Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут» імені Ігоря Сікорського
Факультет інформатики та обчислювальної техніки
Кафедра автоматики та управління в технічних системах

Лабораторна робота № 9

із дисципліни: «Технології розроблення програмного забезпечення»

на тему: «Взаємодія компонентів системи.**»**

Виконав:
студент групи ІА-34
Вінницький Г.Р.
Перевірил:
Мягкий Михайло Юрійович

Київ-2025

Мета: Вивчити види взаємодії додатків (Client-Server, Peer-to-Peer, Service oriented Architecture), та реалізувати в проєктованій системі одну із архітектур.

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати функціонал для роботи в розподіленому оточенні відповідно до обраної теми.

• Реалізувати взаємодію розподілених частин:

– *Для клієнт-серверних варіантів*: реалізація клієнтської і серверної частини додатків, а також загальної частини (middleware); зв'язок клієнтської і серверної частин за допомогою WCF, TcpClient, .NET-Remoting на розсуд виконавця.

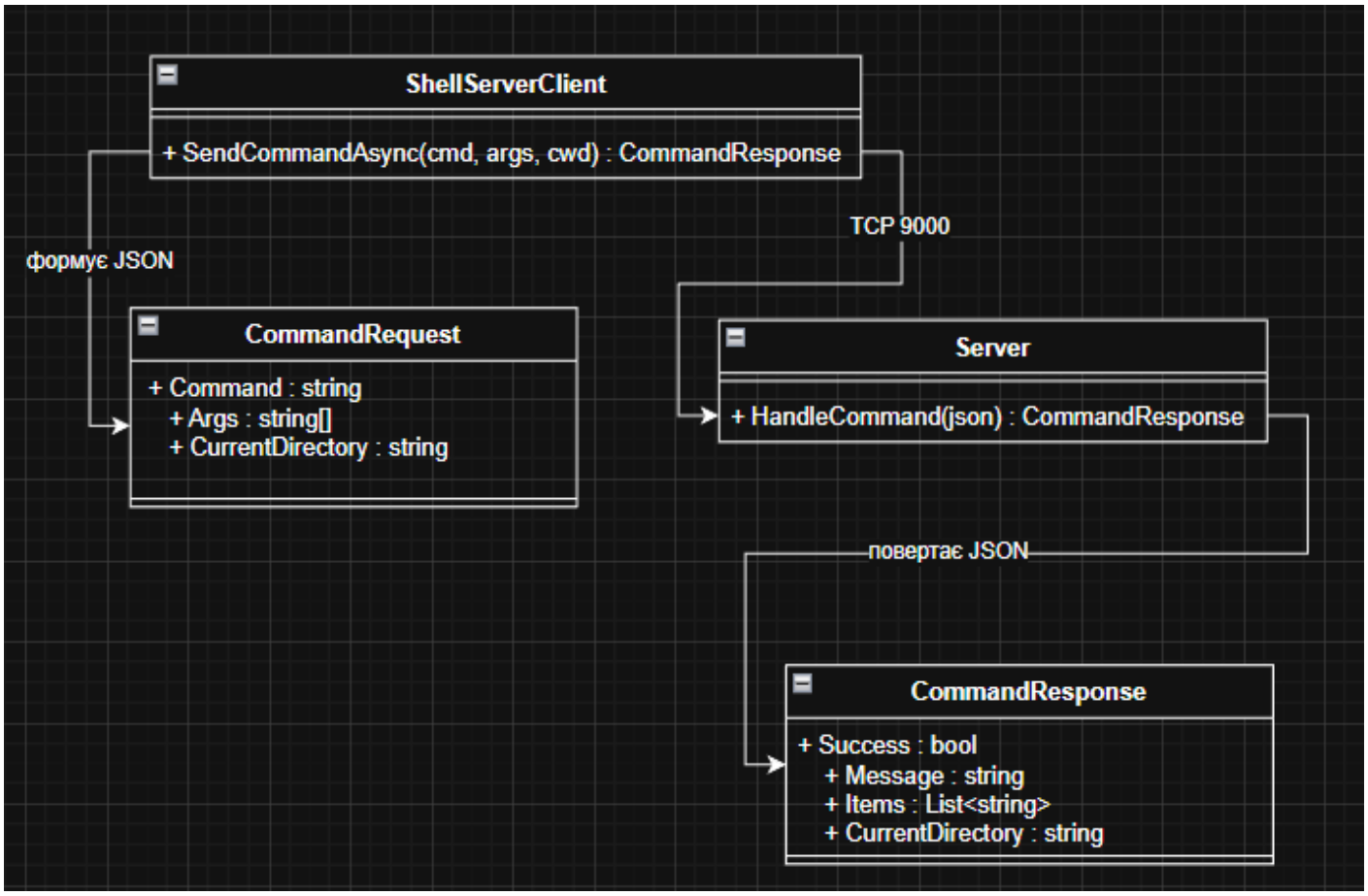
– *Для одноранговых мереж*: реалізація взаємодії клієнтських додатків за допомогою WCF Peer to peer channel.

