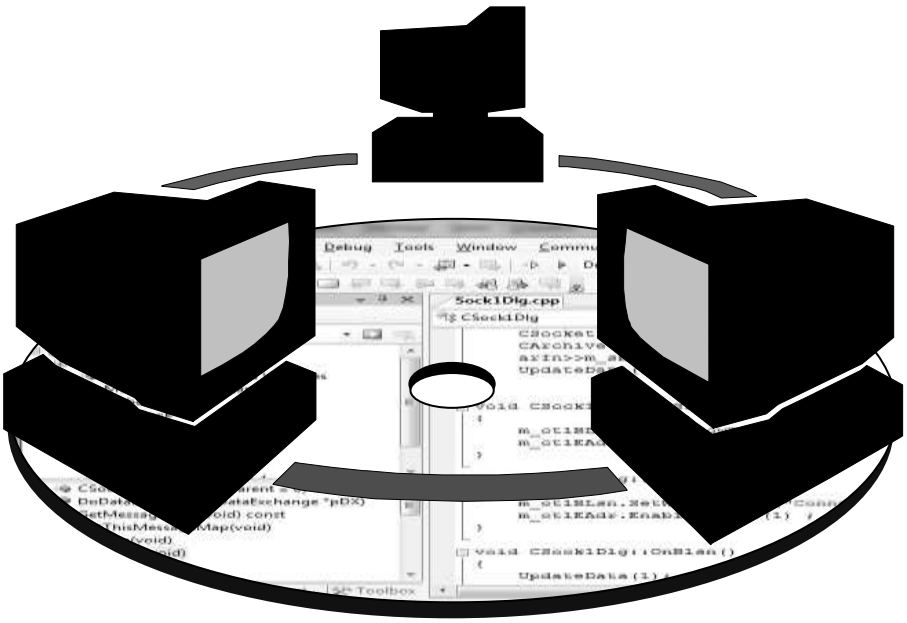
О. Д. Азаров, О. І. Черняк, Л. А. Савицька

Прикладне програмування у комп’ютерних мережах



Міністерство освіти і науки України
Вінницький національний технічний університет

О. Д. Азаров, О. І. Черняк, Л. А. Савицька

Прикладне програмування у комп'ютерних мережах

Електронний навчальний посібник
комбінованого (локального та мережного) використання

Видання друге, доповнене і перероблене

Вінниця
ВНТУ
2023

УДК 681.3.062(075)
А35

Рекомендовано до видання Вченою радою Вінницького національного технічного університету Міністерства освіти і науки України (протокол № 15 від 29.06.2023 р.)

Рецензенти:
Яремко С., к.т.н., доцент кафедри економічної кібернетики та інформаційних систем Вінницького торговельно-економічного інституту ДТЕУ;
Квєтний Р. Н., д.т.н., професор кафедри АІІТ ВНТУ;
Романюк О. Н., д.т.н., професор кафедри ПЗ ВНТУ.

Азаров, О. Д.

А35 Прикладне програмування у комп'ютерних мережах : електронний навчальний посібник комбінованого (локального та мережного) використання [Електронний ресурс]. – [Вид. 2-е, переробл. та доповн.]. / Азаров О. Д., Черняк О. І., Савицька Л. А. – Вінниця : ВНТУ, 2023. – 129 с.

Посібник орієнтовано на програмування в мережах Windows. Для вивчення матеріалу посібника потрібно мати знання з програмування мовою Visual C++. Посібник розроблено відповідно до плану кафедри та програми дисципліни «Прикладне програмування».

У навчальному посібнику подано теоретичні та практичні відомості з прикладного програмування у комп'ютерних мережах мовою Visual C++ на основі використання WinAPI. Розглянуто такі технології розробки мережевих програм: перенаправлювач, поштові скриньки, іменовані канали, інтерфейс NetBIOS, сокети.

УДК 681.3.062(075)

ЗМІСТ

ПЕРЕДМОВА

Навчальний посібник призначений для вивчення дисципліни «Прикладне програмування».

Дисципліна «Прикладне програмування» вивчається на першому курсі магістратури студентами спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання. Дисципліна базується на вивченні таких дисциплін: «Програмування», «Системне програмування», «Системне програмне забезпечення», «Візуальне програмування», «Інженерія програмного забезпечення», «Основи комп'ютерних систем та мереж». Дисципліна є завершальною у підготовці спеціалістів зі спеціальності «Комп'ютерні системи та мережі».

Внаслідок вивчення дисципліни студенти мають освоїти такі теоретичні питання: інформаційну структуру протоколів обміну даними, аспекти клієнт-серверної організації програмного забезпечення, основи організації комп'ютерних мереж, аспекти обмеження доступу до ресурсів, аспекти програмування файлів і потоків, аспекти роботи поштових скриньок, аспекти роботи іменованих каналів, властивості і особливості інтерфейсу *NetBIOS*, принципи організації обміну даними за допомогою інтерфейсу прикладного програмування *WINSOCK*, аспекти використання адрес комп'ютерів та портів, види встановлення з'єднань між комп'ютерами та відповідні протоколи, технологію введення–виведення інформації у *WINSOCK*, мовні засоби, що застосовуються під час створення прикладних програм для роботи у мережах, переваги та недоліки технологій прикладного програмування у мережах.

Крім того, студенти мають вміти створювати програми для роботи у мережі на основі потокового та дайтаграмного передавання інформації з використанням поштових скриньок, іменованих каналів та сокетів. Студенти також мають вміти створювати клієнт-серверні програми з використанням синхронного і асинхронного режимів роботи функцій.

Для успішного вивчення матеріалу цього посібника необхідно мати знання принципів організації комп'ютерних мереж та техніки програмування мовою *C++*, а також мати практичні навички з програмування у середовищі *Visual C++*.

МЕТОДИЧНІ РЕКОМЕНДАЦІЇ

У посібнику викладено теоретичні і практичні основи створення прикладних програм, що дозволяють обмінюватись інформацією між комп'ютерами, розташованими у мережі. В основу цього посібника покладено матеріали, подані у [1]. Всі приклади програм у посібнику написані мовою *Visual C++* і являють собою консольні програми (*console applications*). Принципи будови операційної системи (*operation system*) *WINDOWS* та техніка створення програм для роботи в ній, знання яких необхідні для вивчення матеріалу посібника, описані в [2]. Принципи організації роботи комп'ютерних мереж (*computer network*) з використанням протоколу *TCP/IP* описані в [3]. Техніка програмування мовою *Visual C++* описана в [4–7]. Для отримання практичних навичок з програмування у середовищі *Visual C++* можна порекомендувати [8].

