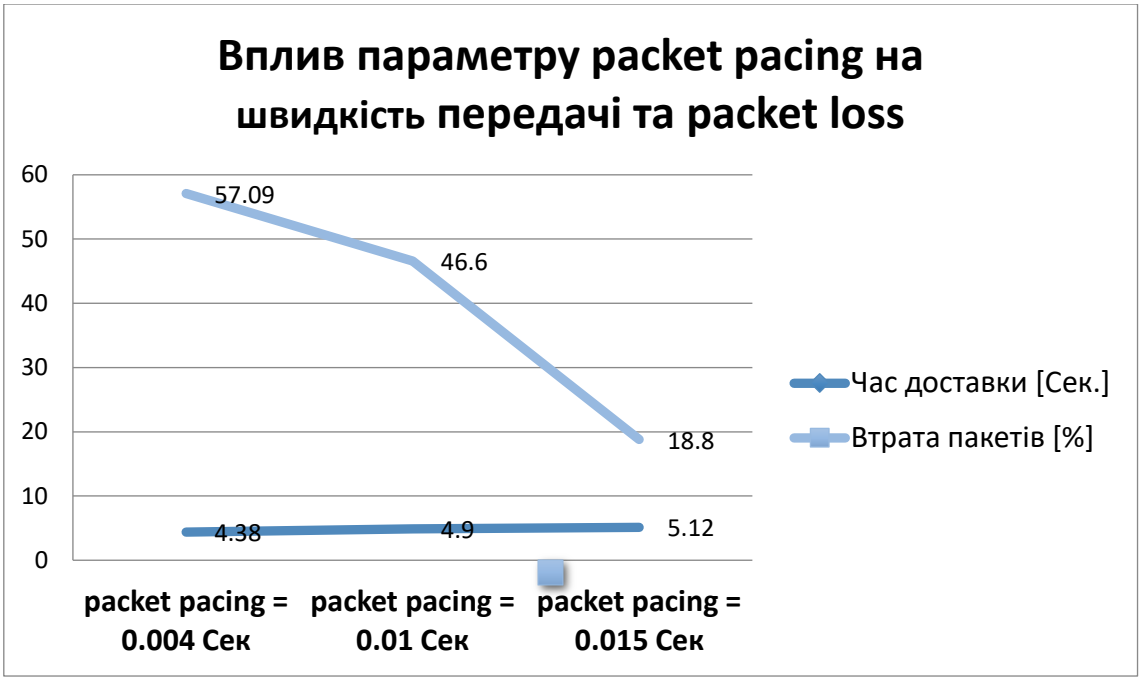


Рисунок 5.12 Залежність часу доставки та втрати пакетів модифікованого протоколу UDP від параметру packet pacing

5.3 Оптимальні показники модифікованого протоколу UDP при передачі

Оптимальні показники параметрів це ті показники при яких швидкість передачі висока, а packet loss мінімальний. В попередніх дослідах було показано як впливають на передачу такі показники як: буфер відправки та packet pacing.

Проведемо новий дослід, але будемо передавати не картинку а відео. Встановимо буфер відправки розміром 1500 Байт, а packet pacing 0.015 сек. Передавати будемо відео потік з сумарним розміром 136657 Байт, котрий має назву “video.mp4” та спробуємо визначити оптимальні параметри для передачі. Але перед тестуванням модифікованого протоколу спробуємо передати це ж саме відео стандартним протоколом UDP.

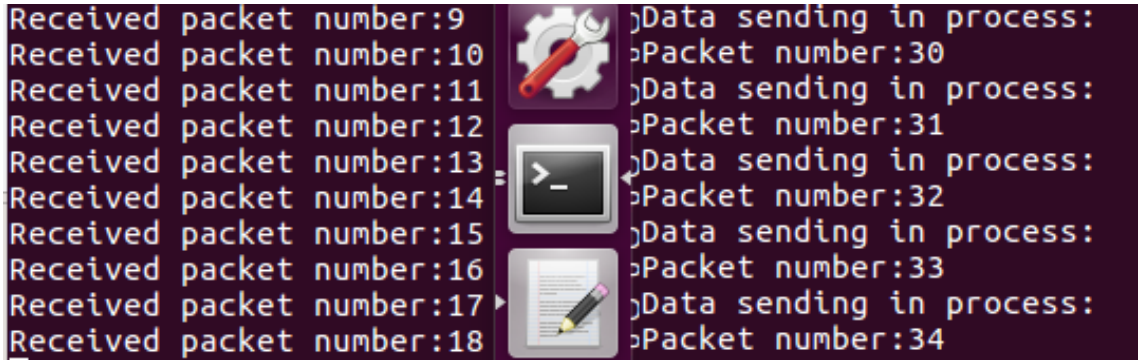


Рисунок 5.13 Відправка відео потоку стандартним протоколом UDP.

З віртуальної машини ubu1 було надіслано 34 датаграми, до адресата надійшло лише 18 датаграм за час 2.8с, при цьому packet loss = 47.05%. Така втрата пакетів в мережі не є допустимою.

