МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

КАФЕДРА ЕЛЕКТРОНІКИ І КОМП'ЮТЕРНОЇ ТЕХНІКИ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи бакалавра на тему:

«Організація локальної мережі та програмне забезпечення комунікацій для видавничого центру»

Завідувач кафедри А.С. Опанасюк

Керівник кваліфікаційної роботи Т.О. Протасова

Виконав студент гр. ТК-61 Д.А. Чехута

Суми 2020 р.

Сумський Державний Університет

Факультет ЕлІТ

Кафедра електроніки і комп'ютерної техніки

Спеціальність 172 "Телекомунікації та радіотехніка"

ЗАТВЕРДЖУЮ**:**

Зав. кафедри Опанасюк А.С.

«___» _____ 2020 р.

ЗАВДАННЯ

на кваліфікаційну роботу бакалавра

студенту **Чехуті Дмитру Андрійовичу**

1. Тема роботи: «Організація локальної мережі та програмне забезпечення комунікацій для видавничого центру»

затверджено наказом по кафедрі від 13.04.2020 р. № 0544-III

2. Термін здачі студентом закінченої роботи: 10.06.2020 р.

3. Вихідні дані до роботи: 1. Організувати локальну мережу для п'яти робочих місць. 2. Розробити та адаптувати програмне забезпечення під локальну мережу.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці): 1. Організація локальної мережі для видавничого центру. 2. Розробка клієнт-серверного додатку для видавничого центру.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): Презентація, яка складається з 18 слайдів.

Дата видачі завдання: 10.03.2020 р.

Завдання прийняв до виконання: _____ Чехута Д.А.

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів кваліфікаційної роботи	Термін виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літератури та формулювання задачі розробки	25.03.2020	
2	Аналіз моделі OSI та архітектур мережі	10.04.2020	
3	Організація локальної мережі	25.04.2020	
4	Розробка клієнтської частини	30.04.2020	
5	Розробка серверної частини	10.05.2020	
6	Організація взаємодії клієнтської та серверної частин	17.05.2020	
7	Структуризація всього матеріалу та оформлення кваліфікаційної роботи	24.05.2020	
8	Здача кваліфікаційної роботи на кафедру для рецензування	10.06.2020	

Студент Чехута Д.А.

Керівник кваліфікаційної роботи Протасова Т.О.

«___» _____ 2020 р.

РЕФЕРАТ

У даній роботі організовано архітектуру локальної мережі для п’яти робочих станцій видавничого центру. Розроблено клієнт-серверний додаток для локальної мережі.

Призначенням організованої локальної мережі та розробленого клієнт-серверного додатку полягає в забезпеченні потреб комунікації персоналу за умови відсутності доступу до глобальної мережі Internet.

В роботі побудовано архітектуру локальної мережі, функціональні схеми серверної та клієнтської частин програмного-забезпечення. Вирішено питання масштабованості, як для локальної мережі, так і для клієнт-серверного додатку.

Кваліфікаційна робота бакалавра містить 52 сторінки тексту, 24 рисунки, 1 таблицю, 4 додатки, графічний матеріал у вигляді презентації, яка складається з 18 слайдів.

У першому розділі проводиться аналіз комп’ютерних мереж, їх архітектур. На основі проаналізованих даних організовано архітектуру локальної мережі.

У другому розділі проводиться аналіз архітектур розробки клієнт-серверного додатку. Організовується алгоритм взаємодії окремих блоків програмного забезпечення. Приводяться функціональні блок-схеми алгоритму роботи розробленого клієнт-серверного додатку.

В кінці пояснювальної записки наводяться висновки та приведений перелік літературних джерел.

Кількість літературних джерел — 9.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧОК

ЛОМ — локальна обчислювальна мережа;

OSI — модель взаємодії відкритих систем (The Open Systems Interconnection);

PC — персональний комп’ютер.

JS — JavaScript;

HTTP — протокол передачі гіпер-текстових документів.

ВВЕДЕННЯ

Підтримка такої бізнес-функціїї, як внутрішня взаємодія між працівниками у всіх передових компаніях світу ставиться в ряди найприорітетніших завдань. Компаніям необхідно розробити інструмент досягнення стратегічних цілей через організований процес обміну внутрішньо корпоративної інформації між всіма працівниками. Такий інструмент неможливо розробити при повноцінному доступі до мережі Internet. Тому більшість компаній змушені або повністю відмовлятися від доступу до мережі Internet або значно його обмежувати з організацією службового доступу. В такому випадку більшість працівників «відрізані» від можливості комуні кувати за робочим місцем. Це призводить до таких наслідків як: зниження ефективності працівників, зниження стійкої корпоративної культури, несвоєчасне і неповне інформування працівників тощо.

Такі компанії потребують ефективно організованої локальної мережі, а також програмне забезпечення, яке буде забезпечувати потребу внутрішньо-корпоративної комунікації.

Локальна мережа має бути адаптована до потреби масштабування, а також до програмної надбудови, яка може виконувати різні корпоративні функції та потреби.

Програмне забезпечення має відповідати вимогам надійності, масштабування, можливості підримки, інтуїтивно зрозумілий користувачу.

1 ОРГАНІЗАЦІЯ ЛОКАЛЬНОЇ МЕРЕЖІ ДЛЯ ВИДАВНИЧОГО ЦЕНТРУ

1.1 Призначення комп’ютерної мережі

Локальні обчислювальні мережі (ЛОМ) — це сукупність комп’ютерів, кабелів, мережевих адаптерів, що працюють під управлінням мережевої операційної системи й прикладного програмного забезпечення, розташовані на обмеженій території офісу чи будівлі. Для мережевої комп‘ютерної системи часто використовується абревіатура NOS (Network Operating System) [1].

У локальних мережах поняття інтерактивного зв'язку комп'ютерів - це обмін повідомленнями в реальному режимі часу; мережі дозволяють цілому ряду користувачів одночасно «володіти» програмами, базами даних, периферійними пристроями й т.п. Наприклад, якщо декільком користувачам треба роздрукувати документ, усі вони можуть звернутися до мережевого принтера [1,2].

У даний час більшість організацій береже і спільно використовує в мережевому середовищі величезні обсяги життєво важливих даних і спеціальних програм, створених для спільного використання в умовах локальних мереж, наприклад, ряд спеціальних банківських програм, програм бухобліку [1].

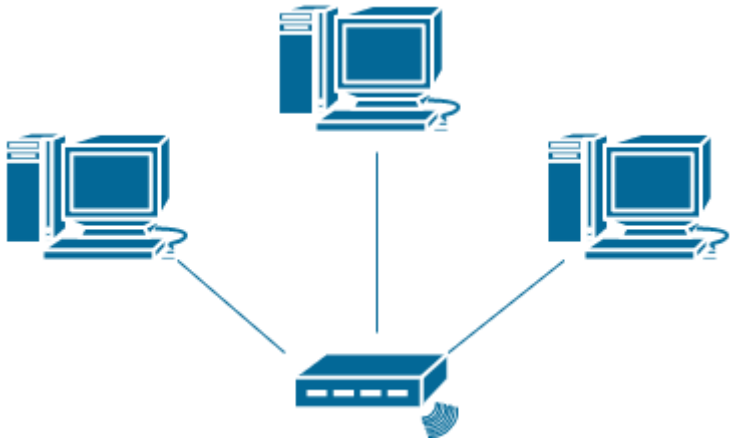

Рисунок 1.1 — Приклад простої ЛОМ

Саме тому у сучасному світі ЛОМ — це цілий, складний, важливий механізм, який забезпечує або приймає участь у функціонуванні усіх важливих для людства процесах.

1.2 Поняття відкритої системи. Модель OSI

Важливо розуміти, що ЛОМ – це дуже складана система, яка містить в собі ще ряд систем, які взаємодіють між собою. Єдині правила використання таких систем, а також їх взаємодії — дуже важливий крок у організації ЛОМ. Звісно, якщо б кожна людина розробляла свої правила та специфікації, людство мало би набагато більше проблем у питаннях модернізації, масштабуванні, оптимізації ресурсів ЛОМ тощо. Тому доречно ввести поняття відкритої системи. В широкому розумінні слова відкрита система – це система, що побудована у відповідності з відкритими специфікаціями. Де під специфікаціями розуміється формальний опис апаратних, або програмних компонентів, способів їх функціонування, взаємодії з іншими компонентами, умов експлуатації і особливих характеристик.

Під відкритими специфікаціями розуміються опубліковані,

загальнодоступні специфікації, що прийняті зацікавленими сторонами в якості стандартів. Використання відкритих специфікацій дозволяє:

– розробляти апаратні і програмні засоби для розширення і модифікації;

– існуючих систем за допомогою третіх осіб;

– створювати комплексні системи з продуктів різних виробників [3].

Практично, повна відкритість систем часто виявляється недосяжною. Як правило, визначенню відкритих відповідає лише частина специфікацій, що підтримуються через зовнішні інтерфейси. Чим більше відкритих специфікацій використано для розробки, тим система є відкритішою. Особливого значення відкриті специфікації набувають для побудови комп’ютерних мереж, що складаються з різноманітного комп’ютерного і

комунікаційного обладнання і тому проблема сумісності для них є однією з найбільш гострих.[1,2,3]

Дотримання принципів відкритості при побудові мереж надає наступні переваги:

– можливість використання апаратних і програмних засобів від незалежних виробників;

– можливість удосконалення окремих елементів мережі без зміни інших;

– можливість легкого об’єднання мереж;

– простота у освоєнні та обслуговуванні [2].

З метою допомоги виробникам в стандартизації програмного і апаратного забезпечення комп’ютерних мереж, Міжнародна організація з стандартизації (International Standard Organization – ISO) у 80-х рр. XX ст. разом з іншими розробила еталонну модель взаємодії відкритих систем — Open System Interconnection (OSI) Reference Model, що показана на рис. 1.2.

Модель OSI збудована на основі великого досвіду, набутого в 70-х рр. при створенні глобальних комп’ютерних мереж. Модель будувалася так, щоб розділити функції стеків протоколів і забезпечити можливість їх розробки незалежними організаціями, тобто щоб процес розробки протоколів став більш раціональним. Модель OSI, визначає сім рівнів взаємодії систем і функції, які має виконувати кожний рівень [2].

Модель OSI описує тільки системні засоби взаємодії (які реалізуються операційною системою, системними утилітами, системними апаратними засобами) і не включає засоби взаємодії з прикладними процесами користувачів. Свої власні протоколи взаємодії прикладні процеси реалізують, звертаючись до системних засобів. Тому необхідно розрізняти рівень взаємодії додатків і прикладний рівень. В табл. 3.1 показано призначення кожного з рівнів моделі та приклади протоколів [1, 2].

Рисунок 1.2 — Модель OSI

Таблиця 1.1 — Призначення рівнів моделі OSI

№	Рівень	Призначення	Приклади
1	2	3	4
7	Прикладний (Application)	Забезпечує послуги, що надаються безпосередньо прикладним програмам	SMTP, HTTP, FTP і т.п.
6	Представлення (Presentation)	Забезпечує проведення сеансів зв'язку(тобто установку, підтримку і переривання зв'язку). Цей же рівень розпізнає логічні імена абонентів, контролює надані їм права доступу.	Стандарти кодування (GIF, JEPEG, TIFF MPEG і т.п.)
5	Сеансовий (Session)	Забезпечує проведення сеансів зв'язку (тобто установку, підтримку і переривання зв'язку). Цей же рівень розпізнає логічні імена абонентів, контролює надані їм права доступу.	Remote Procedure Call, Session Control Protocol (SCP)

Продовження таблиці 1.1

1	2	3	4
4	Транспортний (Transport)	Забезпечує доставку даних від одного вузла до іншого без помилок і втрат, а також в необхідній послідовності, через їх розбивку на пакети і нумерування пакетів. Доставка пакетів можлива як з встановленням з'єднання (віртуального каналу), так і без нього.	TCP, UDP
3	Мережевий (Network)	Забезпечує логічну структуризацію мережі і маршрутизацію пакетів між підмережами. Цей же рівень здійснює перетворення мережевих адрес в фізичні (наприклад, IP-адрес в МАСадреси).	IP
2	Канальний (Data Link)	Забезпечує надійну передачу даних в рамках підмережі з тим чи іншим каналом зв'язку (шляхом формування низькорівневих кадрів для даного виду підмережі)	Ethernet, Token Ring, FDDI, Frame Relay, PPP (Point-toPoint Protocol)
1	Фізичний (Physical)	Забезпечує умови прийому-передачі по фізичному каналу зв'язку шляхом визначення вимого до його фізичних, механічних, електричних та інших характеристик (рівні напруги, частота, опір і т.п.)	LAN категорії 3, LAN категорії 5, V.90

Наступним питанням є опис блоків даних у протоколах – PDU (Protocol Data Units). Термін пакет (packet), зокрема, визначає блок даних, що передається через мережеве середовище, хоча він також застосовується і для опису даних на будь-якій стадії процесу [2].

На фізичному рівні відбувається передача бітів (bits). На канальному рівні інформація сформована в кадри (frames). На мережевому рівні ще не сформовані кадри носять назву дейтаграми (datagrames) [2].

На транспортному рівні інформація розбивається на сегменти (segments). Відповідно. на прикладному рівні інформація розглядається як повідомлення (messages). Протоколи представницького і сеансового рівня не

формують своїх заголовків і тому до даних у них застосовують термін повідомлення. З урахуванням цього модель взаємодії відкритих систем набуває вигляду, показаного на рис. 1.3. [2].