– *Для SOA додатків*: реалізація сервісу, що надає послуги клієнтським застосуванням; викладання сервісу в хмару або підняття у вигляді Web Service на локальній машині; використання токенів для передачі даних про автентифікації, двостороннє шифрування.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє спроєктовану архітектуру. Навести фрагменти програмного коду, які є суттєвими для відображення реалізованої архітектури.

18. **Shell (total commander)** (state, prototype, factory method, template method, interpreter, client-server)

Оболонка повинна вміти виконувати основні дії в системі – перегляд файлів папок в файлової системі, перемикання між дисками, копіювання, видалення, переміщення об'єктів, пошук.



Діаграма класів клієнт-серверної взаємодії

Діаграма показує всю архітектуру клієнт–серверної частини проєкту Shell Total Commander, включаючи:

- ShellServerClient — клієнт, який формує запити
- Server — сервер, що приймає та обробляє запити
- CommandRequest / CommandResponse — структура даних
- TCP 9000 — канал передавання
- JSON — формат серіалізації

Детальний опис компонентів:

1. ShellServerClient — клієнтська частина

Це клас у додатку Shell Total Commander, який відповідає за:

1) встановлення TCP-з’єднання на порт 9000
2) формування запиту CommandRequest
3) серіалізацію його у JSON
4) надсилання запиту серверу
5) отримання JSON-відповіді
6) десеріалізацію у CommandResponse

Метод:

SendCommandAsync(cmd, args, cwd) : CommandResponse

Живе повністю на стороні клієнта (WPF/GUI застосунок).

2. CommandRequest — структура запиту

Містить мінімум необхідних даних для віддаленого виконання команди:

1) Command : string — ім’я команди (copy, ls, delete, …)
2) Args : string[] — аргументи
3) CurrentDirectory : string — поточний каталог клієнта

Саме цей об’єкт серіалізується в JSON і передається серверу.

3. Server — серверна частина

Сервер працює в окремому .NET-проєкті (ServerApp) і:

1) слухає порт 9000
2) приймає JSON від клієнта
3) десеріалізує його в CommandRequest
4) виконує команду на серверній машині
5) формує CommandResponse
6) серіалізує відповідь у JSON і надсилає назад клієнту

Метод:

HandleCommand(json) : CommandResponse

Фактично це контролер серверної системи.

4. CommandResponse — структура відповіді

Містить дані, які повертаються клієнту:

1) Success : bool — чи вдалося виконати команду
2) Message : string — текстове пояснення результату
3) Items : List<string> — список файлів/каталогів (для ls, search)
4) CurrentDirectory : string — нова поточна директорія

Цей формат уніфікований для всіх команд.

5. Механізм зв’язку (middleware)

Транспорт — TCP

Стабільне двостороннє з’єднання між клієнтом та сервером.

Формат повідомлень — JSON

Сервер і клієнт не передають «сирі» об’єкти —

обидві сторони конвертують їх у JSON за допомогою:

System.Text.Json

Обмін даними

Клієнт - JSON - TCP - Сервер

Сервер - JSON - TCP - Клієнт

Це і є middleware — рівень, що відповідає за передавання, серіалізацію, структуру та угоди комунікації.

2. Як працює клієнт–сервер у твоїй системі (пояснення алгоритму)

1. Користувач вводить команду у програмі (GUI WPF).
2. Програма перевіряє, чи доступний сервер.
3. Якщо сервер працює — команда виконується віддалено.
4. Клієнт формує CommandRequest.
5. Конвертує в JSON і надсилає серверу через TCP 9000.
6. Сервер приймає JSON і десеріалізує його.
7. Виконує команду на своїй файловій системі.
8. Формує CommandResponse.
9. Серіалізує JSON і надсилає назад.
10. Клієнт отримує відповідь і оновлює UI.