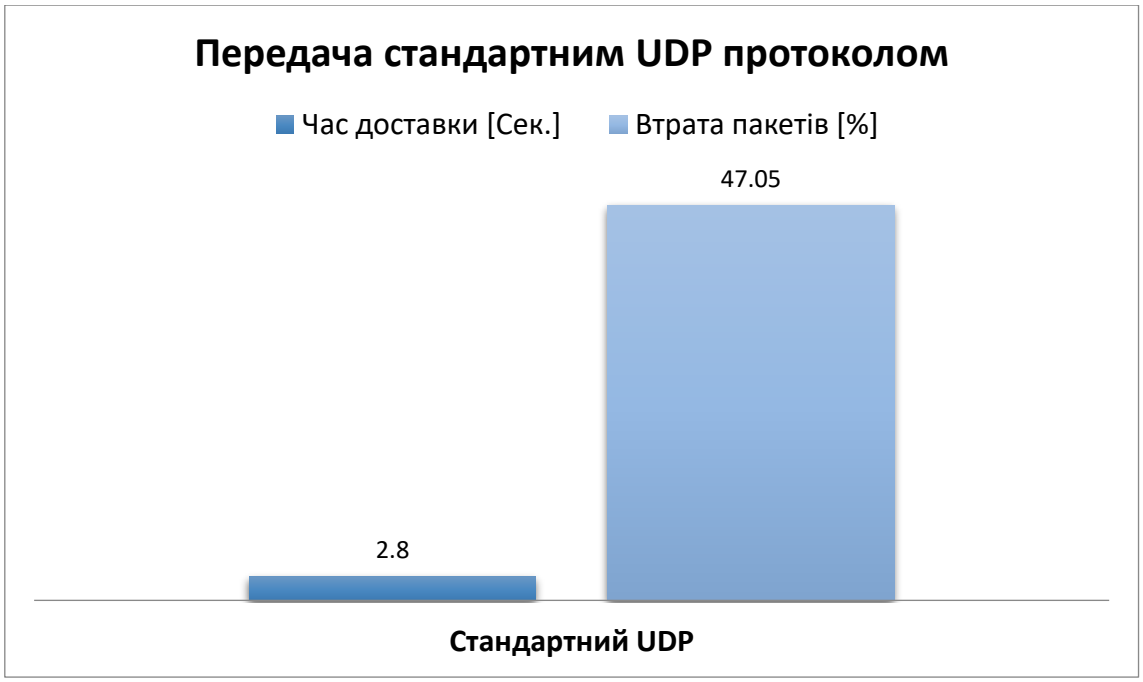


Рисунок 5.14 Показники передачі стандартним протоколом UDP

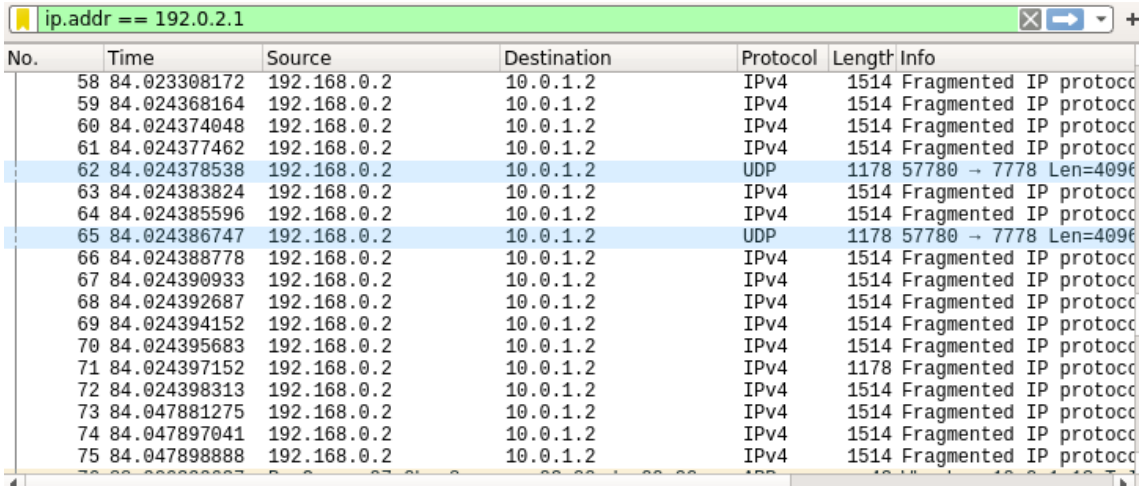


Рисунок 5.15 Початок фрагментації при відправці стандартним протоколом UDP

Кожна датаграма надіслана стандартним протоколом UDP була фрагментована, це сповільнює швидкість передачі та підвищує ймовірність втрати пакетів в мережі.

Проведемо цей самий дослід з модифікованим протоколом UDP. Буфер відправки матиме розмір 1450 Байт, а packet pacing = 0.015с.

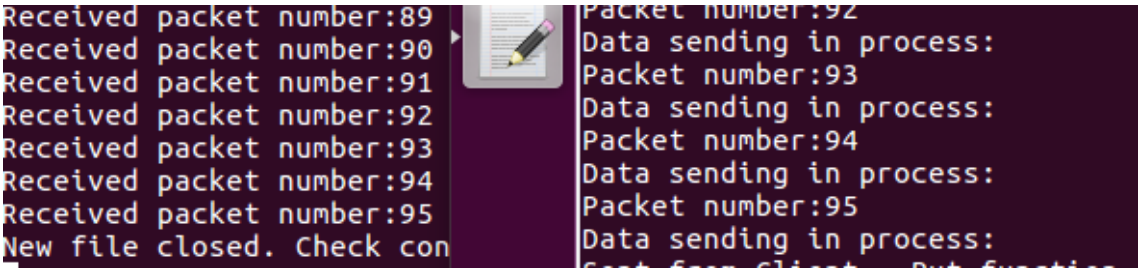
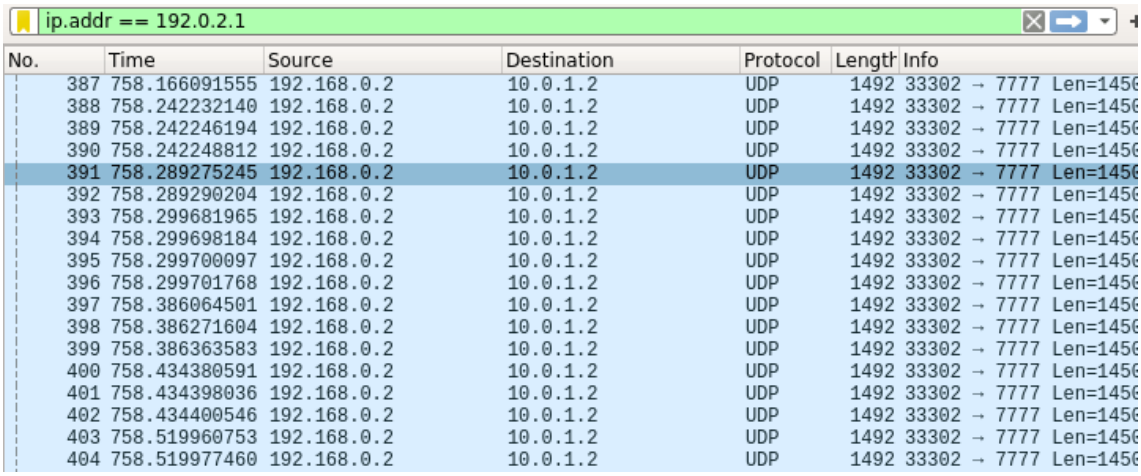


Рисунок 5.16 Відправка відео потоку модифікованим протоколом UDP

Повідомлення було надіслано за час 3.4сек., при цьому packet loss = 0%. Жодна з датаграм не була фрагментована.



ip.addr == 192.0.2.1

No.	Time	Source	Destination	Protocol	Length	Info
387	758.166091555	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
388	758.242232140	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
389	758.242246194	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
390	758.242248812	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
391	758.289275245	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
392	758.289290204	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
393	758.299681965	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
394	758.299698184	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
395	758.299700097	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
396	758.299701768	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
397	758.386064501	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
398	758.386271604	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
399	758.386363583	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
400	758.434380591	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
401	758.434398036	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
402	758.434400546	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
403	758.519960753	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456
404	758.519977460	192.168.0.2	10.0.1.2	UDP	1492	33302 → 7777 Len=1456

Рисунок 5.17 Скріншот Wireshark отримувача датаграм відправлених модифікованим протоколом UDP.

Спробуємо пришвидшити швидкість передачі змінивши параметр packet pacing до 0.002с. Та проведемо цей дослід ще раз.

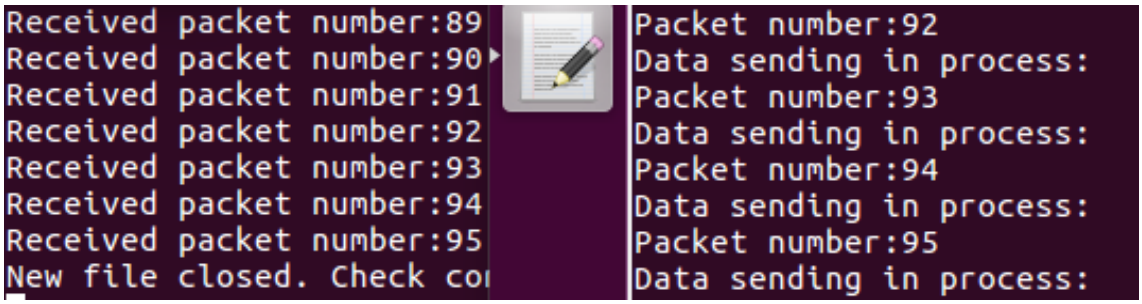


Рисунок 5.18 Відправка модифікованим протоколом UDP з параметром packet pacing 0.002сек.