Посібник складається з семи розділів.

У першому розділі коротко розглянуто аспекти організації мереж і безпеки доступу. Приділено увагу таким питанням, як мережева модель *OSI*, характеристики транспортних протоколів, мережеві протоколи, що підтримуються *Windows*, та механізми безпеки.

У другому розділі описано основні функції для роботи з файлами у *WinAPI* та їх параметри.

У подальших розділах вивчаються технології мережевого програмування на основі бібліотеки *WinAPI*. Кожен розділ описує окрему технологію.

У третьому розділі описано технологію перенаправлювача (*redirector*), що є компонентом операційної системи *Windows* і дозволяє програмам обмінюватися інформацією у мережі за допомогою вбудованих служб файлової системи, так званої мережевої операційної системи (*network operating system*, *NOS*).

У четвертому розділі описано технологію поштових скриньок, що має клієнт-серверну архітектуру і використовує перенаправлювач. Технологія поштових скриньок реалізує простий однонаправлений механізм міжпроцесного зв'язку без встановлення з'єднання. Поштові скриньки дозволяють здійснювати передавання інформації від клієнта до сервера без забезпечення надійності передавання і цілісності даних.

У п'ятому розділі описано технологію іменованих каналів. Іменовані канали – це простий механізм зв'язку між процесами (*interprocess communication*, *IPC*), підтримуваний у *Windows*. Іменовані канали

забезпечують надійне одностороннє і двостороннє передавання даних між процесами на одному або на різних комп'ютерах у режимі потоку байтів (*byte stream*) або у режимі потоку повідомлень (*message stream*).

Шостий і сьомий розділи присвячено сучасному інтерфейсу програмування мережевих програм (*network applications*) – *Winsock*. *Winsock* – це не протокол, а інтерфейс прикладного програмування мережевих взаємодій, реалізований на всіх платформах *Windows*. У цих розділах описано характеристики протоколів, що підтримуються *Windows* і, зі свого боку, підтримують інтерфейс *Winsock*. Описано режими сокетів, алгоритм встановлення з'єднання та функції, необхідні для встановлення з'єднання і виконання обміну, а також моделі введення–виведення.

У кожному розділі наводяться приклади програм, що демонструють використання відповідних технологій та контрольні питання для самоперевірки.

Для самостійної роботи з вивчення мережевого програмування за допомогою цього посібника бажано мати доступ до комп'ютерної мережі і можливість виконувати програми хоча б на двох комп'ютерах у мережі, а також відповідний дозвіл або права адміністратора. Проте, можна виконувати мережеві програми і на одному комп'ютері. Для цього потрібно у програмах вказувати локальну адресу (*local address*). На комп'ютері має бути встановлена операційна система *Windows 7* або більш високої версії. Крім того, має бути встановлено середовище програмування *Visual Studio 2010* або більш високої версії. Якщо на комп'ютері встановлено програми чи працюють служби, що обмежують можливість обміну інформацією у мережі, то потрібно зняти ці обмеження за допомогою встановлення відповідних параметрів або тимчасово відключити обмежувальні програмні засоби.

ВСТУП

Мережеве програмування є однією з центральних задач під час розробки бізнес-програм, оскільки потреба в ефективній і безпечній взаємодії різних комп'ютерів, що знаходяться в одній будівлі або розкидані по всьому світу, залишається основною для успішної діяльності більшості організацій.

З самого початку технології мережевого програмування були орієнтовані на використання у *UNIX*-подібних операційних системах. Широке використання операційної системи *Windows* і зростання ролі комп'ютерних мереж змусило *Microsoft* розробити інструментарій мережевого програмування (*network programming*) для своєї операційної системи. Потрібно відзначити, що свої засоби мережевого програмування Microsoft свідомо розробляла схожими на вже відомі в *UNIX*. Це розумна політика лідера інформаційного ринку, що дозволила значно спростити перенесення мережевих програм з однієї операційної системи на іншу.

У цьому посібнику описано основні технології мережевого програмування, доступ до яких надає бібліотека *WinAPI*. Бібліотека *WinAPI* є основним інструментальним засобом *Microsoft*, призначеним для створення ефективних *Windows*-програм. Всі технології більш високого рівня, такі як *MFC*, *DOT NET*, базуються на цій бібліотеці. Тому освоєння принципів використання *WinAPI* дозволить не тільки створювати високоефективні програми, але й краще розуміти концепцію і особливості мережевого програмування технологій більш високого рівня. Для розробки програми з використанням *WinAPI* у *Visual Studio* потрібно створити консольний додаток *C++* і підключити заголовний файл *windows.h*.

Під час розробки мережевих програм *Windows* найчастіше використовуються сокети, проте для ефективної організації обміну інформацією між комп'ютерами потрібно вміти використовувати також інші програмні засоби мережевого програмування. У цьому посібнику розглянуто основний набір таких засобів, що надається фірмою Microsoft у рамках середовища програмування *Visual C++*. Для обміну інформацією між комп'ютерами *Microsoft* надає такі основні засоби: перенаправлювач, поштові скриньки (*mailslots*), іменовані канали (*named pipes*), інтерфейс

прикладного програмування *WINSOCK*. Кожен з цих засобів ефективний у відповідних випадках – саме тому *Microsoft* внесла їх всіх до *Visual Studio*.

Перенаправлювач є основною складовою (*basic component*) набору засобів *Windows*, що відповідає за роботу з дисками, файлами і папками на віддалених комп'ютерах. Перенаправлювач працює над транспортними протоколами і здійснює перенаправлення запитів файлової системи на віддалений комп'ютер, забезпечуючи водночас досить ефективний механізм захисту від несанкціонованого доступу (*access protection*), оснований на дескрипторах безпеки (*security descriptors*). Це дозволяє програмам використовувати для доступу до файлів через мережу і роботи з ними звичайні *API*-функції для роботи з файлами типу *CreateFile*, *ReadFile* і *WriteFile*. У цьому посібнику докладно розглядається використання перенаправлювача для передавання запитів введення–виведення (*input-output query*) на віддалені пристрої (*removed devices*) (саме на цьому оснований зв'язок у технологіях поштових скриньок і іменованих каналів). Перенаправлювач – найбільш простий механізм, ефективний у випадках, коли не потрібно підтримувати надійний обмін і забезпечувати цілісність передаваної інформації.