Це повноцінний робочий TCP-клієнт і TCP-сервер.

Висновок:

У ході виконання лабораторної роботи №9 було реалізовано повноцінну клієнт–серверну архітектуру файлового менеджера Shell Total Commander. Було створено два окремих застосунки:

клієнтський (WPF GUI) — для взаємодії з користувачем;

серверний (.NET TCP-сервер) — який виконує файлові команди.

Зв’язок між клієнтом і сервером реалізовано через мережевий протокол TCP, порт 9000, з використанням формату обміну JSON. Це дозволило створити власний middleware-шар, який відповідає за структуру повідомлень, серіалізацію даних та узгодження протоколу взаємодії.

Реалізація включає два основні контейнери даних — CommandRequest та CommandResponse, що забезпечують гнучкий та універсальний спосіб передавання інформації між частинами системи. Архітектура є розширюваною, модульною та

відповідає сучасним принципам побудови розподілених систем.

Таким чином, у рамках лабораторної роботи було:

1) реалізовано повний цикл клієнт–серверної взаємодії;
2) створено middleware-протокол на основі JSON;
3) продемонстровано застосування TCP-з’єднання;
4) забезпечено розмежування відповідальностей між клієнтським та серверним застосунками.

Мета лабораторної роботи була повністю досягнута.

Питання до лабораторної роботи:

1. Що таке клієнт-серверна архітектура?

Клієнт-серверна архітектура — це модель організації програмного забезпечення, у якій функціональність системи розділяється між двома основними сторонами: клієнтом і сервером.

Клієнт — це компонент, що ініціює запит, звертається до сервера за певною інформацією чи послугою та отримує результат. Сервер — це програмна або апаратна система, яка обробляє запити клієнтів, виконує необхідні дії (обчислення, пошук даних, збереження інформації) і повертає відповідь.

Основна ідея моделі полягає в централізації ресурсів і обчислень: сервери концентрують логіку та дані, тоді як клієнти виконують роль інтерфейсу й ініціатора взаємодій. Такий підхід забезпечує високу керованість, безпеку і масштабованість системи, дозволяючи обслуговувати одночасно велику кількість клієнтів.

2. Розкажіть про сервіс-орієнтовану архітектуру.

Сервіс-орієнтована архітектура (SOA — Service-Oriented Architecture) — це підхід до побудови програмних систем, у якому функціональність розділяється на незалежні, слабозв’язані сервіси. Кожен сервіс реалізує певну бізнес-функцію і може бути використаний іншими сервісами або клієнтськими програмами.

Сервіси в SOA зазвичай автономні, розробляються окремо, розгортаються незалежно та спілкуються між собою через стандартизовані інтерфейси. Такі сервіси можуть розташовуватися на різних серверах, бути написані різними мовами програмування, використовувати різні технології, але вони повинні підтримувати загальні протоколи взаємодії.

Основна перевага SOA полягає в можливості повторного використання сервісів та гнучкості побудови великих корпоративних систем, де різні компоненти можуть легко комбінуватися і масштабуватися.

3. Якими принципами керується SOA?

SOA базується на кількох ключових принципах:

1. Слабка зв’язаність — сервіси повинні мати мінімум залежностей один від одного. Зміни в одному сервісі не повинні впливати на інші.
2. Повторне використання — сервіси створюються так, щоб їх можна було використовувати в різних контекстах.
3. Автономність — сервіс сам управляє своїм станом, логікою і даними.
4. Чіткі контракти — кожен сервіс має формально описаний інтерфейс, який визначає, як ним користуватися.
5. Стандартизовані протоколи взаємодії — передача даних та виклик методів повинні здійснюватися через загальноприйняті формати, щоб сервіси могли бути сумісними.
6. Безстанова взаємодія — за можливості сервіси не зберігають інформацію про попередні запити.
7. Відкритість і доступність — сервіси можуть бути знайдені й використані через централізований реєстр або каталог.

4. Як між собою взаємодіють сервіси в SOA?

Взаємодія між сервісами в SOA відбувається через стандартизовані протоколи обміну повідомленнями. Сервіси спілкуються один із одним через інтерфейси, які описані у спеціальних контрактах. Найчастіше використовується обмін XML, JSON або SOAP-повідомленнями через протоколи HTTP/HTTPS.