Виходячи з інформації, наведеної у таблиці 1.1 можна побачити, що чотири нижніх основних рівні моделі OSI забезпечують виключно транспортні функції, які реалізуються за допомогою відповідних апаратних засобів (мережева карта, повторювач, концентратор, комутатор, маршрутизатор). Їх прийнято називати мережевим транспортом. А три верхніх рівні є виключно програмною надбудовою над мережевим транспортом і їх задача полягає у наданні послуг мережі прикладним додатком [3].

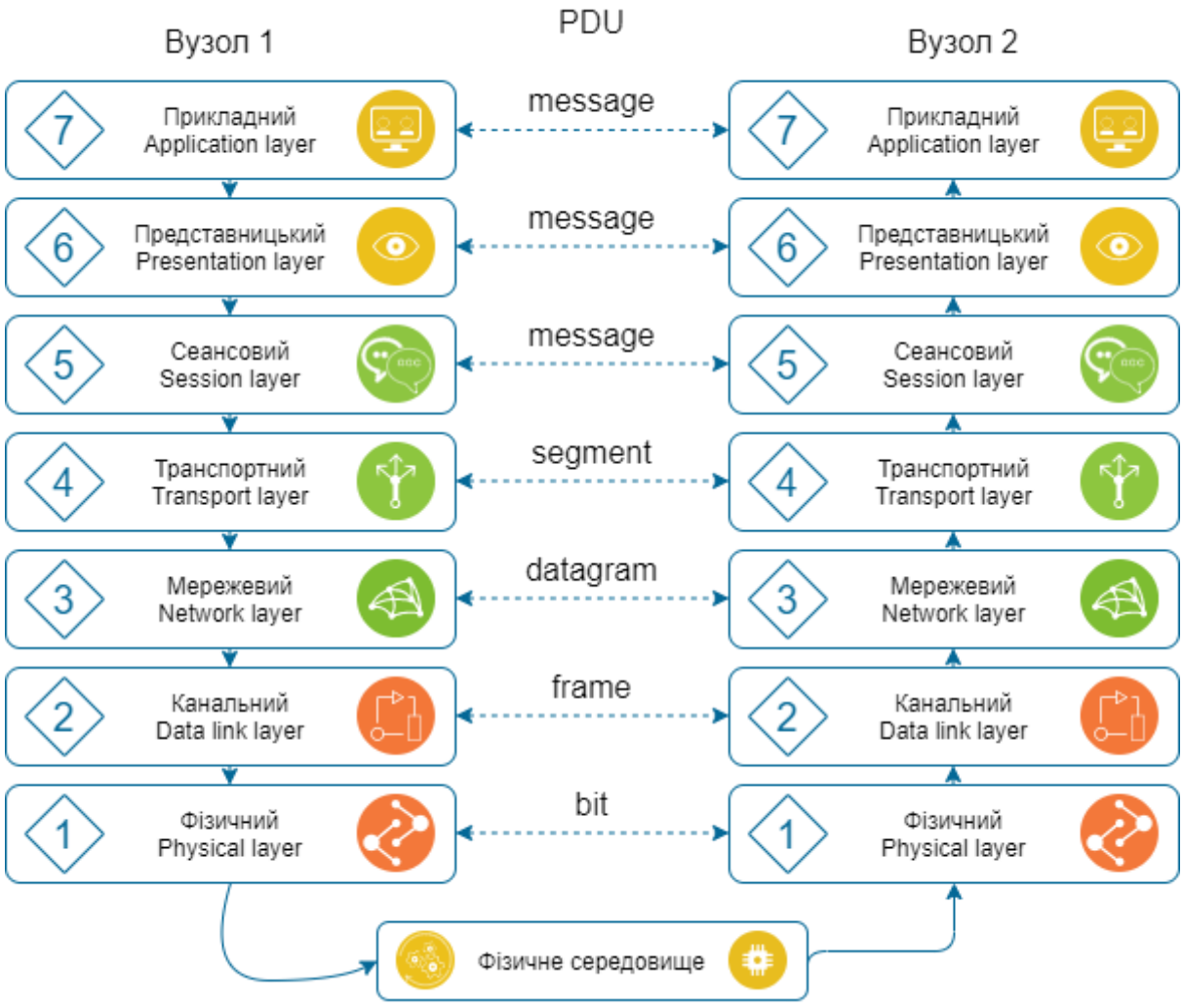

Рисунок 1.3 — Приклад взаємодії двох вузлів мережі

1.3 Концепції організації мережі

Найпростіша мережа (network) складається як мінімум із двох комп'ютерів, сполучених один з одним кабелем. Як правило, для такого з'єднання використовували послідовні порти, один із комп'ютерів призначався майстром, а інший – підлеглим. Користувач комп'ютера-майстра мав значну перевагу в правах (міг копіювати файли та каталоги з комп'ютера-майстра на підлеглий комп'ютер і навпаки, установлювати програми на підлеглому комп'ютері, видаляти, зберігати інформацію й т.п.). Тобто таке з'єднання дозволяло їм використовувати дані спільно. Усі мережі (незалежно від складності) засновуються саме на цьому простому принципі. Комп'ютер, або сервер, або робоча станція, підключається до мережі за допомогою внутрішньої плати - мережевого адаптеру (хоча бувають і зовнішні мережеві адаптери, що підключаються до комп'ютера через паралельний порт). Мережеві адаптери перетворюють коди, які використовуються всередині комп'ютера, у послідовний потік потужних сигналів для передавання у мережі. Мережеві адаптери повинні бути сумісні з кабельною системою мережі, внутрішньою інформаційною шиною ПК (персонального комп'ютера) і мережевою операційною системою [3].

У мережі розрізняють: client — комп'ютер, що під'єднаний до мережі, але який не надає свої власні ресурси іншим мережевим машинам; server — комп'ютер, що під'єднаний до мережі, який має ресурси, призначені для спільного використання; client/server — термін, що означає, виконання мережевої задачі розбивається на дві частини: одна виконується на робочій станції, інша — на сервері; file server — мережевий диск (диски), доступний користувачам інших комп'ютерів; ―print server”- комп'ютер, відповідальний за мережевий друк; local resources — (локальні ресурси) диски, принтери й інші пристрої, пов'язані безпосередньо з робочою станцією; network resources ― диск, принтер або інший пристрій, розташований на сервері, який надає ресурси іншим користувачам, на відміну від локальних ресурсів; mail server — серверний комп'ютер, на якому зберігаються повідомлення, які надійшли електронною поштою; coaxial cable — кабель, що складається з двох

провідників: центральний провідник покритий ізоляційним прошарком і одягнений у металевий чохол другого провідника; “repeater” (підсилювач) - пристрій, що збільшує потужність сигналу таким чином, щоб він без перекручувань міг передаватися далі мережею; shielded twisted pair — кабель типу «скручена пара», що складається з однієї й більше пар проводів, скручених із метою поліпшення їхніх електричних характеристик; волокнисто-оптичний кабель (двох типів): багатоходовий кабель (fiber optic cable multimode) і одномодовий кабель (fiber optic cable single mode); computer name — ім'я, яке призначається кожному комп'ютеру мережі; modem — пристрій, що перекладає комп'ютерні сигнали в сигнали, які можна передавати лініями до такого модему, що перетворить їх у початкову форму; network drive — диск, який використовується комп'ютером, але не під'єднаний до нього безпосередньо; off-line — недоступний компонент у мережі; on-line — доступний компонент у мережі; operator — користувач, що може контролювати роботу мережі, але не має повноважень для призначення або зміни прав користувача, адміністрування; packets — дані, що передаються мережею, розбиті на порції, названі пакетами, розмір і структура пакета визначаються протоколом; print queue — черга на друк в мережі; protocol — це правила і технічні процедури, що дозволяють декільком комп'ютерам при об'єднанні в мережу спілкуватися один з одним. Протоколи передають дані, керуючись стандартизованими форматами, виявляють і виправляють помилки і т.д. [1.3].

1.4 Архітектура мережі

Архітектура локальної мережі – це концепція її побудови, яка визначає:

- основні елементи мережі;
- топологію мережі і функції кожного її елементу;
- фізичну і логічну організацію взаємодії елементів мережі.

За формою представлення комп’ютерних мереж розрізняють фізичну та логічну архітектуру [3].

Фізична архітектура – форма представлення комп'ютерної мережі у вигляді взаємодіючих апаратних засобів. Приклад фізичної архітектури локальної мережі зображено на рис. 1.4 [3].

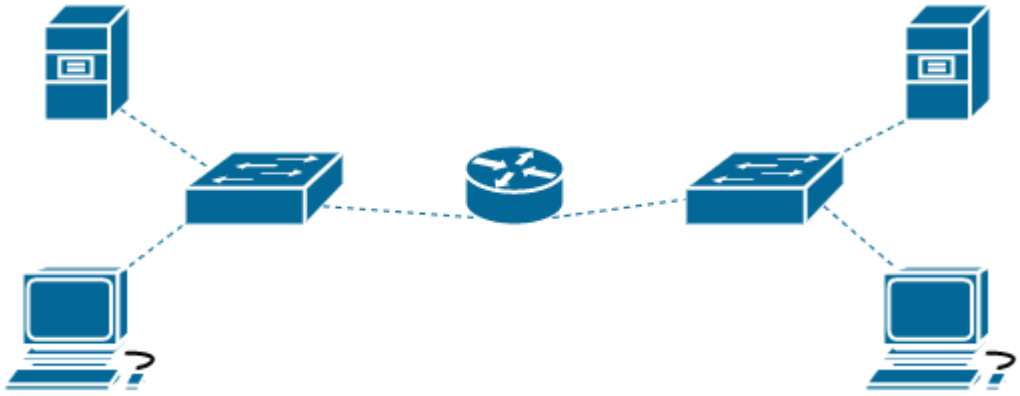

Рисунок 1.4 — Приклад фізичної архітектури локальної мережі

Логічна архітектура – форма представлення комп'ютерної мережі у вигляді взаємопов'язаних елементів (функцій). Приклад логічної архітектури комп'ютерної мережі зображено на рис. 1.5. Логічна архітектура відбиває цілісну технологію комп'ютерної мережі і може бути деталізованою через рівні фізичної архітектури. [2,3]

У комп'ютерних мережах розрізняють п'ять архітектурних шаблонів:

– архітектура «термінал-головний комп'ютер»;
– архітектура «клієнт-сервер»;
– однорангова архітектура;
– архітектура «комп'ютер-мережа»;
– архітектура інтелектуальної мережі.

Рисунок 1.5 — Приклад логічної архітектури локальної мережі

1.4.1 Архітектура «термінал-головний комп’ютер»

Архітектура «термінал-головний комп’ютер» (terminal-host computer architecture) – це така концепція комп’ютерної мережі, коли вся обробка даних здійснюється одним, або групою головних комп’ютерів.

Така архітектура передбачає три основних типи обладнання, що показано на рис. 1.6:

– головний комп’ютер (mainframe) – здійснює управління мережею, збереження і обробку даних.

– термінали (terminal) – забезпечують передачу головному комп’ютеру команд для організації сеансів роботи, введення даних і отримання результатів.

– мультиплексори (multiplexor, MUX) – забезпечують «об’єднання» потоків даних від терміналів в спільний вихідний потік. Отже, мультиплексор – це комбінаційний пристрій, який забезпечує передачу даних, що надходять з кількох входів на один вихід.

Класичним прикладом архітектури «термінал-головний комп’ютер» є системна мережева архітектура – System Network Architecture (SNA) – запатентована компанією IBM мережева архітектура, що забезпечує підключення систем локальних мереж до мейнфреймів і мінікомп’ютерів IBM. Під мейнфреймом розуміється комп’ютер з архітектурою IBM/370, а під мінікомп’ютером – AS/400 [2].

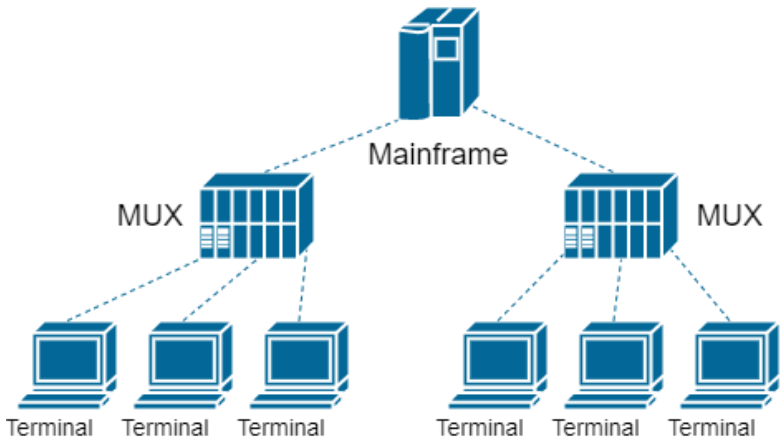

Рисунок 1.6 — Приклад шаблону архітектури «термінал — головний комп’ютер»

1.4.2 Однорангова архітектура

Однорангова архітектура (peer-to-peer architecture) – архітектурний шаблон комп'ютерної мережі, що базується на рівнозначності комп'ютерів в мережі. Тобто – кожний вузол (peer) виступає як в ролі клієнта, так і сервера. Відповідно – жоден комп'ютер не має ні вищого пріоритету на доступ, ні підвищеної відповідальності за надання ресурсів у спільне використання.