Зі зміненим параметром packet pacing вдалося надіслати повідомлення за час 2.23с. Було відправлено 95 датаграм з них 95 надійшло до отримувача.

Таким чином, вдалося зменшити швидкість передачі у порівнянні з стандартним протоколом на 20.35% та при цьому здійснити передачу без втрат. Проведемо дослід з параметром packet pacing = 0.001 сек.. При передачі 95 датаграм було втрачено в мережі 1, це означає, що зменшувати цей параметр більше не має сенсу, оскільки з’являються втрати.

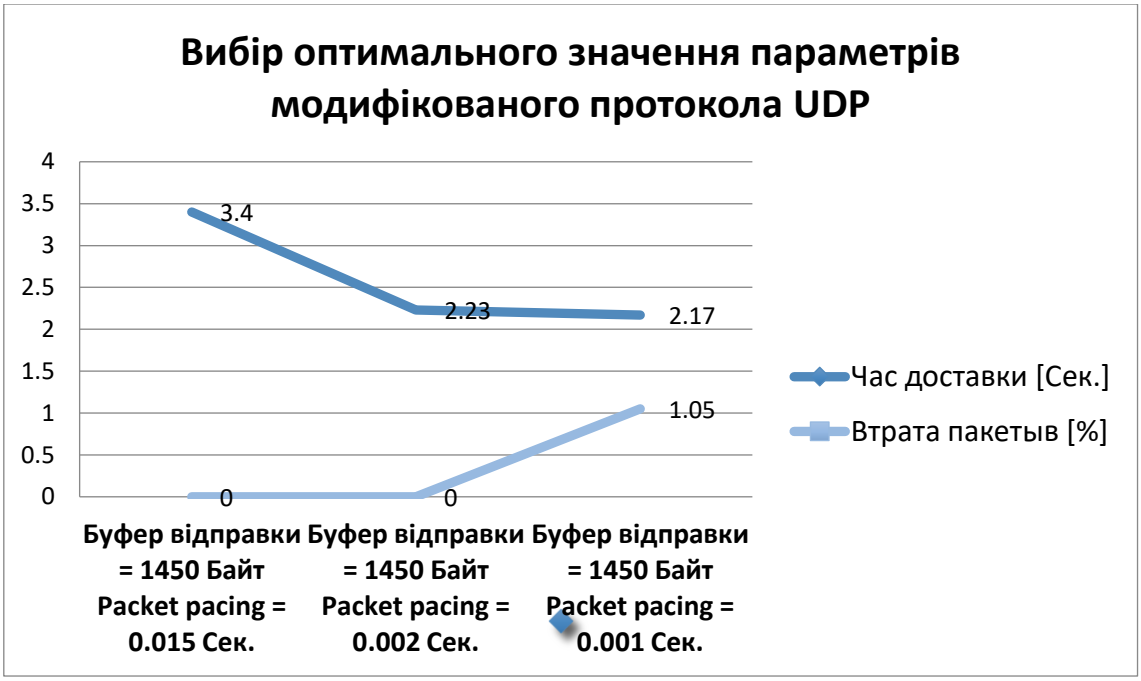


Рисунок 5.19 Проведення дослідів для того щоб обрати оптимальні параметри модифікованого протоколу UDP.

Отже, можна зробити висновок, що для мережі на котрій були проведені досліди оптимальними параметрами є буфер відправки розміром 1450 Байт, а packet pacing 0.002сек. Розмір датаграми не повинен перевищувати 1500 Байт, щоб не почалася фрагментація. Можна приступати до порівняння модифікованого протоколу UDP з TCP.

5.4 Оцінка ефективності модифікованих параметрів протоколу UDP в порівнянні з TCP

Після визначення оптимальних параметрів передачі проведемо порівняння модифікованого протоколу UDP з TCP. Проведемо дослід з передачею відео та картинки для обох протоколів. Розпочнемо з передачі зображення котре має назву “1.jpeg” та розмір 86661 Байт. Спочатку передамо по TCP.

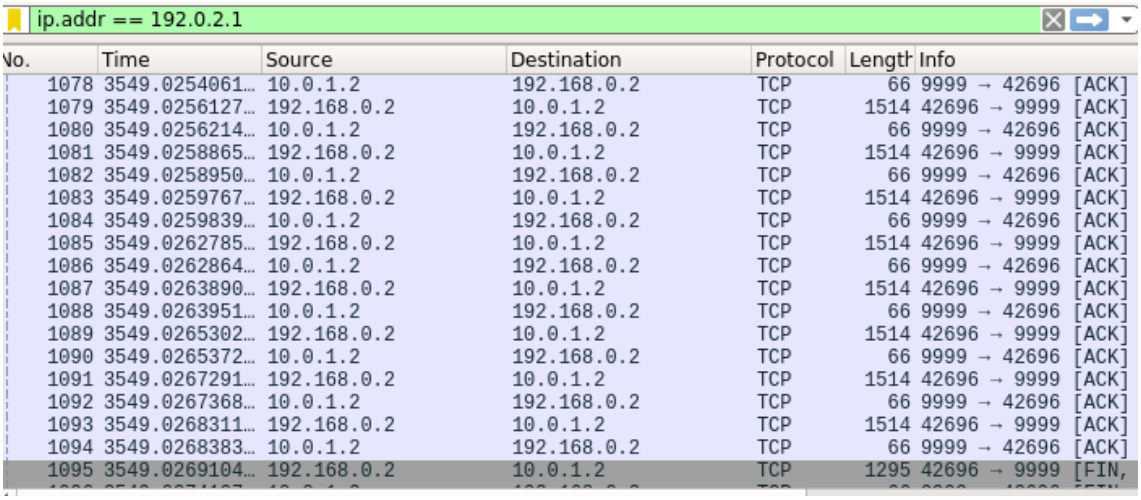


Рисунок 5.20 Відправка зображення 1.jpeg протоколом TCP.

Повідомлення було надіслано за час 1.21сек., фрагментації не було, оскільки протокол TCP враховує розмір MTU на відміну від стандартного UDP протоколу. Проведемо той самий дослід з модифікованим протоколом UDP.

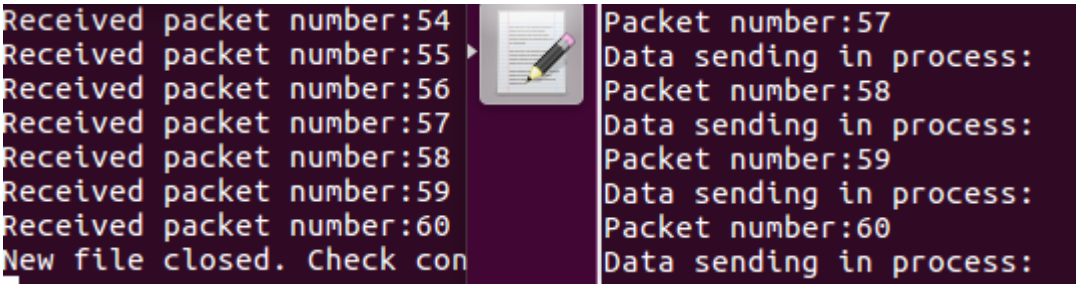


Рисунок 5.21 Відправка зображення 1.jpeg модифікованим протоколом UDP.