Поштові скриньки використовують перенаправлювач і є засобом швидкого одностороннього обміну (*one-directional exchange*) інформацією за рахунок можливості записування її у спеціально створений розділ пам'яті віддаленого комп'ютера. Цей розділ і називається, власне, поштовою скринькою. Поштові скриньки реалізують клієнт-серверну архітектуру і дозволяють клієнтському процесу передавати повідомлення одному чи декільком серверним процесам. Вони також дозволяють передавати повідомлення між процесами на тому самому комп'ютері чи на різних комп'ютерах у мережі.

Розробка програм, що використовують поштові скриньки, не потребує знання таких мережевих транспортних протоколів (*network transport protocols*) як *TCP/IP* чи *IPX*. Оскільки поштові скриньки основані на архітектурі широкомовлення (*broadcasting architecture*), вони не гарантують надійного передавання даних, але корисні, коли доставка даних не є життєво важливою.

Іменовані канали (*named pipes*) також використовують

перенаправлювач. Вони надають можливість двостороннього як синхронного, так і асинхронного обміну інформацією. Розробка програм, що працюють з іменованими каналами, не є складною і не потребує особливих знань механізму роботи основних мережевих протоколів (*network protocols*) (таких як *TCP/IP* або *IPX*). Деталі роботи протоколів приховані від програми, оскільки для обміну даними між процесами через мережу іменованих каналів використовують перенаправлювач мережі *Microsoft*. До прикладів використання іменованих каналів можна віднести розробку системи керування даними, що дозволяє виконувати транзакції тільки певній групі користувачів.

Сокети є сучасним широко розповсюдженим інструментом мережевого програмування, що забезпечує можливість двонаправленого надійного передавання інформації у режимі встановлення зв'язку або швидкого передавання інформації без встановлення зв'язку. Інтерфейс *Winsock* розроблено на основі *Berceley Sockets* (*BSD*) на платформах *UNIX*, що працює з багатьма мережевими протоколами. У середовищі *Windows* такий інтерфейс став повністю незалежним від мережевих протоколів. За допомогою бібліотеки *ws2-32.lib* середовище програмування *Visual C++* надає великий набір типів і функцій *WinAPI* для тонкого керування параметрами передавання інформації у мережі.

У посібнику розглянуто основні аспекти всіх перерахованих технологій мережевого програмування та наведено конкретні приклади програм, що реалізують ці технології.

1 ОСНОВИ РОБОТИ МЕРЕЖ І БЕЗПЕКИ ДОСТУПУ

1.1 Мережева модель OSI

Модель відкритого з'єднання систем (*Open Systems Interconnect, OSI*) забезпечує високорівневе подання мережевих систем. Сім її рівнів цілком описують фундаментальні мережеві концепції: від програми до способу фізичного передавання даних. Ось ці рівні:

- прикладний – надає інтерфейс для передавання даних між програмами;
- представницький – форматує дані;
- сеансовий – керує зв'язком між двома вузлами;
- транспортний – забезпечує передавання даних (надійне або ненадійне);
- мережевий – підтримує механізм адресації між вузлами і маршрутизацію пакетів даних;
- канальний – керує взаємодією між вузлами на фізичному рівні, відповідає за групування даних, передаваних через фізичний носій;
- фізичний – керує передаванням даних у вигляді електричних сигналів.

Програми, написані у *WinAPI*, використовують функції, що звертаються до транспортного і мережевого рівнів.

1.2 Характеристики транспортних протоколів

1.2.1 Режими передавання

Транспортні протоколи можуть працювати у режимі потокового або дайтаграмного передавання даних.

Дайтаграмний режим. Дані передаються повідомленнями із збереженням їх меж. За один виклик команди записування або читання передається або приймається одне повідомлення. Такий режим використовується для обміну структурованими даними.

Потоковий режим. Дані передаються потоком без збереження їх меж. Передавач безупинно передає дані, а приймач зчитує стільки даних, скільки є у наявності, незалежно від меж повідомлень. Передавач для формування пакетів даних може розбивати повідомлення на частини чи поєднувати кілька повідомлень. Приймач у мережевому стеку накопичує

дані, що надходять для конкретної програми. Обсяг інформації, зчитаної за одну операцію, залежить від розмірів буфера у програмі.

Псевдопотік. У псевдопотоковому режимі передавач відправляє дані повідомленнями, а приймач отримує дані потоком.

1.2.2 Режими з'єднання

Транспортний протокол може працювати або у режимі із встановленням з'єднання, або у режимі без встановлення з'єднання. У першому режимі перед обміном даними встановлюється сеанс зв'язку між сторонами обміну, що гарантує існування між ними маршруту і коректний обмін інформацією. Це призводить до додаткових витрат часу і системних ресурсів. Протоколи без встановлення з'єднання не гарантують, що приймач дійсно приймає дані. Проте вони потребують менше ресурсів і працюють швидше.

1.2.3 Надійність і порядок доставки повідомлень

Надійність чи гарантована доставка забезпечується тим, що на приймальній стороні перевіряється контрольна сума кожного пакета. Порядок доставки забезпечується перевіркою порядкового номера кожного пакета на приймальній стороні.

Якщо порушено порядок приймання пакетів або якісь пакети не надійшли, чи надійшли з неправильною контрольною сумою, то відбувається повторне їх передавання. Проте забезпечення надійності і порядку ще не гарантує цілісності даних, оскільки в цьому випадку не перевіряється, чи всі пакети надійшли.

1.2.4 Коректне завершення роботи

Коректне завершення роботи підтримують тільки орієнтовані на з'єднання протоколи. Водночас одна сторона ініціює завершення сеансу зв'язку, а інша все ще може зчитувати дані, що затрималися у каналі чи у мережевому стеку. Протокол, який не підтримує коректне завершення роботи, терміново розриває сеанс зв'язку, ігноруючи будь-які дані, що не були зчитані приймачем.

У *TCP* виконується такий алгоритм коректного завершення роботи. Ініціатор завершення відправляє партнеру дайтаграму з сигналом *FIN*. Це означає, що він відправляти дані більше не буде, але продовжує їх приймати. Партнер для підтвердження відправляє дайтаграму з сигналом *ACK*, але все ще може відправляти дані. Як тільки він завершить передавання даних, то також відправляє дайтаграму з сигналом *FIN*. Ініціатор підтверджує одержання цього сигналу дайтаграмою з сигналом

ACK. Після цього сеанс зв'язку вважається завершеним.