Однорангові мережі називають також робочими групами. Кожний користувач у такій мережі є одночасно і адміністратором мережі, оскільки, через використання паролів, він керує доступом до ресурсів свого комп'ютера [2,3].

Розглянемо переваги та недоліки однорангової архітектури.

Переваги наступні:

- простота в установленні та налаштуванні мережі;
- невисока вартість і простота експлуатації мережі;
- незалежність комп'ютерів (від сервера);
- простота в управлінні ресурсами (кожний користувач управляє
- доступом до ресурсів власного комп'ютера);
- відсутність необхідності в персоналі для адміністрування мережі.

Недоліки однорангової архітектури:

- невелика кількість (близько 10) комп'ютерів в мережі;
- необхідність використання великої кількості паролів, якими забезпечується доступ до ресурсів мережі;
- зменшення продуктивності тих комп'ютерів, ресурси яких інтенсивно використовуються, відсутність централізованих можливостей для пошуку і управління даними.

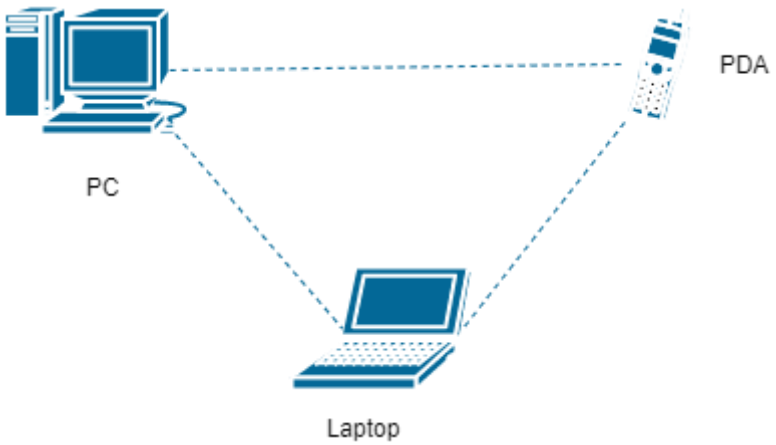

Рисунок 1.7 — Приклад шаблону однорангової архітектури мережі з використанням Wi-Fi

1.4.3 Архітектура «комп'ютер-мережа»

Архітектура «комп'ютер-мережа» (computer-network architecture) – концепція комп'ютерної мережі, в якій програмне забезпечення надається користувачу як Інтернет-сервіс: користувач отримує доступ до даних, але не може управляти операційною системою і програмним забезпеченням, з яким він працює. До даного шаблону архітектури, що показаний на рис. 1.8, вживається також термін Cloud computing, що перекладається як «обчислення в хмарі», або як «хмарні обчислення». Ідеологія Cloud computing започаткована в 2007 р. та широко розвивається. завдяки розвитку каналів зв'язку [2].

Рисунок 1.8 — Приклад шаблону архітектури «комп'ютер-мережа»

Концепція Cloud computing дає користувачам недосяжні раніше можливості: обмежені в ресурсах компанії можуть дозволити собі власні бізнес-додатки і поштові сервери, реально маючи тільки доступ до Інтернет. Прикладом Cloud computing є компанія Google, яка надає користувачам необмежений дисковий простір для збереження електронної пошти (Google Mail) та стандартні офісні додатки в режимі on-line (Google Apps) [2].

1.4.4 Архітектура інтелектуальної мережі

Поняття інтелектуальної мережі (Inteligent Network – IN) пов'язане з наданням користувачам комутованої телекомунікаційної мережі розширеного і постійно розширюваного набору послуг. Головна ідея полягає у виокремленні процесу традиційної комутації викликів від процесу введення нових послуг. Для цього потрібні певні інтерфейси між комутаторами мережі і «інтелектуальною надбудовою», що показана на рис. 1.9. Модернізація послуг в цьому випадку виконується лише шляхом модернізації програмного забезпечення «інтелектуальної надбудови», що дозволяє швидко впроваджувати на існуючих мережах будь-які послуги незалежно від виробника комунікаційного обладнання [2,3].

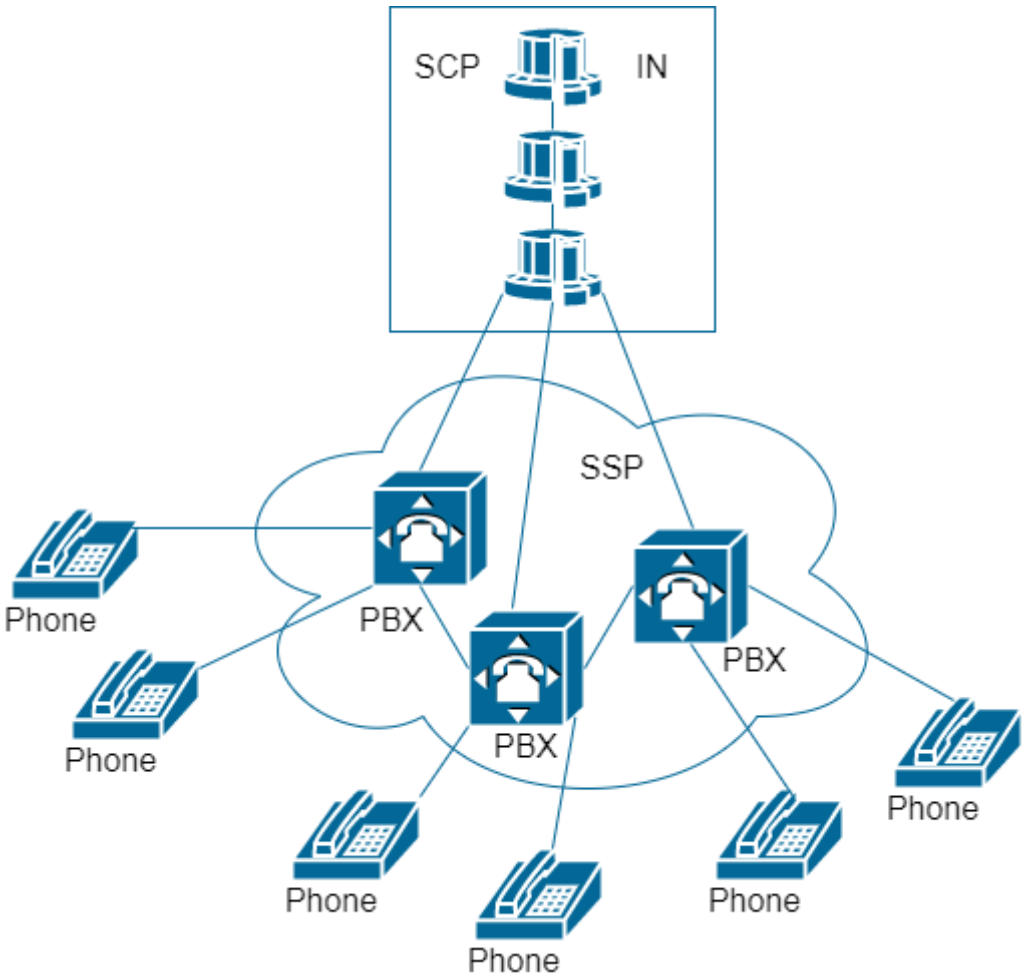

Рисунок 1.9 — Приклад шаблону архітектури «комп’ютер-мережа»

Відповідно до такої архітектури, для надання послуг впроваджують пункти комутації послуг (SSP – Service Switching Point) та пункти управління послугами (SCP – Service Control Point). У пункті управління послугами SCP міститься основний «інтелект» у вигляді певних алгоритмів надання послуг і наборів баз даних [2,3]

Пункт комутації послуг SSP розпізнає виклики до IN, що надходять з комп’ютерної мережі та направляє їх в SCP для обробклення, виступаючи в ролі інтерфейсу до «інтелектуальної надбудови», прикладами елементів системи SSP може бути, наприклад, IP-PBX (Private Branch eXchange) - це система корпоративної телефонії, основним каналом передачі голосу в якій є VoIP. PBX об'єднує всі офісні телефони в одну інтелектуальну мережу, в якій реалізуються різні голосові сервіси.. Система управління послугами SMS (Service Management System) забезпечує введення нових послуг, корекцію старих, містить відомості про абонентів, оригінали програм [2,3].

1.4.5 Архітектура «клієнт-сервер»

Архітектура «клієнт-сервер» (client-server architecture) – це концепція комп'ютерної мережі, в якій основна частина ресурсів зосереджена на серверах, що обслуговують своїх клієнтів [3].

Зв'язок між комп'ютерами в мережі даної архітектури відбувається за рахунок відправки/прийому спеціальних повідомлень, що передаються через мережеві адаптери і лінії зв'язку. На рис. 1.10 показано, що за допомогою таких повідомлень один комп'ютер (PC-1) може надсилати серверу (Server) запити на доступ до його локальних ресурсів – даних на диску, периферійних пристроїв (принтерів, модемів і т.п.). Або взаємодіяти з іншим комп'ютером (PC-2) за допомогою додаткового функціонуючого програмного забезпечення на сервері. Способи використання серверу в локальній мережі задля досягнення певних цілей дуже багато. Прикладу розповсюджених функціональних можливостей серверу будуть розглянуті далі [2,3]

Для забезпечення можливості обміну повідомленнями операційні системи комп'ютерів (OS) доповнюються відповідними програмними модулями – клієнтами і серверами [1].

На тих комп'ютерах мережі, ресурси яких повинні бути доступні іншим користувачам, додаються модулі, які постійно знаходяться в режимі очікування запитів від користувачів; такі модулі отримали назву програмних серверів, оскільки їх основним завданням є обслуговування (serve) запитів користувачів [2,3].

На тих комп'ютерах мережі, які здійснюють доступ до ресурсів інших комп'ютерів, додаються програмні модулі, які можуть формувати запити і передавати їх по мережі, такі модулі отримали назву програмних клієнтів [2,3].

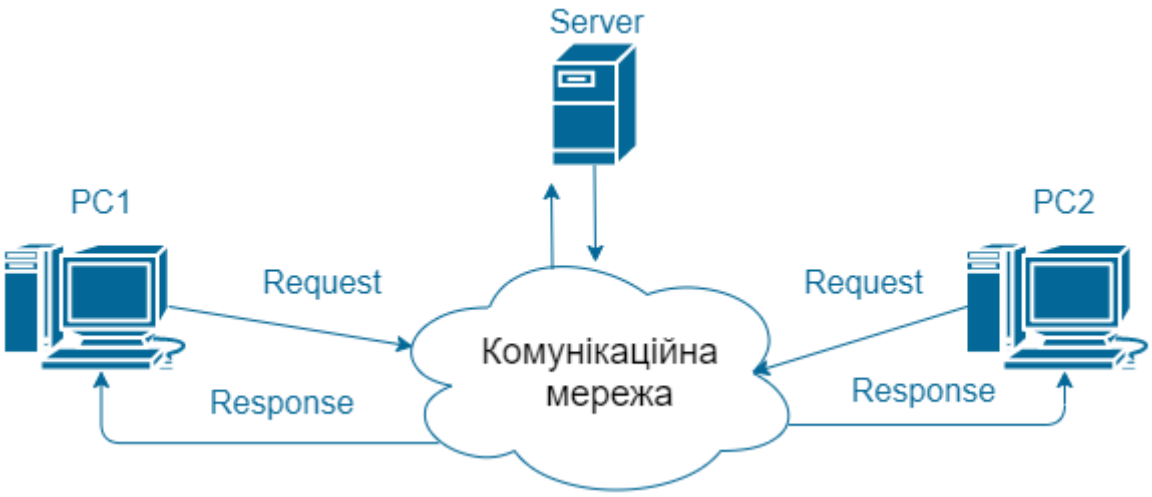

Рисунок 1.10 — Приклад шаблону архітектури «клієнт-сервер»

Тобто, сервер (Server) – програмний прикладний процес, що забезпечує виконання сервісної функції.

Клієнт (Client) – програмний або людино-машинний прикладний процес, що викликає сервісну функцію.

Сервісна функція – комплекс прикладних програм, у відповідності з яким виконуються різноманітні прикладні процеси.

Пара модулів «клієнт»-«сервер», що забезпечує сумісний доступ користувачів до певного типу ресурсів, називають мережева службою.

Терміни «клієнт» і «сервер» використовуються не тільки для позначення програмних модулів, а й для позначення місця, що має комп'ютер в мережі. Якщо комп'ютер надає свої ресурси у спільне використання, він називається сервером, а якщо споживає ресурси інших – клієнтом [2,3].

У цьому разі:

Сервер (Server) – спеціалізований комп'ютер, що надає сервіс іншим комп'ютерам мережі за їх запитами.

Клієнт (Client) – комп'ютер, який використовує ресурси сервера і забезпечує користувача інтерфейсом для роботи.

Сервіс – процес обслуговування клієнта.

Інтерфейс користувача – процедури взаємодії користувача з мережею.

У шаблоні архітектури «клієнт-сервер» комп'ютери клієнтів називають робочими станціями. Сервери спеціально оптимізуються для швидкої

обробки запитів клієнтів до розподілених ресурсів і для управління захистом файлів [2,3].

За функціональним призначенням розрізняють (рис. 1.11) кілька типів серверів: файлові, друкування, додатків, поштовий, комунікаційний, базданих. Розглянемо їх детальніше.