Сервіси можуть викликатися синхронно (коли відправник чекає на відповідь) або асинхронно (через черги повідомлень). Завдяки такому механізму сервіси можуть розташовуватися на різних серверах і різних мережах, але при цьому взаємодія між ними залишається уніфікованою.

5. Як розробники взнають про існуючі сервіси і як робити до них запити?
У традиційній SOA існує спеціальний компонент — реєстр сервісів (UDDI, WSDL). Він виконує роль каталогу, у якому описані всі доступні сервіси, їхні інтерфейси, адреси розташування та правила виклику.
Розробники отримують доступ до цих описів і, керуючись контрактом сервісу, формують запити, які відповідають очікуваному формату.
Наприклад, веб-сервіси описуються за допомогою WSDL-файлів — формальних специфікацій, які визначають методи, їх параметри, типи даних та способи взаємодії.
Сучасні системи часто використовують REST API, де опис інтерфейсу надається через OpenAPI/Swagger.

6. У чому полягають переваги та недоліки клієнт-серверної моделі?
Переваги:
Централізоване зберігання даних: легше управляти інформацією та забезпечувати безпеку.
Легке масштабування: сервер може обслуговувати багато клієнтів.
Спрощене адміністрування: оновлення проводяться лише на сервері.
Висока продуктивність: ресурси серверів можуть бути значно потужнішими за пристрої клієнтів.
Недоліки:
Залежність від сервера: якщо сервер недоступний, недоступна вся система.
Необхідність у надійній мережевій інфраструктурі.
Потенційні проблеми з продуктивністю при високому навантаженні.
Вартість підтримки та обслуговування серверів.

7. У чому полягають переваги та недоліки однорангової моделі взаємодії?
Однорангова модель (P2P) передбачає, що всі учасники є рівноправними: кожен вузол може виконувати роль і клієнта, і сервера.
Переваги:
Відсутність центрального вузла, що робить систему стійкішу до збоїв.
Добре масштабується при великій кількості вузлів.
Ефективний розподіл навантаження між учасниками.

Менша вартість, оскільки не потрібна серверна інфраструктура.

Недоліки:

Складніші алгоритми координації та пошуку даних.

Менший контроль над безпекою та доступом.

Різна продуктивність вузлів може знижувати якість роботи системи.

Важче забезпечити цілісність і надійність даних.

8. Що таке мікро-сервісна архітектура?

Мікросервісна архітектура — це стиль проєктування програмних систем, у якому додаток розбивається на набір дрібних, автономно розгорнутих сервісів. Кожен мікросервіс виконує одну чітку бізнес-функцію, має власну базу даних або власний шар зберігання, і взаємодіє з іншими сервісами через легкі мережеві протоколи.

Особливості:

автономне розгортання та оновлення;

можливість використовувати різні технології для різних сервісів;

висока масштабованість;

підвищена стійкість до збоїв (відмова одного сервісу не зупиняє весь застосунок).

9. Які протоколи використовуються для обміну даними в мікросервісній архітектурі?

Найчастіше використовуються такі протоколи:

HTTP/HTTPS (REST API)

gRPC

AMQP (RabbitMQ)

Kafka (подієві стріми)

WebSocket для реального часу

Protobuf як формат серіалізації

JSON або XML для REST

Мікросервіси активно використовують як синхронні, так і асинхронні протоколи, залежно від задачі.

10. Чи можна назвати підхід сервіс-орієнтованою архітектурою, коли ми в проєкті між шаром веб-контролерів та шаром доступу до даних реалізуємо шар бізнес-логіки у вигляді сервісів?

Такий підхід не є сервіс-орієнтованою архітектурою (SOA) у строгому сенсі.

У типовій трирівневій або багатошаровій архітектурі бізнес-логіка дійсно реалізується у вигляді сервісів, але ці «сервіси» представляють собою звичайні класи в межах одного застосунку. Вони не є автономними компонентами, не мають власних контрактів, не розгортаються окремо і не взаємодіють через мережеві протоколи.

SOA передбачає, що кожен сервіс — це незалежна, відокремлена одиниця, яка може працювати самостійно та бути викликана віддалено.

Тому використання сервісного шару в межах одного монолітного застосунку — це просто внутрішня логічна організація, а не справжня сервіс-орієнтована архітектура.