1.2.5 Широкомовне передавання

Широкомовне передавання – це передавання даних від одного комп'ютера всім комп'ютерам цієї локальної мережі. Його підтримують лише не орієнтовані на з'єднання протоколи. Недоліком таких повідомлень є те, що всі комп'ютери локальної мережі записують їх у свої мережеві стеки і змушені витрачати час на перевірку необхідності цих повідомлень. Наслідком є високе навантаження мережі, що може суттєво сповільнити її роботу. Як правило, маршрутизатори не транслюють широкомовних повідомлень.

1.2.6 Багатоадресне передавання

Багатоадресне передавання – це передавання даних групі одержувачів. Методика приєднання програми до багатоадресного сеансу залежить від мережевого протоколу. Для *IP* воно є видозміненою формою широкомовлення, за якого комп'ютери, що обмінюються інформацією, є членами групи багатоадресного передавання. У разі приєднання до групи на мережевому адаптері комп'ютера встановлюється фільтр, що пропускає тільки ті дані, які призначені цій групі. Багатоадресне передавання використовується, наприклад, у програмах для відеоконференцій.

1.2.7 Якість обслуговування

Якість обслуговування (*Quality of Service, QoS*) дозволяє програмі зарезервувати певну частину пропускної спроможності мережі для монопольного використання. Наприклад, у відеоконференціях.

1.2.8 Фрагментарні повідомлення

Фрагментарні повідомлення (*partial message*) підтримують тільки орієнтовані на повідомлення протоколи.

Під час відправлення великого повідомлення ресурсів приймача може вистачити, щоб вмістити лише його частину. Якщо протокол підтримує фрагментарні повідомлення, то програма зможе прочитати отриманий фрагмент. Інакше мережевий стек блокує зчитування доти, доки повідомлення не надійде повністю. Якщо ж повідомлення повністю так і не надійшло, протоколи, що не підтримують фрагментарні повідомлення, просто відкинуть його.

1.2.9 Маршрутизація

Якщо протокол є маршрутизованим, то між двома робочими станціями можна встановити віртуальний канал зв'язку і передавати інформацію

незалежно від того, яка мережева апаратура їх розділяє.

1.2.10 Інші характеристики

Кожен протокол, що підтримується *Windows*, крім основних має додаткові характеристики, наприклад, порядок передавання байтів чи максимальний розмір пакета. У *Winsock 2* передбачено механізм перерахування всіх доступних протоколів та визначення їхніх характеристик.

1.3 Транспортні IP-протоколи, що підтримуються Windows

У табл. 1.1 перераховано основні доступні транспортні протоколи *IP*-мереж і їхні характеристики.

Таблиця 1.1 – Характеристики транспортних протоколів мережі *IP*

Назва протоколу	Формат інформації	Встановлення з'єднання	Надійність	Порядок пакетів	Коректне завершення сеансу	Підтримка широкомовлення	Підтримка багатоадресності	*QoS*	Макс. розмір повідомлення (байтів)
MSAFD TCP	Потік	+	+	+	+	-	-	-	Без обм.
MSAFD UDP	Повід.	-	-	-	-	+	+	-	64 КВ
RSVP TCP	Потік	+	+	+	+	-	-	+	Без обм.
RSVP UDP	Повід.	-	-	-	-	+	+	+	64 КВ

1.4 Безпека доступу

Під час спроби програми одержати доступ до ресурсів операційна система перевіряє, чи має ця програма відповідні права. Основні види доступу: читання, записування і виконання. Windows керує доступом на основі дескрипторів безпеки (*security descriptors*) і маркерів доступу (*access tokens*).

1.4.1 Дескриптори безпеки

Захищені об'єкти операційної системи мають дескриптори безпеки, які встановлюють права доступу до цих об'єктів. Дескриптор безпеки складається зі структури типу *SECURITY_DESCRIPTOR* і пов'язаної з нею

інформації, в якій міститься:

- ідентифікатор безпеки власника (*Security Identifier, SID*) – визначає власника об'єкта;
- *SID* групи – визначає основну групу, до якої входить власник об'єкта;
- список керування доступом (*Discretionary Access Control List, DACL*) – вказує, хто і який вигляд доступу (читання, записування, виконання) має до цього об'єкта;
- системний список керування доступом (*System Access Control List, SACL*) – задає види доступу, які фіксуються у журналі аудиту.

Програми не можуть безпосередньо змінювати вміст структури дескриптора безпеки. Утім, для цього можна використовувати відповідні *API*-функції.

1.4.2 Списки і записи керування доступом

Поля *DACL* і *SACL* у дескрипторі безпеки – це списки керування доступом, що містять нуль чи більше записів керування доступом (*access control entities, ACE*). Кожна *ACE* керує доступом чи здійснює контроль за доступом до об'єкта певного користувача чи групи і містить:

- *SID* користувача чи групи, до яких застосовується *ACE*;
- маску, що задає права доступу (читання, записування, виконання);
- прапорець, що позначає тип *ACE* – дозвіл на доступ, заборона на доступ чи системний аудит.

Якщо захищений об'єкт не має списку *DACL*, то усім користувачам надається повний доступ до цього об'єкта. Якщо об'єкт має непорожній *DACL*, система надає тільки ті типи доступу, що зазначені у його *ACE*. Якщо у *DACL* немає жодного *ACE*, система не надає нікому ніякого доступу.

У більшості випадків задають тільки дозвільні записи, за винятком, якщо потрібно дозволити доступ групі, але заборонити його деяким членам цієї групи. У цьому випадку заборонний запис має передувати дозвільному запису для групи, оскільки система припиняє читання, з'ясувавши режим доступу користувача чи його групи.

1.4.3 Маркери доступу

Процеси мають маркери доступу, які можуть використовуватись для перевірки прав доступу цих процесів до захищених об'єктів. Під час входу користувача *Windows* створює маркер доступу і заносить у нього *SID* користувача. Кожен процес, запущений від імені цього користувача, одержить копію маркера. У разі спроби доступу процесу до захищеного об'єкта *SID* у списках *DACL* об'єкта порівнюється з *SID* у маркері доступу процесу для визначення прав.

1.4.4 Реквізити безпеки

Реквізити користувача бувають двох типів: основні реквізити входу

(*primary login*) і реквізити сеансу (*session login*). Коли користувач реєструється на робочій станції, вводяться ім'я і пароль, що стають основними реквізитами входу і заносяться у маркер доступу. В один момент часу користувач може мати лише один набір реквізитів входу.