Теж саме зображення було надіслано до адресата за 0.85сек. тоб-то на 29.75% швидше, при цьому втрачених пакетів не було.

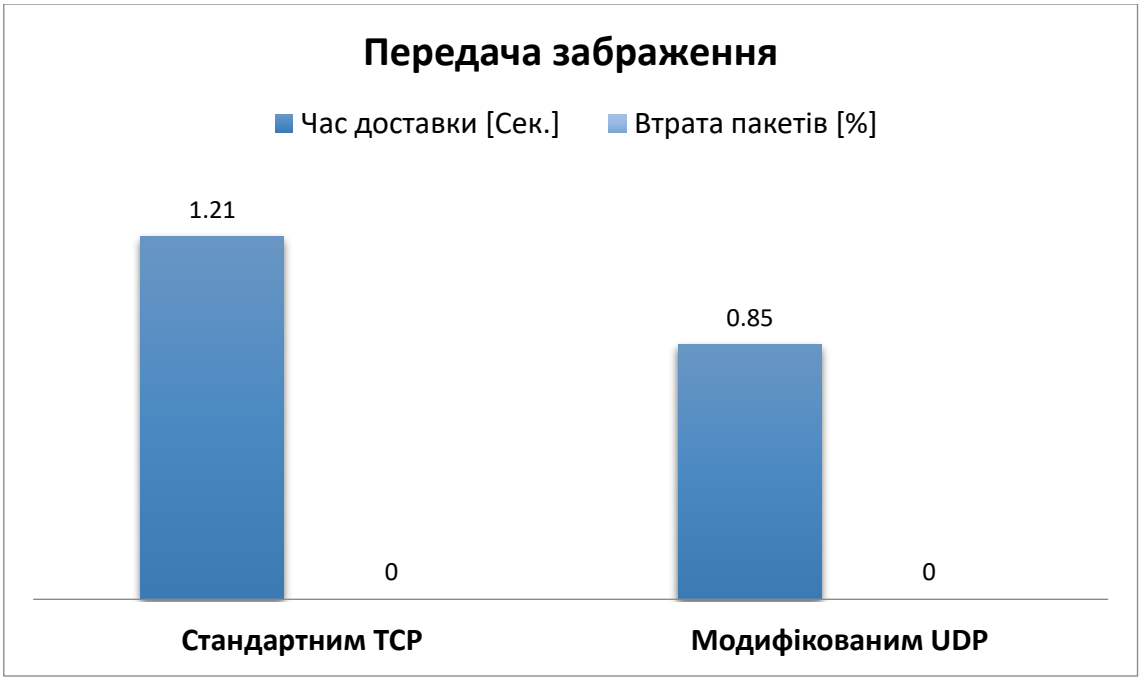
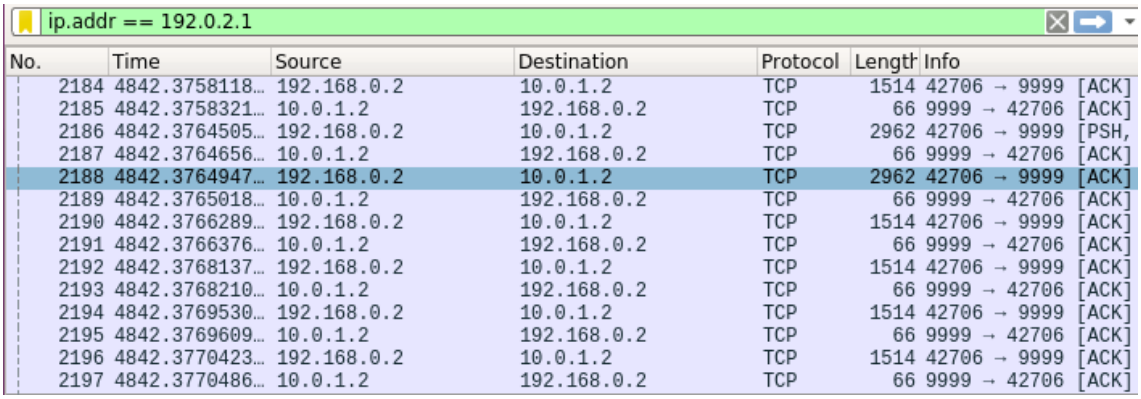


Рисунок 5.22 Передача зображення модифікованим протоколом UDP та стандартним TCP.

Спробуємо передати відео потік з попередніх дослідів ще раз, але тепер по TCP та модифікованому UDP. Розпочнемо передачу по TCP.



No.	Time	Source	Destination	Protocol	Length	Info
2184	4842.3758118…	192.168.0.2	10.0.1.2	TCP	1514	42706 → 9999 [ACK]
2185	4842.3758321…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]
2186	4842.3764505…	192.168.0.2	10.0.1.2	TCP	2962	42706 → 9999 [PSH,
2187	4842.3764656…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]
2188	4842.3764947…	192.168.0.2	10.0.1.2	TCP	2962	42706 → 9999 [ACK]
2189	4842.3765018…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]
2190	4842.3766289…	192.168.0.2	10.0.1.2	TCP	1514	42706 → 9999 [ACK]
2191	4842.3766376…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]
2192	4842.3768137…	192.168.0.2	10.0.1.2	TCP	1514	42706 → 9999 [ACK]
2193	4842.3768210…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]
2194	4842.3769530…	192.168.0.2	10.0.1.2	TCP	1514	42706 → 9999 [ACK]
2195	4842.3769609…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]
2196	4842.3770423…	192.168.0.2	10.0.1.2	TCP	1514	42706 → 9999 [ACK]
2197	4842.3770486…	10.0.1.2	192.168.0.2	TCP	66	9999 → 42706 [ACK]

Рисунок 5.23 Передача відео потоку “video.mp4” по протоколу TCP.

Повідомлення було отримано адресатом за 2.6сек.. Повторимо дослід з модифікованим протоклом UDP. Повідомлення було доставлено до отримувача за 2.19сек. Жодна з 95 датаграм не втратилась в мережі, але при цьому передача здійснена швидше на 15.76%.