Рисунок 1.11 — Функціональне призначення сервера

Сервер друкування (принт-сервер) – комп’ютер, що забезпечує доступ до централізовано поділюваного принтера, створює чергу друку і виконує управління принтером [2].

Комунікаційний, або сервер віддаленого доступу (Access Server) – надає можливість робочим станціям поділяти модем, або вузол зв’язку, з великою ЕОМ, що забезпечує доступ до мережі з віддаленого місця,

обладнаного модемом. Часто комунікаційний сервер суміщає і функції сервера додатків [2].

Поштовий сервер – комп'ютер, що призначений для передачі електронних повідомлень між користувачами мережі [2].

Файловий сервер – комп'ютер, який виконує функції управління локальною мережею, доступом користувачів до файлів, що спільно використовуються [2].

Сервер додатків – комп'ютер, що виконує прикладну задачу, запуск якої здійснюється користувачем зі свого терміналу [2]..

Сервер бази даних – комп'ютер, що забезпечує вибірку необхідних даних з бази даних і пересилку через мережу лише даних, що запитані клієнтом [2]..

В архітектурі «клієнт-сервер» окремий (виділений) сервер забезпечує також централізований захист мережі завдяки перевірці облікових записів користувачів. Зокрема, Windows NT використовує систему доменних імен для управління користувачами, групами і машинами. Перед тим, як користувач отримає доступ до мережевих ресурсів, він має пройти процедуру авторизації, тобто – повідомити своє реєстраційне ім'я і пароль контролеру домену – серверу, який дозволяє доступ тільки у випадку допустимої комбінації імені і паролю [2,3].

Розглянемо переваги і недоліки архітектури «клієнт-сервер». Переваги:

– дозволяє організовувати мережі з великою кількістю робочих станцій;

– спрощує мережеве адміністрування завдяки можливості централізованого управління обліковими записами;

– забезпечує ефективний доступ до мережевих ресурсів (без використання паролів доступу до ресурсів).

Недоліки архітектури «клієнт-сервер»:

– критична по відношенню до працездатності сервера,

– вимагає кваліфікованого персоналу для адміністрування мережі,

– підвищення вартості мережі через використання потужних серверів.

1.5 Організація локальної мережі

Грунтуючись на проведеному аналізі була організована мережа(рисунок 1.12).

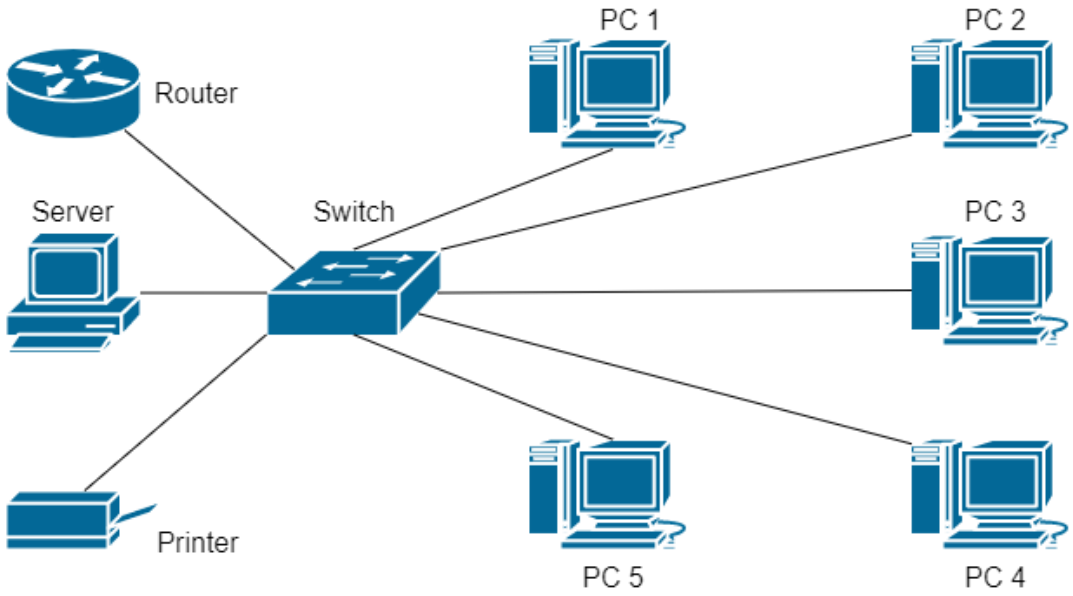

Рисунок 1.12 — Організована локальна мережа

На базі комутатора (switch) побудова локальна мережа, до якого підключені сервер (server), а також користувачі з персональних комп'ютерів (PC1 – PC5). Також в мережу доданий роутер (router) для взаємодії через технологію Wi-Fi та ресурс спільного доступу — принтер.

2 РОЗРОБКА КЛІЄНТ-СЕРВЕРНОГО ДОДАТКУ ДЛЯ ВИДАВНИЧОГО ЦЕНТРУ

Клієнт-серверний додаток, який буде виконувати функції комунікації користувачі у локальній мережі побудований як SPA (Single Page Application). Дуже важливо зупинитися на цьому моменті, оскільки це визначає подальші принципи, концепції, стек технологій при розробці клієнт-серверного додатку.

SPA — це «додаток однієї сторінки», в якому завантаження необхідного коду відбувається на одну сторінку. Середовищем функціонування додатку в такому випадку стає браузер. Важливо розуміти, що це жодним чином не означає, що додаток матиме одну сторінку. Це означає, що додаток використовує єдиний HTML-документ як оболонку для всіх web-сторінок і організовує взаємозв'язок з користувачем через динамічно завантажуванні HTML, CSS, JavaScript модулі. Це дозволяє заощадити та оптимізувати час та ресурси комп'ютера при повторному завантаженні однакових елементів. Обробка даних в SPA відбувається на стороні сервера, браузер користувача відкриває інтерфейс додатку, після чого користувач при взаємодії з додатком відправляє запити на сервер та отримує оброблену інформацію [9].

Переваги такого підходу:

– Доступність та універсальність. Можна отримати моментальний доступ до функціоналу з будь-якого типу пристрою без проблем з сумісністю, достатнім об'ємом пам'яті, необхідними обчислювальними потужностями або ж затратою часу на установку;

– Можливість задіяти великі об'єми даних. Розмір додатку і даних, які він використовує не обмежений пам'яттю пристрою користувача;

– Швидкість. Одна сторінка, яка містить весь необхідний інтерфейс не тільки економить час на повторне завантаження даних, але й підвищує продуктивність роботи з додатком.

Недоліки:

– Необхідність стабільного мережевого з’єднання. Без доступу до мережі використання такого програмного забезпечення неможливе.

– Не працює у користувача з відключеною підтримкою JavaScript. Деякі користувачі відключають відображення JavaScript-елементів в своєму браузері, через що SPA, яке використовує їх в роботі не функціонує.

Оскільки середовище функціонування такого додатку є браузер, то для розробки будуть використовуватися web-технології. А саме:

– HTML/CSS — використовується для відображення DOM-елементів в браузері;

– Javascript — використовується задля забезпечення динамічно оновлюваного контенту на web-сторінці.

Оскільки Javascript буде забезпечувати і клієнтську, і серверну частину були обрані наступні стеки технологій:

– UI частина:
  - Bootstrap;
– Клієнтська частина:
  - React;
  - Socket.
– Серверна частина:
  - NodeJS/ExpressJS;
  - Socket.

2.1 Клієнтська частина

На даному етапі розробка клієнтської частини (frontend частини) відбувається інкапсульовано від серверної. Основна задача на цьому етапі розробки побудувати функціонуючий каркас для серверної частини, який може бути адаптований під будь-який серверний додаток та дані. Основний принцип, або ж паттерн, згідно якому буде розроблятися frontend частина — це MVC (Рисунок 2.1). Статична HTML-сторінка функціонально не здатна реагувати на дії користувача. Для двосторонньої взаємодії необхідні динамічні web-сторінки. MVC започатковує основні принципи динамічних

web-сторінок. MVC розшифровується як модель — відображення — контролер (від англ. model — view — controller). Модель в цьому паттерні відповідає за дані, а також визначає структуру додатку. Відображення — компонент, який відповідає за взаємодію з користувачем, тобто за зовнішній вигляд додатку та способи його використання. Контролер — відповідає за взаємозв'язок між model та view, код цього компонента визначає як додаток реагує на дії користувача [9].

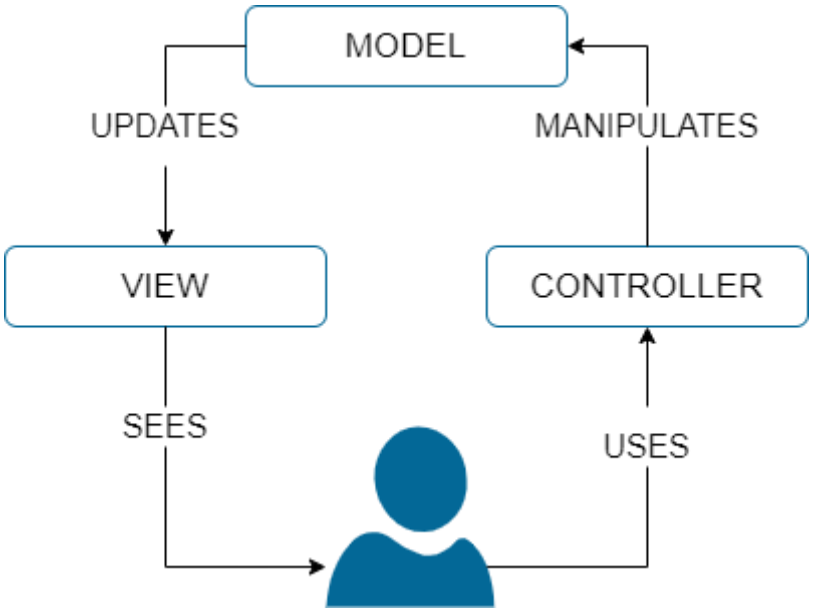

Рисунок 2.1 — Архітектурний шаблон MVC

2.1.1 Відображення інтерфейсу додатку

На цьому етапі реалізовується складова паттерну MVC: view. Ця частина відповідає лише за інтерфейс. При розробці інтерфейсу клієнтської частину додатку буде використовуватись React, а також UI-бібліотека Bootstrap.

При написанні зовнішнього виду додатку використовуватиметься компонентний підхід. Це означає, що кожен структурний елемент буде відокремлений та розробляється окремо. Після розробки усіх таких елементів, усі вони комбінуються у головний файл, де і створюють загальний інтерфейс. Такий підхід дозволяє локалізувати розробку тобто структурувати

елементи розробки, спрощення знаходження помилок, в разі їх виникнення, дозволяє пере використовувати елементи, наприклад, ми розроблюємо компонент повідомлення, в якому міститься текст та автор — кожний раз розробляти інтерфейс для кожного повідомлення дуже неправильне рішення. Тому розробляємо лише один елемент повідомлення, а потім за допомогою деструктуризації масиву даних множимо елемент рівно стільки — скільки необхідно додатку.

Виходячи із задачі створення програмного засобу комунікації ми розробимо два компоненти. Перший назвемо JoinBlock (при розробці використовується лише англійська). Цей блок виконуватиме функцію отримання від користувача даних про кімнату, в яку він хоче додатись, а також його ім’я. Другий компонент — Chat. В цьому компоненті відбувається взаємодія між користувачами, тобто обмін повідомленнями. А також інформування користувача про чат (кількість користувачів, назва чату тощо.

Результат розробки інтерфейсу клієнтської частини в Додатку А.

2.1.2 Організація логіки клієнтської частини

В цьому пункті реалізовуються с складові паттерну MVC: model та controller. React можна використовувати не лише для побудови користувацького інтерфейсу, але й для організації логіки взаємодії між користувачем та додатком [4].

Для побудови такої логіки в JS-бібліотеці React є фундаментальні речі такі як state, props, hooks та функції-обробники, за допомогою яких ми забезпечуємо динамічність. State — це стан компоненту або окремих його елементів, на основі стану компоненту ми можемо задавати алгоритми дій компоненту [4], тобто це model-частина MVC паттерну. При найпростіших ситуаціях такий стан може бути true/false. Наприклад, якщо true, то ми робимо один алгоритм дій, якщо ні то інший. Звідси виходить наступне поняття props. Справа в тому, що дуже часто необхідно так, щоб зміна стану в одному компоненті змінювала деякі елементи іншого компоненту. Інший

компонент не знає, що відбувається в головному компоненті. Саме тому передача даних із одно компоненту до іншого відбувається за допомогою props. Функції-обробники — функції, які прослуховують HTML-елементи, та в залежності від дій користувача, тобто це controller-частина MVC паттерну.

Розробимо схему взаємодії з централізованим управлінням, тобто state у компоненті найвищого рівня ієрархії. Таким чином, цей стан компоненту і залежності від нього передаються іншим компонентам за допомогою props до компонентів нижчими за ієрархією, які займаються лише тим, що відображають цей state. Ну а якщо виникає необхідність змінити цей глобальний state, то використовується механізм функцій-обробників, так з'являється можливість передавати дані компонентам вище за ієрархією (рисунок 2.2).