Під час доступу до віддаленого ресурсу реквізити входу користувача використовуються для перевірки прав доступу до цього ресурсу. Якщо доступ дозволено, *Windows* створює сеанс зв'язку і надає йому реквізити користувача. Це і є реквізити сеансу, які використовуються для створення маркера віддаленого доступу. Далі обмеження доступу до захищених об'єктів забезпечується операційною системою віддаленого комп'ютера.

1.5 Контрольні питання

1. Опишіть рівні системи мережевих протоколів *OS*.
2. Опишіть режими передавання даних транспортних протоколів.
3. Опишіть режими з'єднання у транспортних протоколах.
4. Що означає і як забезпечується надійність і порядок доставки у транспортних протоколах?
5. Для яких протоколів і в який спосіб реалізовано коректне завершення роботи?
6. Охарактеризуйте режими широкомовного і багатоадресного передавання даних. Які протоколи їх підтримують?
7. Що таке якість обслуговування і фрагментарні повідомлення? Які протоколи їх підтримують?
8. Опишіть характеристики транспортних протоколів мережі *IP*.
9. Дайте порівняльну характеристику протоколів *TCP* і *UDP*.
10. Опишіть процес встановлення операційною системою доступу до ресурсів з погляду безпеки.
11. Опишіть призначення і структуру дескрипторів безпеки. З яких полів складається структура *SECURITY_DESCRIPTOR*?
12. Опишіть призначення і структуру маркерів доступу.
13. Опишіть призначення і види реквізитів безпеки.
14. Опишіть роль списків *SACL і DACL*.
15. У якому випадку *DACL* надає повний доступ усім користувачам?
16. У якому випадку *DACL* забороняє доступ усім користувачам?
17. Опишіть структуру *ACE*. Яким чином прикладна програма може змінити запис *ACE*?
18. У яких випадках використовуються дозвільні записи в *ACE*, а у яких – заборонні? Які особливості їхнього використання?

2 ПРОГРАМУВАННЯ ФАЙЛІВ І ПОТОКІВ У WINAPI

2.1 Основи роботи з файлами у WinAPI

Для роботи з файлами у *WinAPI* використовуються три основні функції: *CreateFile*, *WriteFile*, *ReadFile*.

Функція ***CreateFile*** створює чи відкриває файл, файловий потік, каталог для файла, фізичний диск, розділ, буфер консолі, ресурси комунікацій, слот пошти або іменований канал.

```
HANDLE CreateFile(
LPCTSTR lpFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes,
DWORD dwCreationDisposition,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile);
```

Параметри функції:

lpFileName – адреса рядка, що закінчується нулем і вказує на ім'я об'єкта, який буде створений або відкритий. Щоб створити потік файла, потрібно встановити ім'я файла, двокрапку, а потім ім'я потоку.

dwDesiredAccess – тип доступу до об'єкта. Параметру можна задати будь-яку комбінацію з таких значень:

0 – запит про наявність файла на вказаному пристрої. Програма може запитати атрибути пристрою без доступу до нього;

GENERIC_READ – доступ до файла на читання. Дані можна прочитати з файла і курсор файла можна переміщувати;

GENERIC_WRITE – доступ до файла на запис. Дані можна записати у файл і курсор файла можна переміщувати.

dwShareMode – режим доступу до файла інших програм. Параметру можна задати будь-яку комбінацію з таких значень:

0 – інші програми не можуть використовувати файл.

FILE_SHARE_DELETE – інші програми можуть видаляти файл.

FILE_SHARE_READ – інші програми можуть читати з файла.

FILE_SHARE_WRITE – інші програми можуть записувати у файл.

lpSecurityAttributes – вказівник на структуру *SECURITY_ATTRIBUTES*, що визначає, чи може дескриптор, який повертає функція, успадковуватись дочірнім процесом. Якщо *lpSecurityAttributes* має значення *NULL*, дескриптор неможливо успадкували. Член структури *SECURITY_ATTRIBUTES*, який називається *LpSecurityDescriptor*, визначає дескриптор безпеки для об'єкта. Якщо *lpSecurityAttributes* має значення *NULL*, об'єкт отримує дескриптор безпеки за замовчуванням. За замовчуванням для файла або каталогу список керування доступом *ACL* у дескрипторі безпеки успадковується від батьківського каталогу.

dwCreationDisposition – режим створення файла. Параметру можна задати будь-яке з таких значень:

CREATE_ALWAYS – створення файла, навіть якщо він існує.

CREATE_NEW – створення файла, якщо він не існує. *OPEN_ALWAYS*– відкриття файла завжди. Якщо файл не існує, функція створює його.

OPEN_EXISTING – відкриття файла, якщо він існує.

TRUNCATE_EXISTING – обрізання існуючого файла до нульового розміру. Основний процес має відкрити файл з доступом *GENERIC_WRITE*.

dwFlagsAndAttributes – атрибути файлів і прапорці. Цей параметр може містити будь-яку комбінацію з таких атрибутів:

FILE_ATTRIBUTE_ARCHIVE – дозволяється архівація.

FILE_ATTRIBUTE_ENCRYPTED – файл або каталог зашифровані. Для каталогів це означає, що шифрування буде виконуватись також для новостворених у них файлів та підкаталогів. Цей прапорець не діє, якщо встановлено також прапорець *FILE_ATTRIBUTE_SYSTEM*.

FILE_ATTRIBUTE_HIDDEN – прихований файл.

FILE_ATTRIBUTE_NORMAL – ніякі атрибути файла не встановлено. Цей атрибут є дійсним, якщо інших немає.

FILE_ATTRIBUTE_NOT_CONTENT_INDEXED – файл не індексується.

FILE_ATTRIBUTE_OFFLINE – доступ до файла не відразу можна отримати. Цей атрибут вказує на те, що файл даних фізично перемістився до віддаленого сховища даних. Програма не може довільно змінювати цей атрибут.

FILE_ATTRIBUTE_READONLY – файл доступний лише для читання.

FILE_ATTRIBUTE_SYSTEM – системний файл.

FILE_ATTRIBUTE_TEMPORARY – файл для тимчасового зберігання даних.

FILE_FLAG_BACKUP_SEMANTICS – файл для резервного копіювання або відновлення роботи.

FILE_FLAG_DELETE_ON_CLOSE – файл буде знищено після закриття всіх дескрипторів.