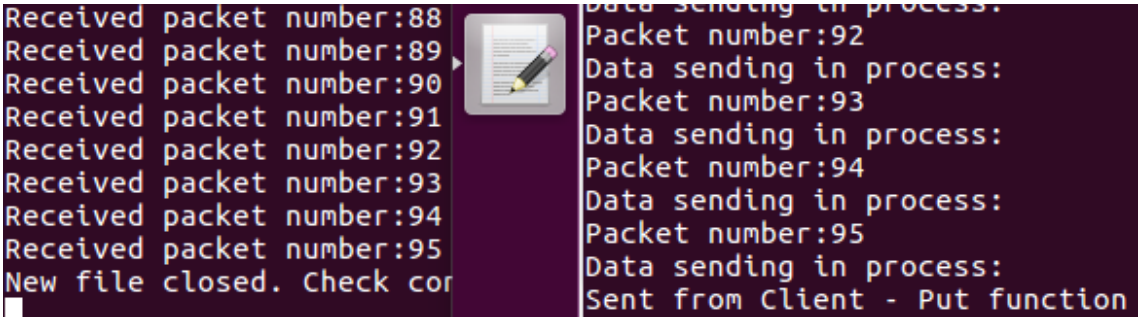


Рисунок 5.24 Передача відео потоку “video.mp4” по модифікованому протоколу UDP.

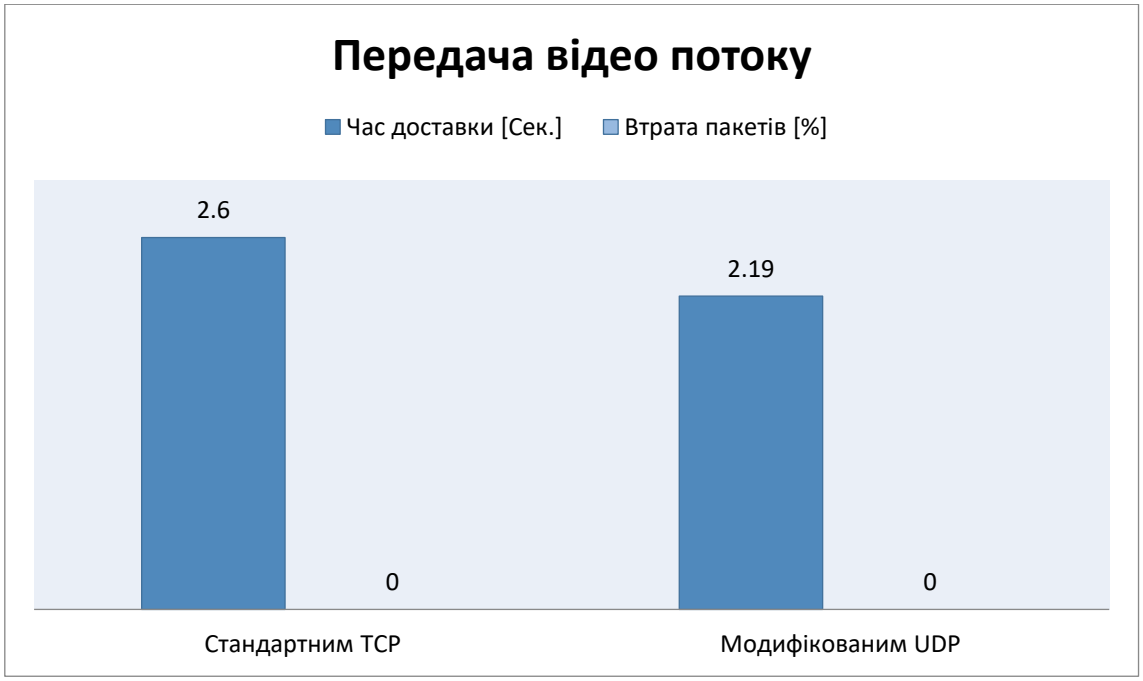


Рисунок 5.25 Передача відео потоку стандартним TCP та модифікованим UDP протоколами.

Проведемо цей самий дослід, але додамо навмисні втрати в мережі packet loss = 1% на підключенні між ubu та Switch1, та повторимо цей дослід знову.

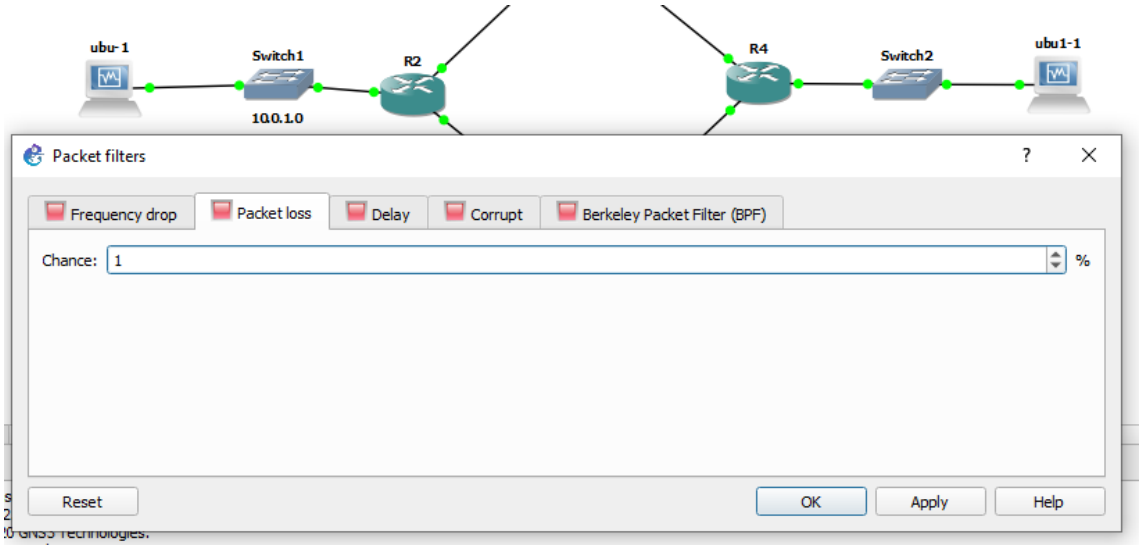


Рисунок 5.26 Додавання packet loss до мережі.

Спочатку, передамо по TCP.

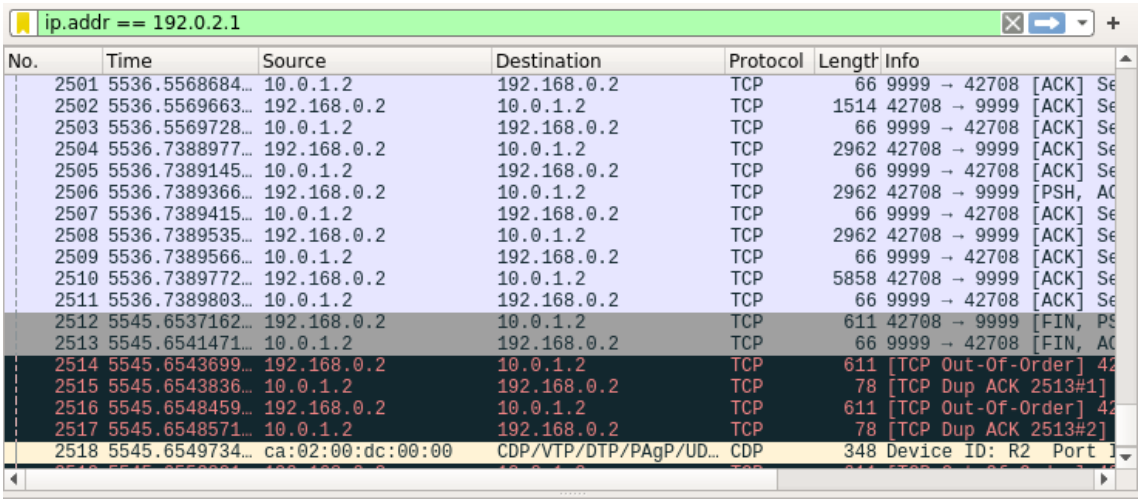


Рисунок 5.27 Передача відео потоку “video.mp4” по протоколу TCP в мережі з втратами.