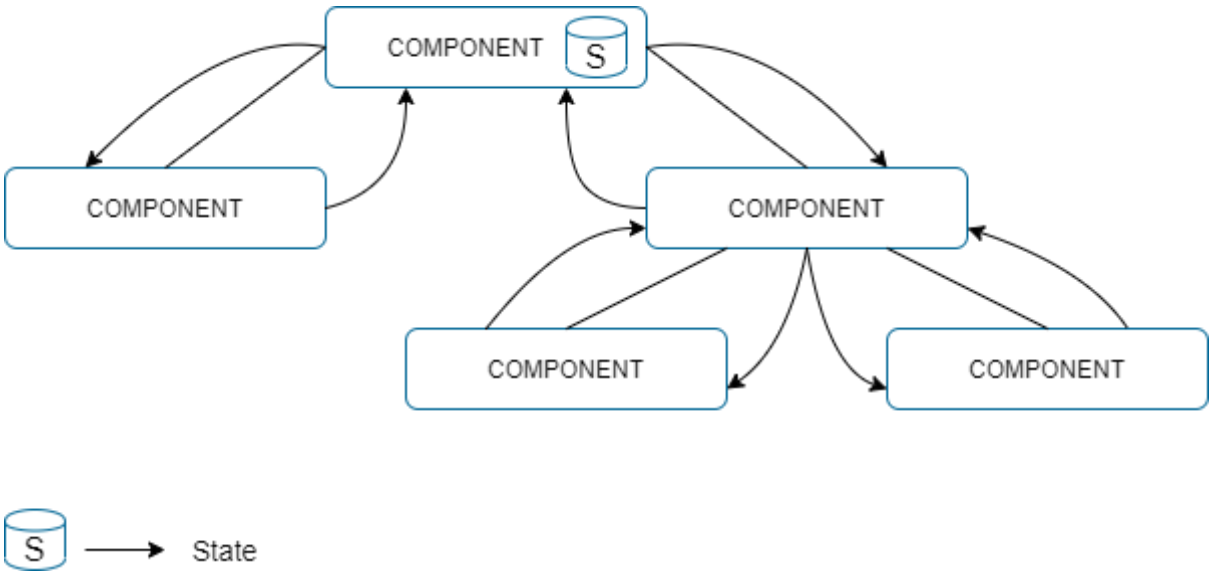

Рисунок 2.2 — Централізована система управління

Перевагою цього підходу є те, що в додатку існує єдине джерело істини, головний state, який диригує іншими компонентами, а також оновлюється також централізовано. З цього випливає й недолік. Для того, що передати state до найнижчих компонентів ієрархії необхідно передати необхідні дані через всю структуру компонентів. Виникає проблема — компоненти, які не мають необхідності працювати з потоком даних вимушені це робити для того, щоб передати ці дані далі по ієрархії. А для того, щоб

оновити state необхідно передати через ті ж самі компоненти абсолютно неважливі для них дані вгору по ієрархії. Ця проблема називається props drilling (з англ. буріння властивостями). Ми як буром проходимо через кожний компонент і це залишає сліди на кожному з рівнів.

На основі цієї структури розроблено іншу — фрагментовану систему управління (рисунок 2.3), в якій проблема props drilling відсутня. Кожний компонент має свій локальний state, а якщо є залежність від state, іншого компоненту оновлення реалізовується за допомогою hooks або методів життєвого циклу [4].

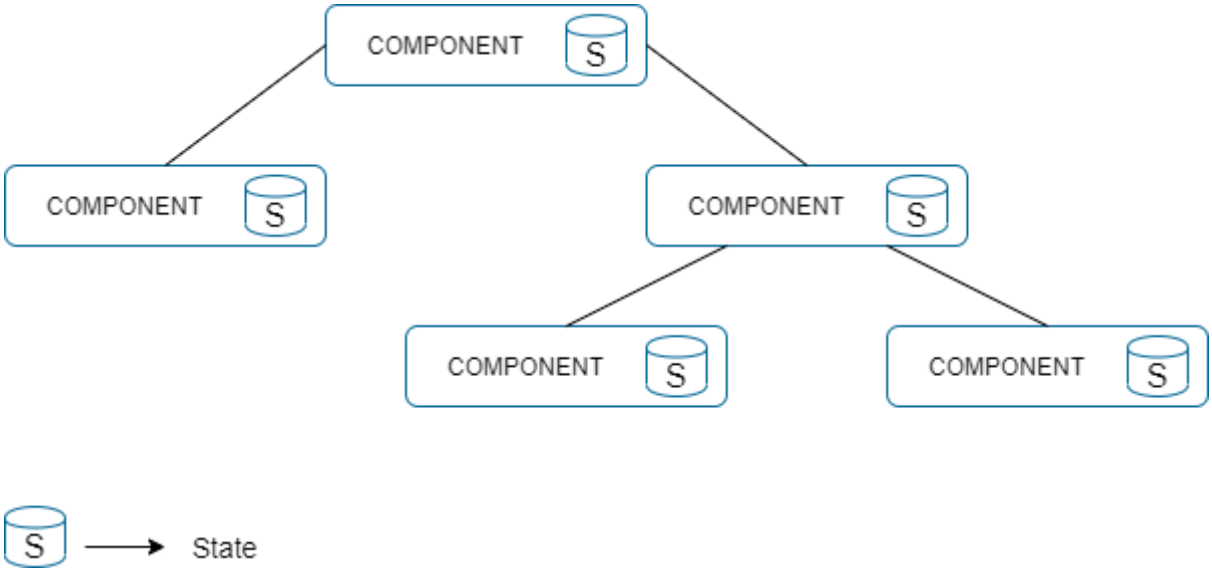

Рисунок 2.3 — Фрагментована система управління

Таким чином, взаємодія між компонентами зводиться до мінімума. Проте такий підхід складний при масштабуванні, коли компонентів стає все більше, взаємодія між ними ускладнюється. При розробці такої системи дуже легко допустити помилку, яка важко відслідковується, а може зовсім не бути виявленою на етапі розробки. Некоректно працююча система – це набагато гірше, аніж непрацююча система.

І централізована, і фрагментована системи управління дуже часто використовуються при розробці інтерфейсів та є дуже потужними інструментами за допомогою яких реалізується бізнес-логіка. І на даному етапі розробки і функціональних потреб клієнтської частини будь-яка із вище

перелічених систем добре підходять для реалізації клієнтської частини. Проте також однією із задач розробки КСД є забезпечення принципів масштабування. Тобто забезпечення КСД можливістю його модернізувати надалі, на базі розробленого. Тому в такому випадку використання централізованої і фрагментованої систем управління не є такими, які повністю задовольняють потреби розробки.

Тоді розробка системи управління логікою у додатку буде організовано за допомогою виключення тих недоліків, які були у попередніх. Усі локальні state винесемо до окремої сутності, яка матиме назву global state. Global state — окрема сутність, яка існує окремо від компонентів. Доступ до global state матимуть усі компоненти. Для того, щоб компоненти не могли змінювати логіку додатку, яка не входить до їх компетенції, дані зі global state можна буде тільки читати. Оскільки компоненти не можуть оновлювати логіку додатку виникає проблема, що додаток вже і не можна називати динамічним. Наступна задача полягає в розробці механізмів оновлення як самих компонентів, так і global state. Створимо ще одну додаткову сутність і назвемо її reducer. Та винесемо усі функції, які керували логікою додатка до reducer. Тепер із reducer та store додаток максимально наближений до моделі MVC. Компоненти (view), в яких залишились лише елементи відображення, reducer (controller)— це логічне осердя, в якому зберігаються функції, які знають як змінювати global state (model), в залежності від значення якого додаток показує в компонентах те, що необхідно користувачу. Залишилось лише поєднати всі ці сутності. Зі сторони компонентів такий зв’язок реалізовується за допомогою базової javascript-сутності — подій [8]. За допомогою певних подій, які провів користувач до reducer будуть передаватися спеціальні об’єкти, які називаються actions. Цей об’єкт має два поля, які необхідні для подальшої ідентифікації, що робити reducer. Перше поле — це тип події, а друге — значення, яке потрібно передати до state. А відправляються до логічного блоку actions за допомогою спеціального методу dispatch. Сумуючи описаний раніше алгоритм дій можна сказати, що при такому підході управління даними — рендерінг (відображення) та блок

логіки (store) додатку існують окремо один від одного і взаємодіють між собою за допомогою спеціальних об’єктів actions та dispatch (рисунок 2.4).

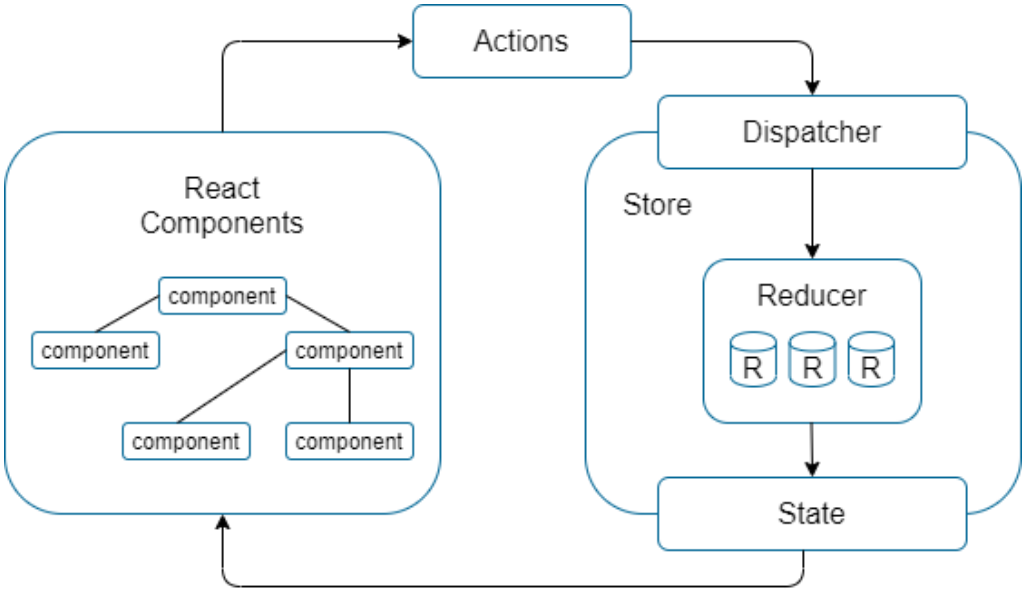

Рисунок 2.4 — Архітектура клієнтського частини додатку

Оскільки архітектура клієнтської частини додатку вже отримано залишається її адаптувати під функціональні потреби програмного забезпечення комунікації — Рисунок 2.5.

У блоці розробки інтерфейсу були розроблені компоненти, в які необхідно додати actions, в залежності від того, які дії від цих елементів очікує користувач. Відповідно до кожного actions розробити reducer, який буде змінювати глобальний state усього додатку.

Програмний код розробки логічної частини на стороні клієнта наведено в Додатку Б.

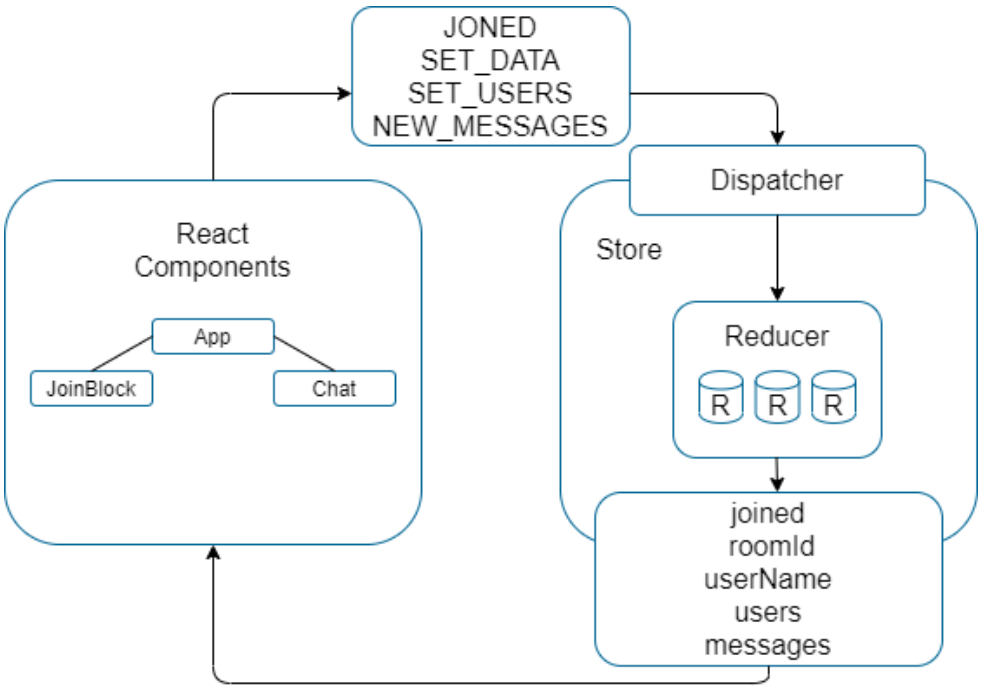

Рисунок 2.5 — Блок-схема алгоритму функціонування каркасу клієнтської частини КСД

2.2 Серверна частина

На даному етапі розробки важливо розробити серверний додаток, який буде функціонально підготовлений взаємодіяти з клієнтською частиною, здатний приймати, оброблювати запити та відповідати на них. Браузери взаємодіють із веб-серверами за допомогою гіпертекстового транспортного протоколу HTTP. При кожній дії на веб-сторінки, яка потребує додаткової інформації із сервера (заповнення форми, залучення пошуку тощо) HTTP-запит відправляється з браузера користувача на цільовий сервер [9].