FILE_FLAG_NO_BUFFERING – файл без системного хешування.

FILE_FLAG_OPEN_NO_RECALL – для використання віддаленого сховища даних.

FILE_FLAG_OVERLAPPED – режим асинхронного введення–виведення.

FILE_FLAG_RANDOM_ACCESS – режим довільного доступу до даних файла (а не послідовного).

FILE_FLAG_SEQUENTIAL_SCAN – режим послідовного від початку до кінця доступу до даних файла.

FILE_FLAG_WRITE_THROUGH – режим запису на диск без хешування.

Параметр *DwFlagsAndAttributes* також може задавати рівні безпеки. Якщо встановлено прапорець *SECURITY_SQOS_PRESENT*, цей параметр може мати одне або більше з таких значень:

SECURITY_CONTEXT_TRACKING – динамічний режим відслідковування безпеки.

SECURITY_ANONYMOUS – режим персоналізації клієнта на рівні анонімності.

SECURITY_DELEGATION – режим персоналізації клієнта на рівні делегування.

SECURITY_IDENTIFICATION – режим персоналізації клієнта на рівні ідентифікації.

SECURITY_IMPERSONATION – режим персоналізації клієнта на рівні виконання.

SECURITY_EFFECTIVE_ONLY – заборона режимів персоналізації клієнта.

hTemplateFile – дескриптор файла-шаблону або *NULL*. Ігнорується при відкриванні існуючого файла.

Значення, що повертаються

Якщо функція виконана успішно, то повертається відкритий дескриптор вказаного файла.

У разі помилки функція поверне *INVALID_HANDLE_VALUE.*

Щоб отримати додаткові відомості про помилку, викликайте *GetLastError()*.

Якщо вказаний файл вже існує, то функція *GetLastError()* повертає *ERROR_ALREADY_EXISTS*. Якщо файл не ще існує, то *GetLastError()* повертає *0*.

Функція ***ReadFile*** використовується для читання інформації з файла і визначена так:

```
BOOL ReadFile(
HANDLE hFile.
LPVOID lpBuffer,
DWORD nNumberOfBytesToRead,
LPDWORD lpNumberOfBytesRead,
LPOVERLAPPED lpOverlapped);
```

Параметри функції

hFile – дескриптор відкритого раніше файла.

lpBuffer – вказівник на масив, в який буде зчитана інформація з файла.

nNumberOfBytesToRead – кількість даних, що може бути зчитана з файла.

lpNumberOfBytesRead – вказівник на змінну, в яку функція після свого завершення записує кількість зчитаних байтів.

lpOverlapped – вказівник на структуру *OVERLAPPED*, яку потрібно створити для задання режиму перекритого введення–виведення.

Значення, що повертаються

У разі успішного виконання функції вона повертає значення *TRUE*, в іншому випадку – *FALSE*.

Функція ***WriteFile*** використовується для записування інформації у файл і визначена так:

```
BOOL WriteFile(
HANDLE hFile.
LPCVOID lpBuffer,
DWORD nNumberOfBytesToWrite,
LPDWORD lpNumberOfBytesWritten,
LPOVERLAPPED lpOverlapped);
```

Параметри функції

hFile – дескриптор відкритого раніше файла.

lpBuffer – вказівник на масив, в який буде зчитана інформація з файла.

nNumberOfBytesToWrite – кількість даних, що може бути записана у файл.

lpNumberOfBytesWritten – вказівник на змінну, в яку функція після свого завершення записує кількість записаних байтів.

lpOverlapped – вказівник на структуру *OVERLAPPED*, яку потрібно створити для задання режиму перекритого введення–виведення.

Значення, що повертаються

За успішного виконання функції вона повертає значення *TRUE*, в іншому випадку – *FALSE*.

У лістингу 2.1 подано приклад програми, що демонструє роботу з файлами.

Лістинг 2.1 – Програма демонстрації роботи з файлами

```
#include <iostream>
#include <windows.h>
using namespace std;
int main()
{
    HANDLE hFile;
    DWORD kilk;
    char buf[1024];
    hFile = CreateFile(L"\\\\.\\D:\\Temp\\MyFile.txt", GENERIC_WRITE, NULL, NULL,
CREATE_NEW, FILE_ATTRIBUTE_NORMAL, NULL);
    WriteFile(hFile, "Hello, world!", 15, &kilk, NULL);
    CloseHandle(hFile);
    hFile = CreateFile(L"\\\\.\\D:\\Temp\\MyFile.txt", GENERIC_READ, NULL, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    ReadFile(hFile, buf, 1024, &kilk, NULL);
    buf[kilk] = 0;
    cout << buf << endl;
    CloseHandle(hFile);
}
```

2.2 Основи роботи з потоками у WinAPI

Потоки створюються за допомогою функції ***CreateThread***, яка має такий вигляд:

```
HANDLE CreateThread(
```

LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
THREAD_START_ROUTINE lpStartAddress,
LPVOID lpParamter,
DWORD dwCreationFlags,
LPDWORD lpThreadID);

Параметри функції:

lpThreadAttributes – вказівник на структуру *SECURITY_ATTRIBUTES*. Член структури *SECURITY_ATTRIBUTES*, який називається *LpSecurityDescriptor*, визначає дескриптор безпеки для об'єкта. Якщо *lpThreadAttributes* має значення *NULL*, об'єкт отримує дескриптор безпеки за замовчуванням.

dwStackSize – розмір стека, який виділяється потоку; якщо цей параметр дорівнює 0, то розмір стека буде того ж розміру, що і у породжувального потоку.

lpStartAddress – вказівник на функцію потоку; потрібно, щоб вона мала параметр типу *LPVOID* і повертала значення типу *DWORD*, наприклад:

DWORD WINAPI ThreadFunction(LPVOID lpParametr);

lpParamter – параметр функції потоку.

dwCreationFlags – якщо цей параметр дорівнює нулю, то виконання потоку починається одразу, якщо ж він дорівнює *CREATE_SUSPENSED*, то потік створюється у неактивному режимі і його потрібно активувати, наприклад, за допомогою функції *ResumeThread()*.

lpThreadID – ідетифікатор потоку.

Значення, що повертається

Функція повертає дескриптор потоку.

Для завершення потоку викликають функцію *ExitThread()* або *TerminateThread()*.

Функція ***ExitThread*** викликається з середини потоку для нормального його завершення. Вона має вигляд:

VOID WINAPI ExitThread(DWORD dwExitCode);

Параметр функції:

dwExitCode – код, який поверне функція потоку.