TCP протокол, при передачі в мережі з ймовірністю packet loss в 1%, сповільнив передачу на 4.94сек. Тоб-то, відео котре було передано в попередньому досліді за 2.6 сек, в цьому досліді було передано за 7.54 сек. Швидкість передачі була значно сповільнена, оскільки протокол TCP не може

відправляти нові дані по мережі, до тих пір, поки не отримає підтвердження на старі. Це і є головний недолік TCP протоколу. Проведемо той самий дослід з модифікованим протоколом UDP.

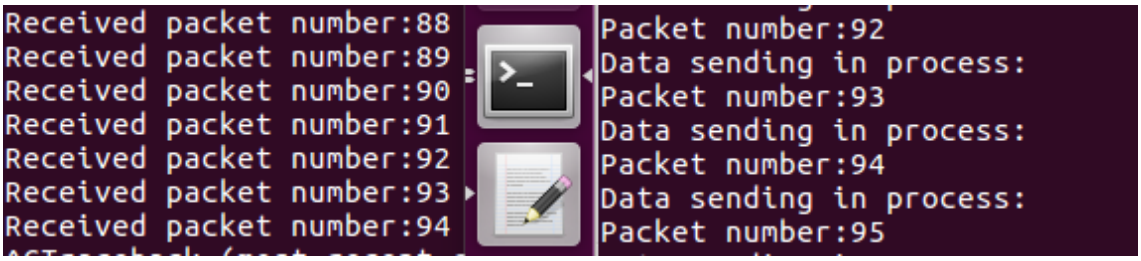


Рисунок 5.28 Передача відео потоку “video.mp4” по протоколу UDP в мережі з втратами.

Було надіслано 95 датаграм з яких до отримувача надійшло 94 за час 2.22. Packet loss = 1.05%, але при цьому швидкість передачі майже в три з половиною раза швидше ніж TCP.

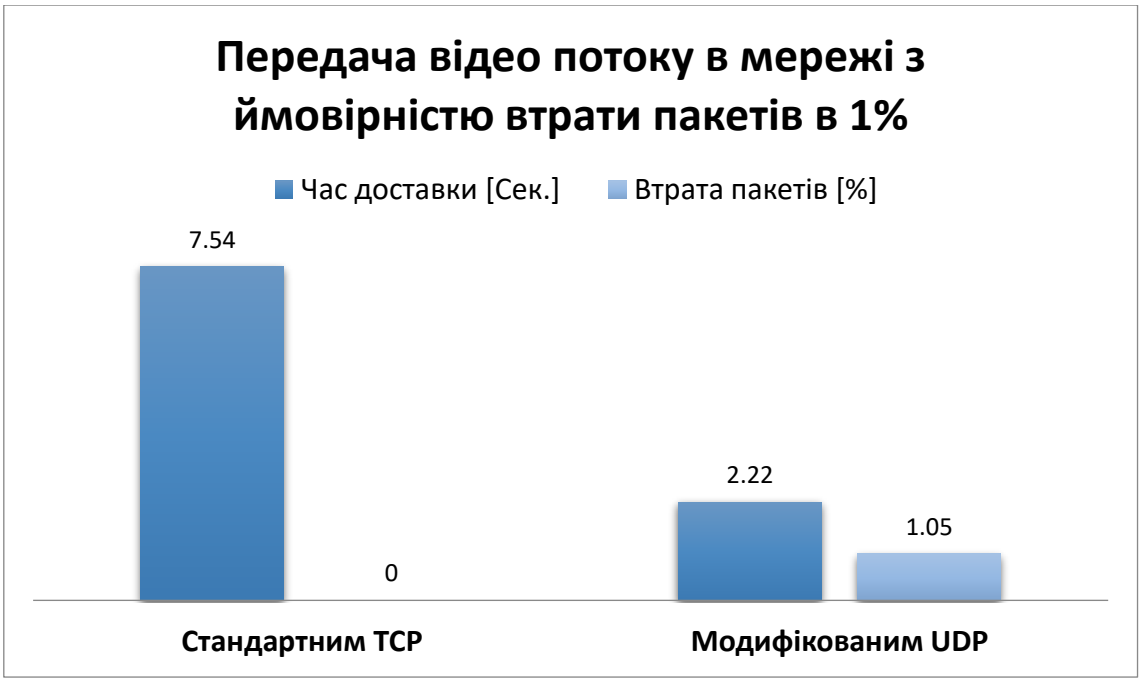


Рисунок 5.29 Передача відео потоку стандартним TCP та модифікованим UDP в мережі з втратами.

Отже, можна зробити висновок, що протокол UDP слід використовувати для даних, для яких втрата пакетів не така важлива як затримка. Також було наглядно продемонстровано переваги у швидкості модифікованого протоколу UDP у порівнянні з TCP. Та вказано головний недолік протоколу TCP.

5.5 Висновки до розділу 5

Цей розділ присвячено порівнянню модифікованого протоколу UDP з стандартним UDP. Було експерементально підтверджено вплив буферу відправки, MTU та packet pacing на швидкість передачі даних по мережі. Модифікувавши ці параметри, вдалося пришвидшити швидкість передачі повідомлень, при цьому з меншими втратами в мережі або взагалі без втрат. Також було порівняно модифікований протокол UDP з TCP та встановлено, що швидкість передачі даних по модифікованому UDP швидша, особливо це видно в мережах з втратами. Де UDP передав дані майже в 3.5 рази швидше.

ВИСНОВОК

Отже, побудува універсальної фізичної мережі з однотипної апаратури практично неможлива, тому що така мережа не могла б задовольняти потреби всіх користувачів. З одного боку, потрібна високошвидкісна мережа для з'єднання мережевих машин в межах будівлі, а з іншого - надійні комунікаційні мережі між комп'ютерами, рознесеними на сотні кілометрів. Попит на високошвидкісну передачу даних постійно зростає. Потреба в цьому може бути знайдена в самих різних галузях - від розваг до наукових досліджень. Однак є декілька проблем, які перешкоджають можливостям мережевих пристроїв. Одним з них є повільна обробка пакетів через значні накладні витрати на системні виклики для простих мережевих операцій. Існують апаратні рішення, але з економічної точки зору кращим є використання застарілого обладнання через високу вартість поновлення мережевої інфраструктури.

Робота має велику практичну цінність, оскільки значну її частину становить саме програмне вирішення проблем зі швидкістю. Рішення полягає у визначенні параметрів протоколу транспортного рівня, котрі відповідають за швидкість передачі та зміна цих параметрів для досягнення вищої швидкості передачі повідомлень, при цьому з мінімальними втратами пакетів в мережі.

В дипломній роботі описано метод зменшення часу доставки повідомлення в IP-мережі за рахунок модифікації протоколу UDP. Було визначено параметри протоколу UDP, котрі відповідають за швидкість передачі повідомлення та було модифіковано протокол UDP шляхом зміни цих параметрів. Також було додано параметр packet pacing, котрий стандартний протокол UDP не враховує. Додавання цього параметру допомогло значно знизити втрати пакетів в мережі “ packet loss” при цьому швидкість доставки повідомлень залишилась майже не змінною.