Цей запит включає в себе:

– Шлях, який визначає цільовий сервер і ресурс, який необхідний (наприклад, визначена точка даних на сервері);

– Метод, який визначає необхідні дії (наприклад, отримання певної інформації, зберегти або оновити певні дані):

- GET — отримати певний ресурс;
- POST — створити певний ресурс;

- HEAD — отримати метадані про певний ресурс без отримання змісту, як це робить GET;
- PUT — створити новий ресурс;
- DELETE — видалити певний ресурс.

– Додаткова інформація, яка може бути закодована в запиті.

Сервера знаходяться в стані очікування повідомлень з клієнтськими запитами, обробляють їх як тільки вони надійдуть і відповідають за допомогою HTTP-запиту у відповідь. Відповідь містить строку стану, яка показує був запит успішним або ні [7].

Розуміння цієї моделі взаємодії дозволяє нам правильно спроектувати серверний додаток, організувати функціональну частину обробки запитів — Рисунок 2.6.

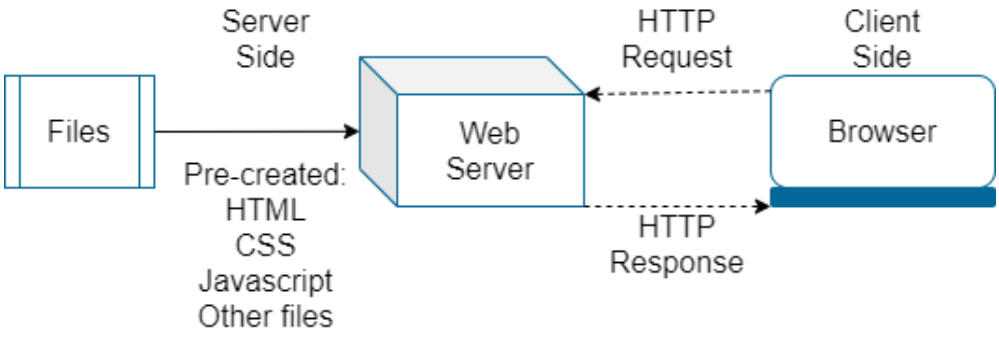

Рисунок 2.6 — Статична клієнт-серверна взаємодія

На рисунку 2.6 зображена типова схема взаємодії клієнту та сервера. Проте ця модель має ряд вад. Сервер повертає один і той самий закодований зміст із сервера. Кожний раз, коли є запит на ресурс. Це типова ситуація статичного веб-серверу. Сервер витягує дані, які необхідні в запиті зі своєї файлової системи і повертає HTTP-відповідь з даними та успішним статусом (зазвичай 200 OK). Якщо сервер не може витягнути ті чи інші дані – повертається статус помилки.

Виходячи із поставленої задачі створення програмного засобу комунікації нам така модель не підходить. Дані, які потрібно повертати серверу це нові повідомлення від користувачів, ці дані динамічно

оновлюються. Тому нам необхідні додаткові елементи в описаній раніше взаємодії. Необхідно, щоб частина відповіді генерувалася динамічно тільки при необхідності, а також, щоб сервер міг повертати різні дані для URL-адреси на основі інформації, яку йому надає клієнт. Схема описана нижче показує просту архітектуру динамічної взаємодії клієнт-сервер. Як і на схемі раніше браузери відправлять HTTP-запит на сервер, після чого сервер оброблює запит і повертає відповідні HTTP-відповіді. Запити статичних ресурсів обробляються так само як і для статичної взаємодії — Рисунок 2.7.

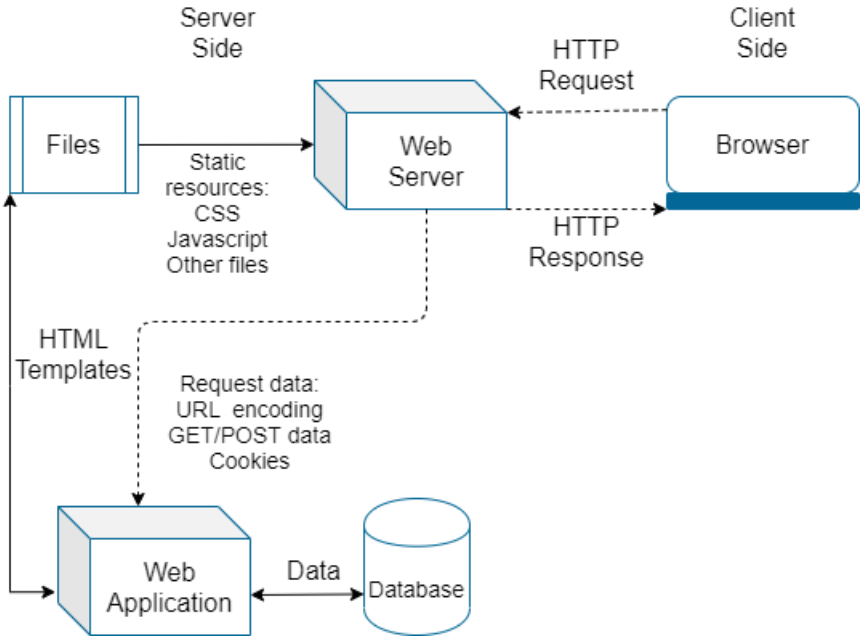

Рисунок 2.7 — Динамічна клієнт-серверна взаємодія

Запити динамічних даних відправляються в код серверної частини (показано як веб-додаток). Для динамічних запитів сервер інтерпретує запит, читає необхідну інформацію з місця зберігання даних (показано як database), комбінує необхідні дані с шаблоном HTML і повертає відповідь, яка містить сгенерований дані.

Маючи головну модель розробимо каркас серверної веб-додатку — Рисунок 2.8.

В якості HTTP-серверу будемо використовувати http-модуль серверної платформи Node.js [3]. Для створення серверного додатку використаємо фреймворк Express [5]. Створюємо прослойку, яка формуватиме отримані

дані в дані типу json. Створення методів обробки get/post-запитів. Створення моделі зберігання даних. Оскільки у даному випадку зберігання даних відбувається у вигляді послідовності байтів, то задача внутрішньої організації цих даних вирішується на рівні додатку. В даному випадку така модель реалізовується за допомогою js-об'єкту Map [8], який містить пру ключ-значення і зберігає порядок вставки.

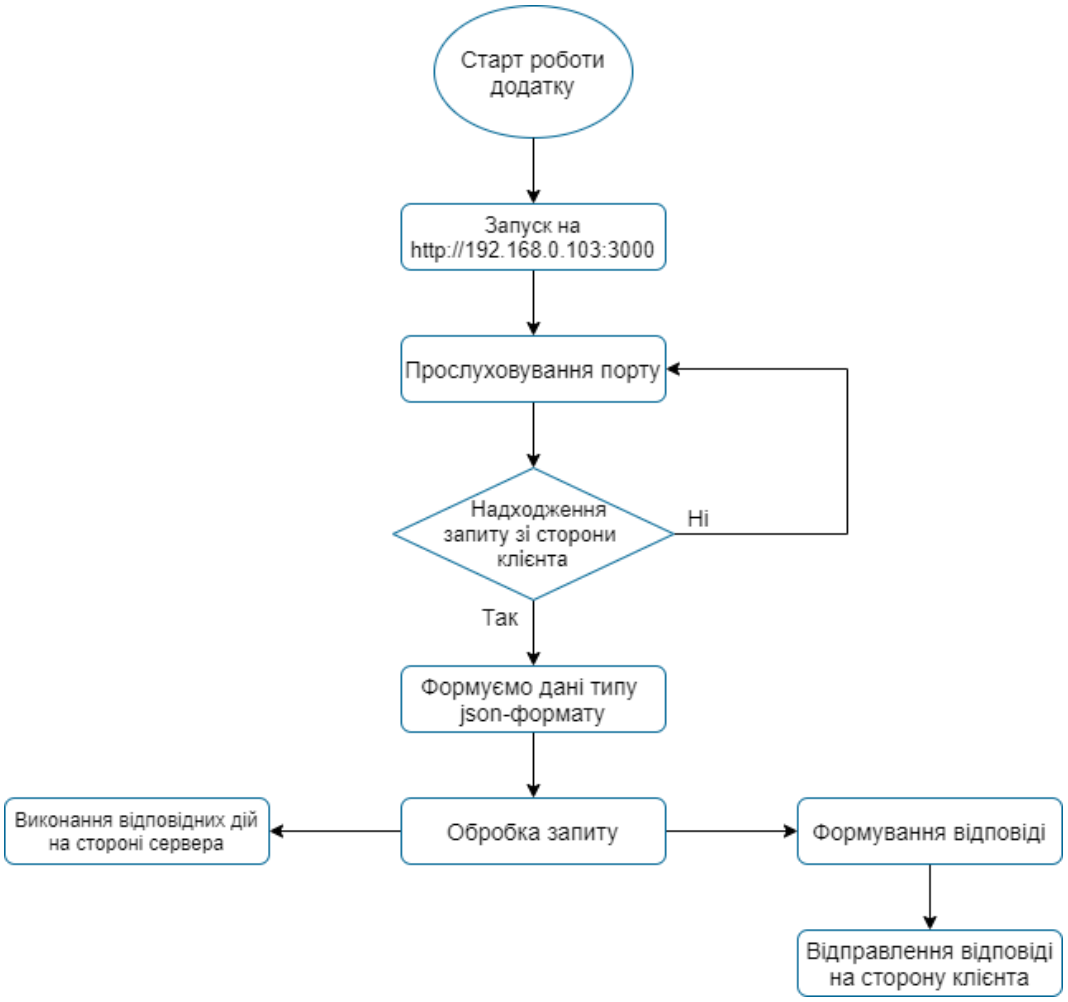

Рисунок 2.8 — Блок-схема алгоритму функціонування каркасу серверної частини КСД

Програмний код каркасу серверної частини розміщено в Додатку В.

2.3 Розробка функціональної взаємодії між клієнтською та серверною сторонами. Socket.

В розробці серверної частини дуже ретельно була проаналізована робота http-запитів, оскільки цей механізм дозволяє взаємодіяти клієнту із

сервером. Як було з’ясовано сервер реагує на дії користувача і надає йому інформацію. Однак, в специфіці розробки програмного засобу комунікації виникає проблема. Коли користувач надсилає повідомлення, він робить необхідні дії, внаслідок яких формується та відправляється http-запит [9]. Але щоб отримати повідомлення від користувача — отримувач нічого не робить і це логічно, він просто очікує на повідомлення. Модель http-запитів передбачає роботу з одно направленим потоком даним. Це означає, що для того, щоб користувач отримав повідомлення клієнт завжди повинен запитувати сервер чи не оновилася інформація (рисунок 2.9).

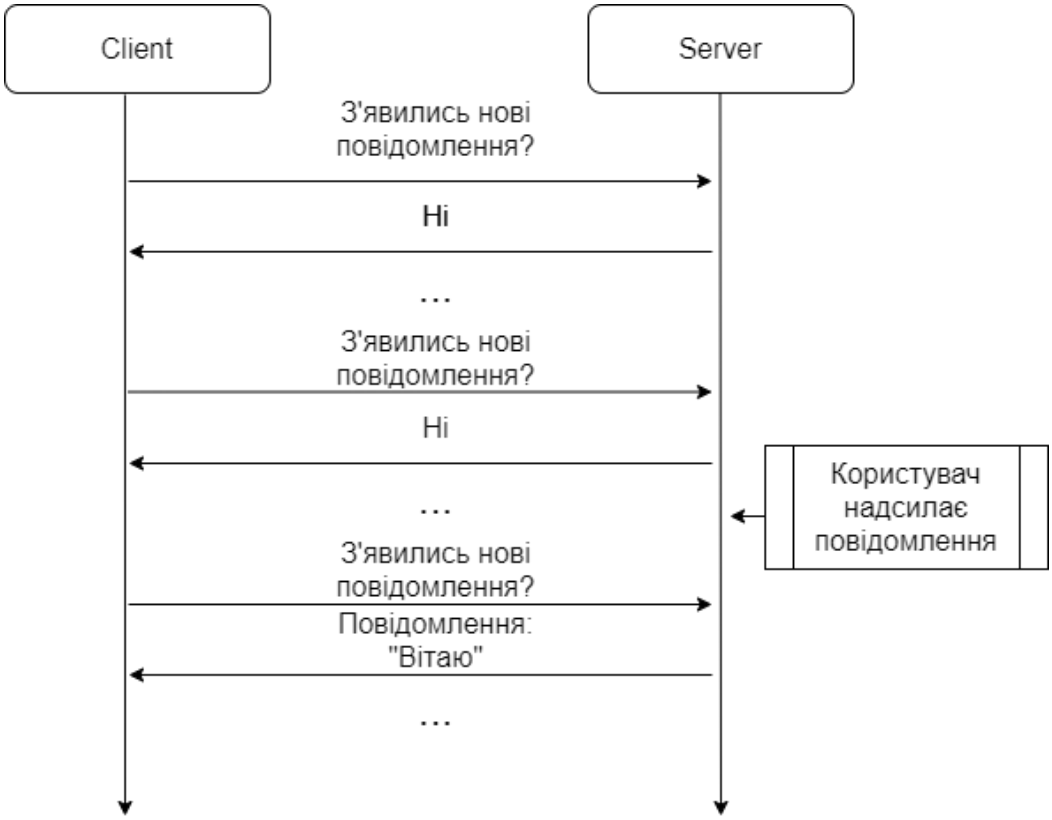

Рисунок 2.9 — Модель http-запитів

В даній задачі є два шляхи вирішення проблеми. Перший — створити спеціальну кнопку, при натисканні якої користувач буде ініціювати запит на сторону сервера: «чи не прийшло йому повідомлення». Цей підхід звісно є дуже поганим рішенням. Друге рішення — створити механізм на стороні клієнта, який буде через певний інтервал часу запитувати у сервера про

оновлення інформації. Такий підхід також є поганим рішенням, оскільки несе певне навантаження на мережу, якщо користувачів буде дуже багато, то це призведе до перенавантаження мережі, а в наслідок чого й до повільної роботи клієнт-серверного додатку.