Функція ***TerminateThread*** викликається ззовні потоку для його термінового закриття у разі зависання, або в інших екстрених випадках. Функція має такий вигляд:

BOOL WINAPI TerminateThread(HANDLE hThread, DWORD dwExitCode);

Параметри функції:

hThread – дескриптор потоку, який потрібно терміново завершити.

dwExitCode – код, який поверне функція потоку.

У лістингу 2.2 подано приклад програми, що демонструє роботу двох потоків.

Лістинг 2.2 – Програма демонстрації роботи двох потоків

```
#include <iostream>
#include <windows.h>
using namespace std;
DWORD WINAPI thF1(LPVOID lpParametr);
DWORD WINAPI thF2(LPVOID lpParametr);
int main()
{
    char s1[] = "Hello";
    char s2[] = ", world\n";
    DWORD thID1,thID2;
    HANDLE hThread1 = CreateThread(NULL, 0, thF1, s1, 0, &thID1);
    HANDLE hThread2 = CreateThread(NULL, 0, thF2, s2, 0, &thID2);
    system("pause");
}
DWORD WINAPI thF1(LPVOID lpParametr)
{
    for (int i = 0; i < 20; i++)
    {
        cout << (char*)lpParametr;
        Sleep(100);
    }
    ExitThread(1);
}
DWORD WINAPI thF2(LPVOID lpParametr)
{
    for (int i = 0; i < 20; i++)
    {
        cout << (char*)lpParametr;
        Sleep(100);
    }
    ExitThread(2);
}
```

2.3 Контрольні питання

1. Яка функція використовується для створення або відкриття файлів? Скільки вона має параметрів? Які типи та імена у цих параметрів? Значення якого типу повертає ця функція?
2. Яка функція використовується для запису у файл? Скільки вона має параметрів? Які типи та імена у цих параметрів? Значення якого типу повертає ця функція?
3. Яка функція використовується для читання з файла? Скільки вона має параметрів? Які типи та імена у цих параметрів? Значення якого типу повертає ця функція?
4. Яких значень може набувати параметр функції *CreateFile*, що називається *dwDesiredAccess*?
5. Яких значень може набувати параметр функції *CreateFile*, що називається *dwShareMode*?
6. Яких значень може набувати параметр функції *CreateFile*, що називається *lpSecurityAttributes*?
7. Яких значень може набувати параметр функції *CreateFile*, що називається *dwCreationDisposition*?
8. Яких значень може набувати параметр функції *CreateFile*, що називається *hTemplateFile*?
9. Який режим встановлює кожне з таких значень параметра dwFlagsAndAttributes функції *CreateFile()*: *FILE_ATTRIBUTE_HIDDEN, FILE_ATTRIBUTE_READONLY, FILE_ATTRIBUTE_ARCHIVE, FILE_ATTRIBUTE_SYSTEM FILE_ATTRIBUTE_TEMPORARY, FILE_ATTRIBUTE_ENCRYPTED*?
10. Яка функція використовується для створення потоку? Скільки вона має параметрів? Які типи та імена у цих параметрів? Значення якого типу повертає ця функція?
11. Опишіть, яких значень може набувати кожен параметр функції створення потоку?
12. Який вигляд має функція потоку? Опишіть тип, призначення і спосіб задання параметра функції потоку. Значення якого типу повертає ця функція?
13. Яка функція використовується для нормального закриття потоку? Які особливості виклику цієї функції? Які параметри має ця функція? Значення якого типу повертає ця функція?
14. Яка функція використовується для термінового закриття потоку? Які параметри має ця функція? Яке призначення цих параметрів? Значення якого типу повертає ця функція?
15. Наведіть приклад програми з використанням потоків засобами *WinAPI*.

3 ПЕРЕНАПРАВЛЮВАЧ

Windows надає можливість програмам обмінюватися інформацією у мережі за допомогою так званої мережевої операційної системи (*network operating system, NOS*), вбудованої у файлову систему. Для обміну даними використовуються функції *CreateFile*, *ReadFile* і *WriteFile*. Обмін здійснюється за архітектурою клієнт–сервер. Програма клієнта посилає запит на читання або записування локальній операційній системі, яка перенаправляє його операційній системі на сервері. Операційна система сервера передає запит драйверу пристрою, на якому знаходиться файл. Це називається перенаправленням введення–виведення (*I/O redirection*).

3.1 Універсальні правила іменування

Для доступу до файлів і пристроїв віддаленої файлової системи використовуються імена *UNC*. Вони мають вигляд

*сервер**ресурс**шлях*

сервер – ім'я віддаленого комп'ютера, на якому знаходиться файл.

ресурс – мережеве ім'я ресурсу, тобто диска або папки, до якої відкритий загальний доступ у мережі.

шлях – позначає шлях у ресурсі до файла.

3.2 Компонент мережевого доступу

Компонент мережевого доступу чи мережевий постачальник (*network provider*) – це служба, що використовує мережеві пристрої для доступу до ресурсів на віддаленому комп'ютері, наприклад, до файлів чи принтерів. Основним компонентом мережевого доступу у *Windows* є "Клієнт для мереж *Microsoft*" (*Client for Microsoft Networks*), який раніше називався *Microsoft Networking Provider* (*MSNP*).

3.3 Перенаправлювач

Компонент *MSNP* обслуговує запити на мережеве з'єднання і обмін інформацією між клієнтом і сервером за допомогою вбудованого в нього перенаправлювача, що напряму взаємодіє з рівнем мережевого транспорту та *NetBIOS*, використовуючи *UNC*-імена. Працюючи поверх мережевих протоколів, перенаправлювач дозволяє програмам обмінюватися

інформацією незалежно від фізичної організації мережі. Однак у цьому випадку необхідно, щоб відповідні комп'ютери мали хоча б один спільний мережевий і транспортний протоколи. Перенаправлювач також керує доступом до ресурсів, що є однією з форм мережевої безпеки.

3.4 Постачальник декількох UNC

Multiple UNC Provider (*MUP*) – це вказівник ресурсу, який вибирає компонент доступу для обслуговування з'єднань. В операційній системі Windows може бути одночасно декілька компонентів доступу, що можуть обслужити *UNC*-запит.