В практичній частині дипломної роботи, було побудовано невелику віртуальну приватну мережу в графічному мережевому стимуляторі GNS3 та

підключено до мережі два кінцевих пристрої (ПК) з операційною системою Ubuntu 16.04. Після чого на комп’ютери було встановлено додатки сервера та клієнта для передачі даних по UDP, розроблені на мові програмування python3. Було проведено досліди передачі повідомлень з одного комп'ютера на інший з різними параметрами модифікованого протоколу UDP. Результати досліду показали, що буфер відправки не впливає на швидкість передачі повідомлень розмір котрих не первищує 1450 Байт. Після проведення дослідів з даними більшого розміру, таких як: текстовий документ, зображення, відео потік, було отримано виграш у швидкості передачі з меншими втратами пакетів в мережі.

Модифікувавши протокол UDP, вдалося пришвидшити передачу текстового документу на 5.76%, зображення на 11.15% у порівнянні з стандартним протоколом UDP. Після чого було проведено порівняння швидкості передачі модифікованим UDP протоколом з TCP. Результати дослідів показали, що модифікований UDP мав виграш у швидкості при передачі зображення на 29.75% та при передачі відео потоку на 15.76%.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1 - TCP/IP Архитектура, протоколы, реализация *Фейт С. Издано: 2000, М., Лори* – ст. 11-15

2 - UDP/IP For Embedded System: Methods, Implementation, Benchmarks, Programming,FPGA, Embedded system Paperback – October 17, 2010 by Muhammad Asif

3 - [Електронний ресурс] – Режим доступу: https://en.wikibooks.org/wiki/Communication_Networks/TCP_and_UDP_Protocols

4 - [Електронний ресурс] – Режим доступу: https://sites.google.com/site/setevyetehnologiisgaki/setevye-tehnologii/setevye-servisy-internet/3-protokol-ip

5 - [Електронний ресурс] – Режим доступу: https://www.opensignal.com/reports/2020/05/russia/mobile-network-experience

6 - [Електроннийресурс] – Режим доступу: https://m.habr.com/ru/company/oleg-bunin/blog/461829/

7 - Web Protocols and Practice HTTP/1.1, Networking Protocols, Caching, and Traffic Measurement Balachander Krishnamurthy Jennifer Rexford 2002 ст. 86-88

8 - Towards a Deeper Understanding of TCP BBR Congestion ControlPublisher: IEEE Dominik Scholz ; Benedikt Jaeger ; Lukas Schwaighofer ; Daniel Raumer ; Fabien Geyer ; Georg Carle 2016 – ст. 12-13

9 – Метод зменшення часу доставки повыдомлень в IP мережы за рахунок модифікації протоколу UDP Маньківський В.Б., Педько А.Д. – ст. 4

10 - UNIX Разработка сетевых приложений У. Р. Стивенс, Б. Феннер, Э. М. Рудофф 3-е издание 2003- ст.209-213

Додаток 1

```
import socket
import sys
import time

what = input('Enter wich data you want to send (txt, message or file: ')
if what == 'message':
   host = '192.168.0.2'
   port = 7777
   client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
   msg = input('Enter message to send: ')
   client.sendto(msg.encode('utf-8'), ('10.0.1.2', 7777))
   d = client.recvfrom(1024)
   reply = d[0]
   addr = d[1]
   print ('Server reply: ' + reply.decode('utf-8'))
   client.close()
elif what == 'txt':
   UDP_IP = '192.168.0.2'
   UDP_PORT = 7777
   buf = 1024
   file_name = str(input(" Name of the documend: "))


   sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
   sock.sendto(file_name.encode(), ('10.0.1.2', 7777))
   print('Sending %s' % file_name)

   f = open(file_name, 'r')
   data = f.read(buf)
   while(data):
      if(sock.sendto(data.encode(), ('10.0.1.2', 7777))):
         data = f.read(buf)
         time.sleep(0.02)
   sock.close()
   f.close()
elif what == 'file':
   def checkArg():

      if len(sys.argv) != 3:
         print(
         "ERROR. Wrong number of arguments passed. System will exit. Next time please supply 2
arguments!")
         sys.exit()
      else:
         print("2 Arguments exist. We can proceed further")



   def checkPort():
```

```
    if int(sys.argv[2]) <= 5000:
      print(
      "Port number invalid. Port number should be greater than 5000 else it will not match with Server port. Next time enter valid port.")
      sys.exit()
    else:
      print("Port number accepted!")

checkArg()
try:
  socket.gethostbyname(sys.argv[1])
except socket.error:
  print("Invalid host name. Exiting. Next time enter in proper format.")
  sys.exit()

host = sys.argv[1]
try:
  port = int(sys.argv[2])
except ValueError:
  print("Error. Exiting. Please enter a valid port number.")
  sys.exit()
except IndexError:
  print("Error. Exiting. Please enter a valid port number next time.")
  sys.exit()

checkPort()


try:
  s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
  print("Client socket initialized")
  s.setblocking(0)
  s.settimeout(15)
except socket.error:
  print("Failed to create socket")
  sys.exit()



while True:
  command = input(
    "Please enter a command: \n1. get [file_name]\n2. put [file_name] ")


  CommClient = command.encode('utf-8')

  try:
    s.sendto(CommClient, (host, port))
  except ConnectionResetError:
    print(
```

```
    "Error. Port numbers are not matching. Exiting. Next time please enter same port numbers.")
    sys.exit()

CL = command.split()
print(
"We shall proceed, but you may want to check Server command prompt for messages, if any.")

if CL[0] == "get":
    print("Checking for acknowledgement")
    try:
        ClientData, clientAddr = s.recvfrom(51200)
    except ConnectionResetError:
        print(
        "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
        sys.exit()
    except:
        print("Timeout or some other error")
        sys.exit()
    text = ClientData.decode('utf8')
    print(text)
    print("Inside Client Get")

    try:
        ClientData2, clientAddr2 = s.recvfrom(51200)
    except ConnectionResetError:
        print(
        "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
        sys.exit()
    except:
        print("Timeout or some other error")
        sys.exit()

    text2 = ClientData2.decode('utf8')
    print(text2)

    if len(text2) < 30:
        if CL[0] == "get":
            BigC = open("Received-" + CL[1], "wb")
            d = 0
            try:
                # number of paclets
                CountC, countaddress = s.recvfrom(1024)
            except ConnectionResetError:
                print(
                "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
                sys.exit()
            except:
                print("Timeout or some other error")
                sys.exit()
```

```
            tillC = CountC.decode('utf8')
            tillCC = int(tillC)
            print("Receiving packets will start now if file exists.")

            while tillCC != 0:
               ClientBData, clientbAddr = s.recvfrom(1024)
               dataS = BigC.write(ClientBData)
               d += 1
               print("Received packet number:" + str(d))
               tillCC = tillCC - 1

            BigC.close()
            print(
            "New Received file closed. Check contents in your directory.")

      elif CL[0] == "put":
         print("Checking for acknowledgement")
         try:
            ClientData, clientAddr = s.recvfrom(1024)
         except ConnectionResetError:
            print(
            "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
            sys.exit()
         except:
            print("Timeout or some other error")
            sys.exit()

         text = ClientData.decode('utf8')
         print(text)
         print("We shall start sending data.")

         if text == "Valid Put command. Let's go ahead ":
            if os.path.isfile(CL[1]):

               c = 0


               size = os.stat(CL[1])
               sizeS = size.st_size

               print("File size in bytes: " + str(sizeS))
               Num = int(sizeS / 1024)
               Num = Num + 1
               print("Number of packets to be sent: " + str(Num))
               till = str(Num)
               tillC = till.encode('utf8')
               s.sendto(tillC, clientAddr)
               tillIC = int(Num)
               GetRun = open(CL[1], "rb")
```

```
            while tillIC != 0:

                time.sleep(0.07)
                Run = GetRun.read(1024)


                s.sendto(Run, clientAddr)
                c += 1
                tillIC -= 1

                print("Packet number:" + str(c))
                print("Data sending in process:")



            GetRun.close()

            print("Sent from Client - Put function")
          else:
            print("File does not exist.")
        else:
          print("Invalid.")


    else:
        try:
          ClientData, clientAddr = s.recvfrom(51200)
        except ConnectionResetError:
          print(
          "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
          sys.exit()
        except:
          print("Timeout or some other error")
          sys.exit()
        text = ClientData.decode('utf8')
        print(text)
  print("Program will end now. ")
  quit()
else:
  print('Inncorect input')
```