Виникає питання доцільності використання моделі одно направленого потоку даних. Насправді ця модель є дуже потужним інструментом і в розробці КСД приймає участь у додаванні користувача до кімнати. Проте в організації роботи обміну повідомленнями цей підхід викликає ряд проблем.

Тому при обміні повідомленнями буде використовуватись двох направлена модель передачі даних. Ця технологія має назву Web Socket (рисунок 2.10).

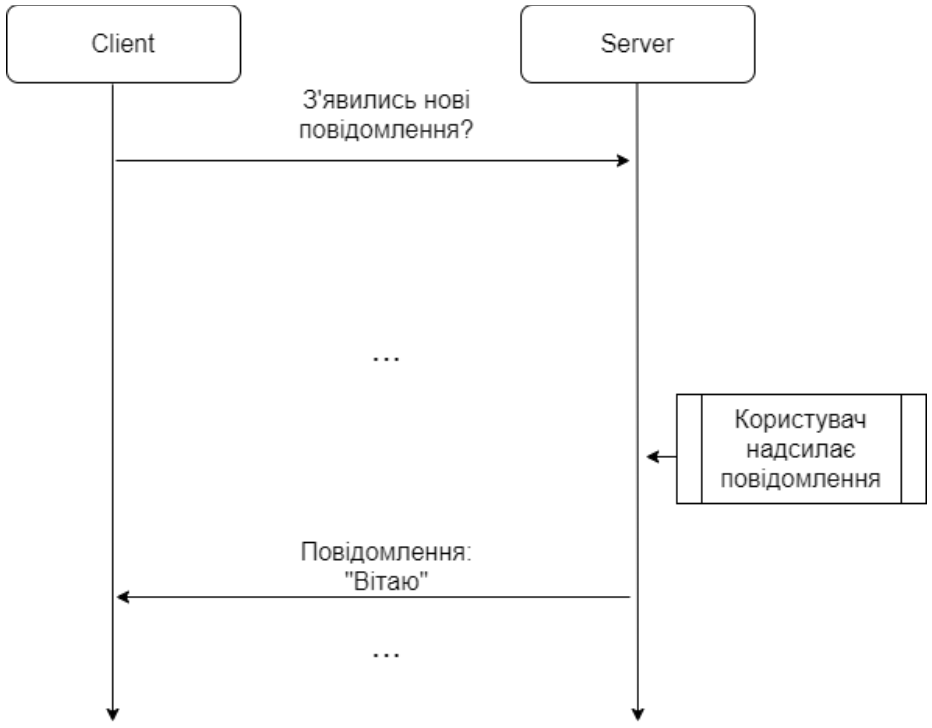

Рисунок 2.10 — Модель socket-запитів

При використанні даної технології залучена спеціальна бібліотека — Socket.io [3]. Web Socket — це передова технологія, яка дозволяє створювати інтерактивне з’єднання між клієнтом(браузером) та сервером для обміну даними в режимі реального часу. Веб-сокети, на відміну від HTTP, дозволяють оброблювати двонаправлений потік даних. Веб-сокетам для

відповіді не потрібні повторні запити. Достатньо виконати один запит і чекати відповіді. Оснащення веб-сокетами КСД відбувається і на клієнтській, і на серверній сторонах. КСД готовий до використання. Алгоритм роботи КСД зображено на рисунках 2.11 та 2.12.

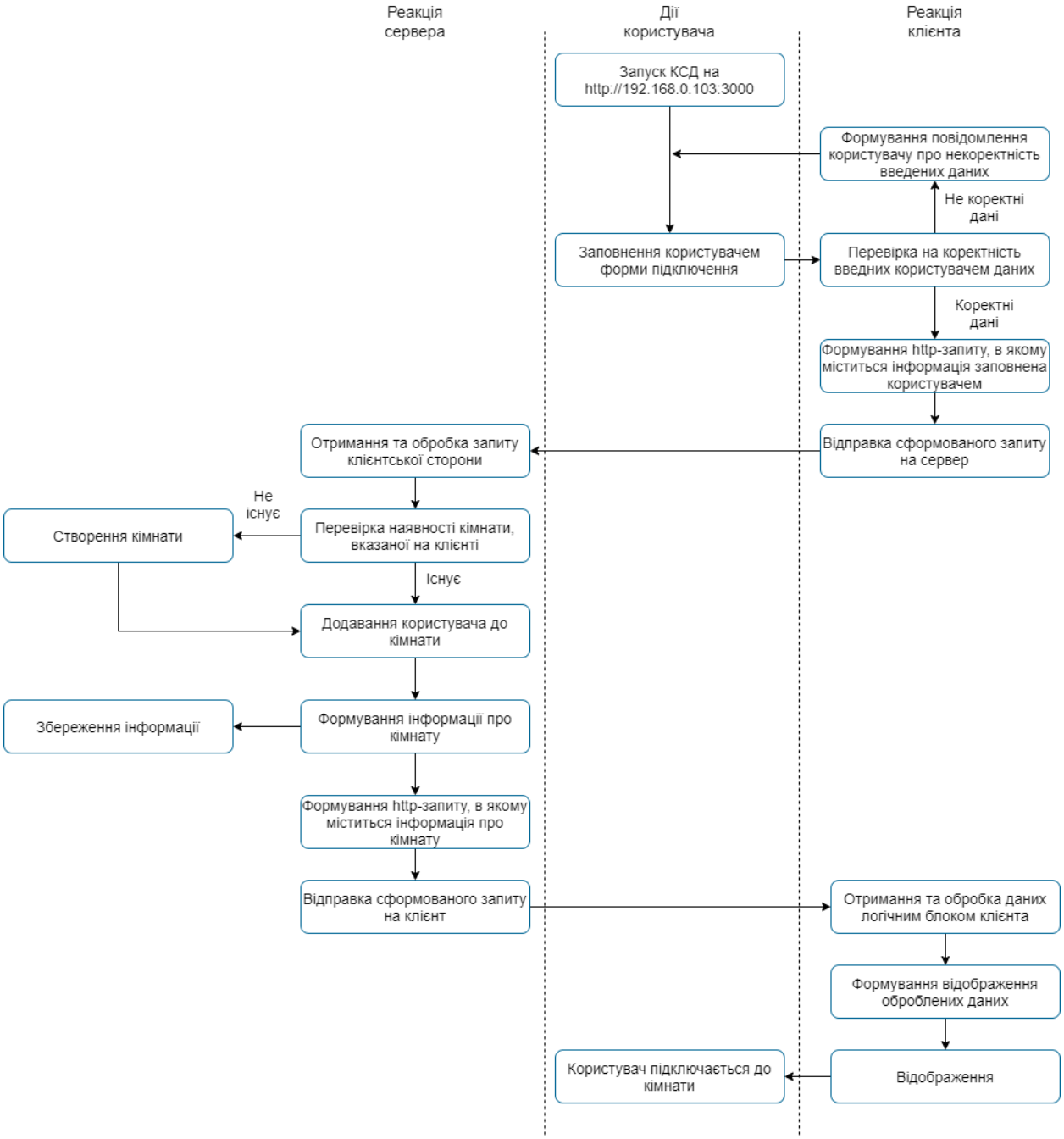

Рисунок 2.11 — Блок-схема алгоритму дій КСД на етапі підключення користувача

Оснащення веб-сокетам, а також основні складові програмного коду кінцевої версії КСД розміщено у Додатку Г.

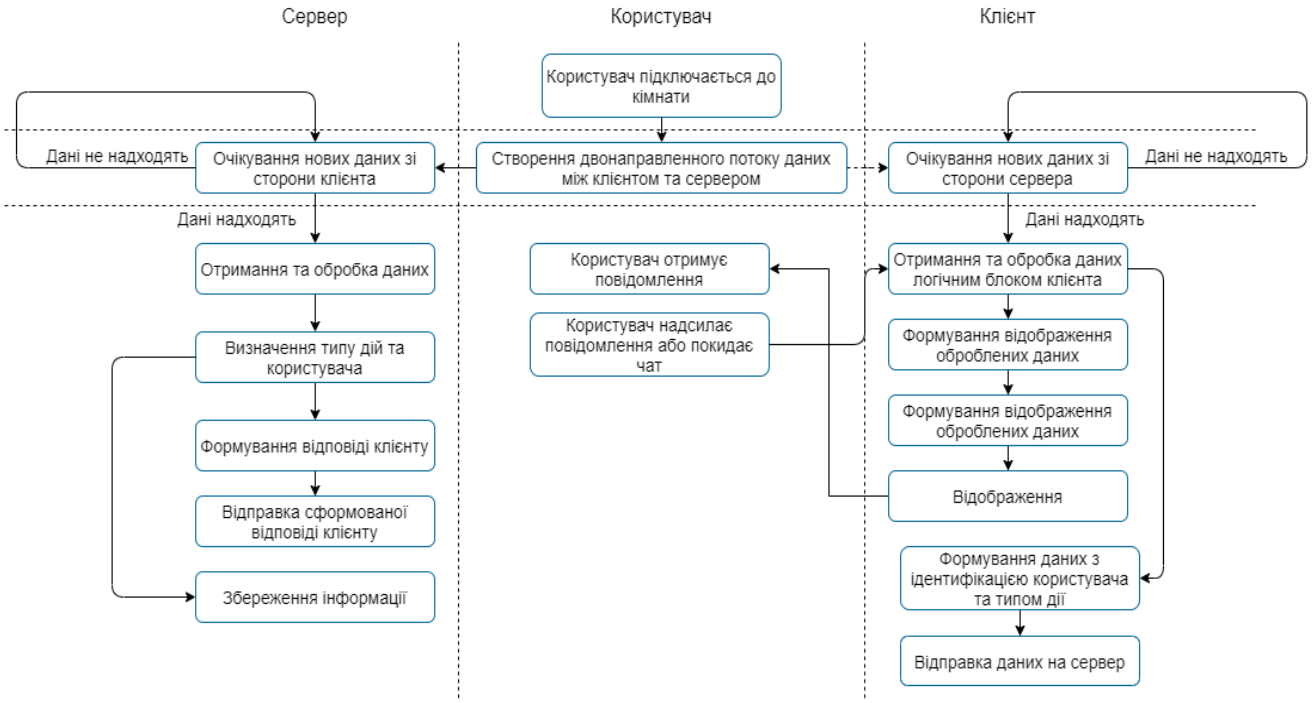

Рисунок 2.12 — Блок-схема алгоритму дій КСД на етапі, коли користувач підключився до кімнати.

Отже, в даному розділі реалізовано клієнт-серверний додаток, який можна використовувати в локальній мережі задля забезпечення потреби у комунікації.

ВИСНОВОК

В результаті виконання кваліфікаційної роботи бакалавра було організовано архітектуру локальної мережі, а також розроблено клієнт серверного додатку задля забезпечення потреб комунікації видавничого центру.

Були досягнуті основні цілі кваліфікаційної роботи бакалавра:

– локальна мережа, яка може бути масштабована і яка може бути забезпечена програмними засобами, які можуть виконувати корпоративні потреби;

– програмне забезпечення, яке відповідає вимогам надійності, масштабування, можливості підтримки, інтуїтивно зрозумілий користувачу.

У даній роботі організовано архітектуру локальної мережі для п’яти робочих станцій видавничого центру, яка може бути ізольована або обмежена в доступі до глобальної мережі Internet.

Адаптовано клієнт-серверний додаток для локальної мережі. Така адаптованість та забезпечення вимогам було досягнуто за допомогою організації архітектур клієнтської та серверної частин. А саме:

– на клієнтській стороні:
  - розроблено інтуїтивно зрозумілий інтерфейс;
  - розроблено логічний блок частини інтерфейсу;
  - розроблено архітектуру взаємодії візуальної та логічного блоків;
– на серверній стороні:
  - розроблено логічний блок, який може приймати запити клієнтської сторони;
  - розроблено логічний блок, який може маніпулювати даними для забезпечення актуальною інформацією клієнтську сторону.
– організація ефективної взаємодії клієнтської та серверної сторін.

ЛІТЕРАТУРА

1. Ахрамович В.М. Комп'ютерні мережі: навчю посіб. /В.М.Ахрамович;Націон. Акад. статистики, обліку та аудиту.-К.: ДП «Інформ.-аналіт.Агентство», 2010.- 245 с. іл. – Бібліограф.: 242.

2. Організація комп’ютерних мереж [Електронний ресурс] : підручник: для студ. спеціальності 121 «Інженерія програмного забезпечення» та 122 «Комп’ютерні науки» / КПІ ім. Ігоря Сікорського ; Ю. А. Тарнавський, І. М. Кузьменко. – Електронні текстові дані (1 файл: 45,7 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2018. – 259 с.

3. An Introduction to Computer Networks. Release 1.9.21 (Peter L Dordal, Mar 04, 2020).

3. [Сайт]. URL: https://nodejs.org/en/docs/ — Official documentation Node.js.

4. [Сайт]. URL: https://reactjs.org/ — Official documentation React.

5. [Сайт]. URL: http://expressjs.com/ — Official documentation Express.

6. [Сайт]. URL: https://getbootstrap.com/ — Official documentation Bootstrap.

7. [Сайт]. URL: https://github.com/axios/axios — Official documentation axios.

8. [Сайт]. URL: https://javascript.info/ — Official documentation JavaScript.

9. [Сайт]. URL: https://developer.mozilla.org/ — Official web-documentation.

Додаток А

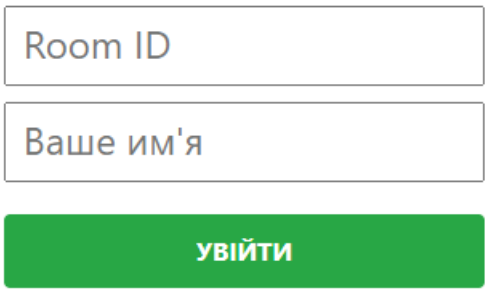

Рисунок 1 — Інтерфейс блоку ідентифікації користувача

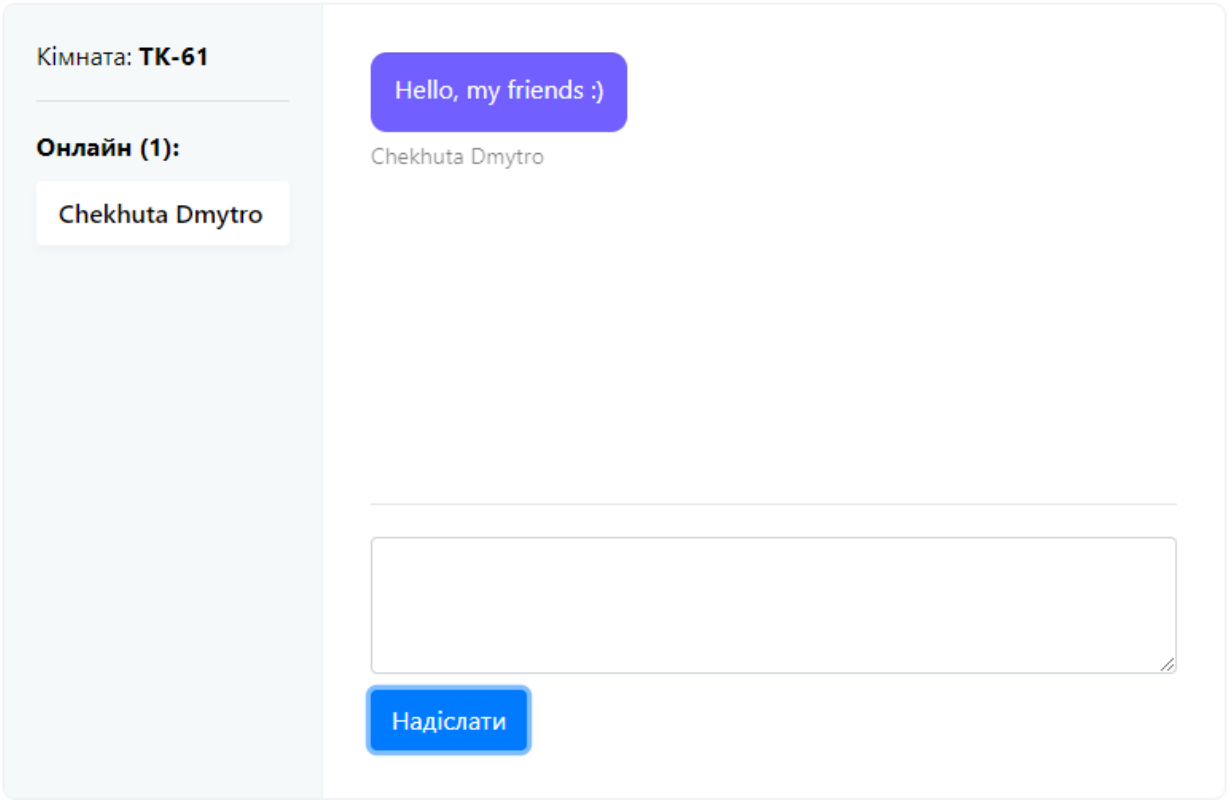

Рисунок 2 — Інтерфейс блоку обміну повідомленнями

Додаток Б

Програмний код блоку state:

```
// Створення state
const [state, dispatch] = React.useReducer(reducer, {
        joined: false,
        roomId: null,
        userName: null,
        users: [],
        messages: [],
});
```

Програмний код блоку actions:

```
// Метод при вході користувача
const onLogin = async (obj) => {
// Інформуємо клієнтську частину про успішний вхід
dispatch({
        type: 'JOINED',
        payload: obj,
});
// Запит на актуальні дані про користувачів в кімнаті
const { data } = await axios.get(`/rooms/${obj.roomId}`);
        // Внесення актуальних даних до state
dispatch({
        type: 'SET_DATA',
        payload: data,
});};
```

Програмний код блоку reducer:

```
export default (state, action) => {
        switch (action.type) {
// Дії при додаванні користувача до кімнати
                case 'JOINED':
```

```
                return {
                        ...state,
                        joined: true,
                        userName: action.payload.userName,
                        roomId: action.payload.roomId,
        };
    // Дії при ініціації оновлення даних
            case 'SET_DATA':
                return {
                        ...state,
                        users: action.payload.users,
                        messages: action.payload.messages,
                };
    // Дії при зміні інформації про користувачів кімнати
            case 'SET_USERS':
                return {
                        ...state,
                        users: action.payload,
                };
    // Дії при надходженні нових повідомлень
            case 'NEW_MESSAGE':
                return {
                        ...state,
                        messages: [...state.messages, action.payload],
                };
    // Стандартні дії
            default:
                return state;
        }
    };
```

Додаток В

Програмний блок каркасу серверної частини:

```
// Підключаємо фреймворк для функціонування HTTP-серверу
const express = require('express');
// Створюємо каркас серверного додатку
const app = express();
// Створюємо HTTP-сервер та додаємо каркас среверного додатку до сервера
const server = require('http').Server(app);
// Використання формату json при обміні даними
app.use(express.json());
// Об'єкт зберігання даних
const rooms = new Map();
// Дії сервера при надходженні йому get-запиту
app.get('/rooms/:id', (req, res) => {
        const { id: roomId } = req.params;
// Перевірка кімнати
        const obj = rooms.has(roomId)
// Якщо існує, то оновлюємо дані
                ? {
                        users: [...rooms.get(roomId).get('users').values()],
                        messages: [...rooms.get(roomId).get('messages').values()],
                }
// Якщо не існує, то створюємо нові дані
                : { users: [], messages: [] };
// Формуємо дані типу json
        res.json(obj);
});

// Дії сервера при надходженні йому post-запиту
app.post('/rooms', (req, res) => {
// Отримуємо запит та деструктктуруємо його ключи
```

```
        const { roomId, userName } = req.body;
// Перевіряємо на наявність кімнати з певним ідентифікатором
        if (!rooms.has(roomId)) {
// Якщо така кімната відсутня - створюємо
                rooms.set(
                        roomId,
                         new Map([
                                ['users', new Map()],
                                ['messages', []],
                        ]),
                );
        }
// Відповідь сервера клієнту про успішний запит зі сторони клієнта
        res.send();
});

// Прослуховування порту
server.listen(9999, (err) => {
// Перевірка на помлки
        if (err) {
// Якщо є, то повідомити про помилку
                 throw Error(err);
        }
// Якщо помилка відсутня — повідомити про успішний запуск сервера
        console.log('Server started!');
});
```

Додаток Г

Основний програмний блок серверної частини:

```
// Підключаємо фреймворк для функціонування HTTP-серверу
const express = require('express');
// Створюємо каркас серверного додатку
const app = express();
// Створюємо HTTP-сервер та додаємо каркас среверного додатку до сервера
const server = require('http').Server(app);
// Додаємо функціонал обміну даними в реальному часу (сокети) до сервера
const io = require('socket.io')(server);
// Використання формату json при обміні даними
app.use(express.json());
// Об'єкт зберігання даних
const rooms = new Map();
// Дії сервера при надходженні йому get-запиту
app.get('/rooms/:id', (req, res) => {
	const { id: roomId } = req.params;
// Перевірка кімнати
	const obj = rooms.has(roomId)
// Якщо існує, то оновлюємо дані
		? {
			users: [...rooms.get(roomId).get('users').values()],
			messages: [...rooms.get(roomId).get('messages').values()],
		}
// Якщо не існує, то створюємо нові дані
		: { users: [], messages: [] };
// Формуємо дані типу json
	res.json(obj);
});

// Дії сервера при надходженні йому post-запиту
app.post('/rooms', (req, res) => {
```

```
// Отримуємо запит та деструктктуруємо його ключи
        const { roomId, userName } = req.body;
// Перевіряємо на наявність кімнати з певним ідентифікатором
        if (!rooms.has(roomId)) {
// Якщо така кімната відсутня - створюємо
                rooms.set(
                        roomId,
                         new Map([
                                ['users', new Map()],
                                ['messages', []],
                        ]),
                );
        }
// Відповідь сервера клієнту про успішний запит зі сторони клієнта
        res.send();
});

// Описуємо алгоритм дії сокетів при підключенні
io.on('connection', (socket) => {
// Якщо користувач додається до кімнати
        socket.on('ROOM:JOIN', ({ roomId, userName }) => {
// Підключитися до кімнати з певним ідентифікатором
                socket.join(roomId);
// Додаємо до цієї кімнати користувача
                rooms.get(roomId).get('users').set(socket.id, userName);
// Визначаємо список користувачів в кімнаті
                const users = [...rooms.get(roomId).get('users').values()];
// Відправлення socket-запиту користувачам кімнати
                socket.to(roomId).broadcast.emit('ROOM:SET_USERS', users);
  });
// Якщо надходить нове повідомлення
        socket.on('ROOM:NEW_MESSAGE', ({ roomId, userName, text }) => {
```

```
                const obj = {
                userName,
                text,
                };
                rooms.get(roomId).get('messages').push(obj);
                socket.to(roomId).broadcast.emit('ROOM:NEW_MESSAGE', obj);
  });

// Якщо користувач покидає кімнату
        socket.on('disconnect', () => {
                rooms.forEach((value, roomId) => {
                if (value.get('users').delete(socket.id)) {
                const users = [...value.get('users').values()];
                socket.to(roomId).broadcast.emit('ROOM:SET_USERS', users);
                }});});
                console.log('user connected', socket.id);
        });

// Прослуховування порту
server.listen(9999, (err) => {
// Перевірка на помлки
        if (err) {
// Якщо є, то повідомити про помилку
                 throw Error(err);
        }
// Якщо помилка відсутня — повідомити про успішний запуск сервера
        console.log('Server started!');
});
```

Основний програмний блок клієнтської частини:

```
import React from 'react'; // імпорт бібліотеки React
import axios from 'axios'; // імпорт бібліотеки для створення http-запитів
```

```
import socket from './socket'; // імпорт бібліотеки для створення socket-запитів
import reducer from './reducer'; // імпорт reducer
import JoinBlock from './components/JoinBlock'; // імпорт інтерфейсу додавання
import Chat from './components/Chat'; // імпорт інтерфейсу спілкування

function App() {
const [state, dispatch] = React.useReducer(reducer, {
   joined: false,
   roomId: null,
   userName: null,
   users: [],
   messages: [],
});

const onLogin = async (obj) => {
// Інформуємо клієнтську частину про успішний вхід
        dispatch({
                type: 'JOINED',
                payload: obj,
        });
// Інформуємо серверну частину про успішний вхід
// За допомгою сокетів надсилаємо серверу тип дії: "ROOM:JOIN"
        socket.emit('ROOM:JOIN', obj);
// Запит на актуальні дані про користувачів в кімнаті
        const { data } = await axios.get(`/rooms/${obj.roomId}`);
// Внесення актуальних даних до state
        dispatch({
                type: 'SET_DATA',
                payload: data,
   });
};
// Оновлення користувачів
```

```
const setUsers = (users) => {
        dispatch({
                type: 'SET_USERS',
                payload: users,
        });
};
// Нове повідомлення
const addMessage = (message) => {
        dispatch({
                type: 'NEW_MESSAGE',
                payload: message,
        });
};
  // Отримання даних з сервера
React.useEffect(() => {
        socket.on('ROOM:SET_USERS', setUsers);
        socket.on('ROOM:NEW_MESSAGE', addMessage);
}, []);

window.socket = socket;

return (
// Відображення інтерфейсу
         <div className="wrapper">
                {!state.joined ? (
                        <JoinBlock onLogin={onLogin} />
                ) : (
                        <Chat {...state} onAddMessage={addMessage} />
     )}
        </div>
  );}
export default App;
```