Лістинг 1 Код програмного забезпечення клієнта

Додаток 2

```
import socket
import sys
import select
import time
what = input('Which type of data do you want receive (txt, message or file)?: ')
if what == 'message':
    timeout = 60
    host = '10.0.1.2'
    port = 7777
    addr = (host, port)
    server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    server.bind(addr)

    while True:
        print('Waiting for data ({0} seconds)...'.format(timeout))
        server.settimeout(timeout)
        try:
            d = server.recvfrom(1024)
        except socket.timeout:
            print('Time is out. {0} seconds have passed'.format(timeout))
            break
        received = d[0]
        addr = d[1]
        print('From: ', addr )
        print('Received data: ', received.decode('utf-8'))

        msg = input('Enter message to send: ')
        server.sendto(msg.encode('utf-8'), addr)
    server.close()
elif what == 'txt' :
    UDP_IP = '10.0.1.2'
    IN_PORT = 7777
    timeout = 3


    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind((UDP_IP, IN_PORT))

    while True:
        data,addr = sock.recvfrom(1024)
        if data:
            print('File name: ', data.decode('utf-8'))
            file_name = data.strip()

        f = open(file_name, 'wb')

        while True:
```

```
        ready = select.select([sock], [], [],timeout)
        if ready[0]:
            data, addr = sock.recvfrom(1024)
            f.write(data)
        else:
            print('%s Finish!' % file_name.decode('utf-8'))
            f.close()
            break
elif what == 'file':
    def checkArg():

        if len(sys.argv) != 2:
            print(
            "ERROR. Wrong number of arguments passed. System will exit. Next time please supply 1
argument!")
            sys.exit()
        else:
            print("1 Argument exists. We can proceed further")


    def checkPort():
        if int(sys.argv[1]) <= 5000:
            print(
            "Port number invalid. Port number should be greater than 5000. Next time enter valid port.")
            sys.exit()
        else:
            print("Port number accepted!")

    def ServerGet(g):
        print("Sending Acknowledgment of command.")
        msg = "Valid Get command. Let's go ahead "
        msgEn = msg.encode('utf-8')
        s.sendto(msgEn, clientAddr)
        print("Message Sent to Client.")

        print("In Server, Get function")

        if os.path.isfile(g):
            msg = "File exists. Let's go ahead "
            msgEn = msg.encode('utf-8')
            s.sendto(msgEn, clientAddr)
            print("Message about file existence sent.")

            c = 0
            sizeS = os.stat(g)
            sizeSS = sizeS.st_size  # number of packets
            print("File size in bytes:" + str(sizeSS))
            NumS = int(sizeSS / 1024)
            NumS = NumS + 1
            tillSS = str(NumS)
```

```
        tillSSS = tillSS.encode('utf8')
        s.sendto(tillSSS, clientAddr)

        check = int(NumS)
        GetRunS = open(g, "rb")
        while check != 0:
            RunS = GetRunS.read(1024)
            s.sendto(RunS, clientAddr)
            c += 1
            check -= 1
            print("Packet number:" + str(c))
            print("Data sending in process:")
        GetRunS.close()
        print("Sent from Server - Get function")

    else:
        msg = "Error: File does not exist in Server directory."
        msgEn = msg.encode('utf-8')
        s.sendto(msgEn, clientAddr)
        print("Message Sent.")


def ServerPut():
    print("Sending Acknowledgment of command.")
    msg = "Valid Put command. Let's go ahead "
    msgEn = msg.encode('utf-8')
    s.sendto(msgEn, clientAddr)
    print("Message Sent to Client.")

    print("In Server, Put function")
    if t2[0] == "put":

        BigSAgain = open(t2[1], "wb")
        d = 0
        print("Receiving packets will start now if file exists.")

        try:
            Count, countaddress = s.recvfrom(1024)
        except ConnectionResetError:
            print(
            "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
            sys.exit()
        except:
            print("Timeout or some other error")
            sys.exit()

        tillI = Count.decode('utf8')
        tillI = int(tillI)
```

```
        while tillI != 0:
            ServerData, serverAddr = s.recvfrom(1024)

            dataS = BigSAgain.write(ServerData)

            d += 1
            tillI = tillI - 1
            print("Received packet number:" + str(d))


        BigSAgain.close()
        print("New file closed. Check contents in your directory.")



def ServerElse():
    msg = "Error: You asked for: " + \
        t2[0] + " which is not understood by the server."
    msgEn = msg.encode('utf-8')
    s.sendto(msgEn, clientAddr)
    print("Message Sent.")


host = ""
checkArg()
try:
    port = int(sys.argv[1])
except ValueError:
    print("Error. Exiting. Please enter a valid port number.")
    sys.exit()
except IndexError:
    print("Error. Exiting. Please enter a valid port number next time.")
    sys.exit()
checkPort()

#port = 7777
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    print("Server socket initialized")
    s.bind((host, port))
    print("Successful binding. Waiting for Client now.")
    # s.setblocking(0)
    # s.settimeout(15)
except socket.error:
    print("Failed to create socket")
    sys.exit()

# time.sleep(1)
while True:
    try:
```

```
            data, clientAddr = s.recvfrom(1024)
        except ConnectionResetError:
            print(
            "Error. Port numbers not matching. Exiting. Next time enter same port numbers.")
            sys.exit()
        text = data.decode('utf8')
        t2 = text.split()
        #print("data print: " + t2[0] + t2[1] + t2[2])
        if t2[0] == "get":
            print("Go to get func")
            ServerGet(t2[1])
        elif t2[0] == "put":
            print("Go to put func")
            ServerPut()
        else:

            ServerElse()

    print("Program will end now. ")
    quit()

else:
    print('Incorrect input')
```

Лістинг 2 Код програманого забезпечення серверу