У разі надходження запиту *MUP* посилає його *UNC*-ім'я паралельно усім встановленим компонентам доступу. Якщо якийсь компонент відповідає, що здатен обслужити цей запит, то йому направляється весь запит. Якщо це можуть зробити кілька компонентів, то *MUP* вибирає того, що має найвищий пріоритет. Пріоритет компонентів визначається порядком, в якому вони були встановлені у системі і зберігається у параметрі *ProviderOrder* у реєстрі *Windows* у розділі *\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ NetworkProvider\Order.*

На рис. 3.1 зображено основні складові, що формують *UNC*-з'єднання у *NOS* в рамках *Windows*, а також показано, як передаються дані між компонентами клієнта і сервера *NOS*.

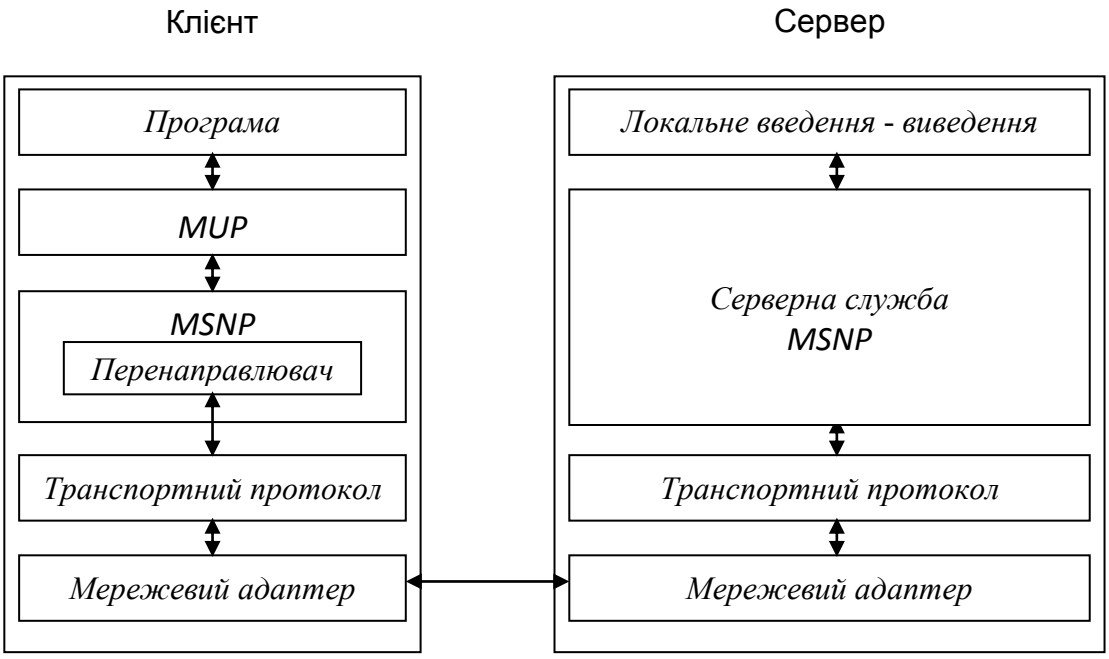


Рисунок 3.1 – Складові UNC-з'єднання

3.5 Протокол SMB

Для встановлення зв'язку з іншими комп'ютерами перенаправлювач посилає повідомлення серверній службі *MSNP* цих комп'ютерів. Такі повідомлення містяться у стандартній структурі, що називається блок повідомлення сервера (*Server Message Block, SMB*). Структура даних *SMB* складається з трьох основних компонентів: код команди, параметри команди і дані користувача.

Протокол, через який перенаправлювач відправляє та одержує повідомлення, називається протоколом використання спільних файлів на основі блоків повідомлень сервера (*Server Message Block File Sharing Protocol*) чи просто протоколом *SMB*. Протокол *SMB* базується на моделі запитів клієнта і відповідей сервера. Перенаправлювач створює структуру *SMB* із вказанням типу запиту у полі коду команди. Якщо команда потребує відправлення даних (наприклад, *SMB*-команда *Write*), то вони додаються до запиту. Потім структура *SMB* відправляється транспортним протоколом серверній службі *MSNP* на віддаленому комп'ютері, яка обробляє запит клієнта і повертає відповідну структуру *SMB*.

3.6 Приклади

Програми *Windows* можуть використовувати *API*-функції *CreateFile*, *ReadFile* і *WriteFile* для створення, відкриття і обміну файлами через мережу з використанням перенаправлювача *MSNP*. У лістингу 3.1 наведено приклад коду програми, що створює файл за допомогою *UNC*-з'єднання. Тут і далі символом ↵ позначається перенесення команди у наступний рядок.

Лістинг 3.1 – *Приклад створення файла і запису у нього тексту*

```
#include <windows.h>
#include <stdio.h>
void main(void)
{
    HANDLE FileHandle;
    DWORD BytesWritten;
    if ((FileHandle = CreateFile("\\\\.\\D:\\Temp\\Sample.txt",
    GENERIC_WRITE | GENERIC_READ, FILE_SHARE_READ |FILE_SHARE_WRITE,
NULL,CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL))==
```

```
INVALID_HANDLE_VALUE)
    {
        printf("CreateFile failed with error %d\n", GetLastError());
        return;
    }
    // Записування 14 байтів у новий файл
    if (WriteFile(FileHandle, "This is a test", 14,&BytesWritten, NULL) == 0)
    {
        printf("WriteFile failed with error %d\n", GetLastError());
        return;
    }
    if (CloseHandle(FileHandle) == 0)
    {
        printf("CloseHandle failed with error %d\n", GetLastError());
        return;
    }
}
```

3.7 Контрольні питання

1. Що таке *UNC*? Опишіть вигляд імені *UNC*.
2. Що таке компонент мережевого доступу?
3. Чим регламентується формат обміну інформацією за допомогою перенаправлювача?
4. Опишіть структуру компонент під час обміну інформацією за допомогою перенаправлювача.
5. Що таке *MUP*, *MSNP*, *SMB*?
6. Опишіть послідовність дій, що виконуються під час відкриття файла на віддаленому комп'ютері.
7. За допомогою яких функцій здійснюється створення файлів на віддаленому комп'ютері і записування у них інформації? Опишіть параметри цих функцій та приклади їх значень.
8. Напишіть програму записування інформації у файл на віддаленому комп'ютері за допомогою перенаправлювача.
9. Напишіть програму зчитування інформації з файла на віддаленому комп'ютері за допомогою перенаправлювача.
10. Де зберігається пріоритет встановлених компонент мережевого доступу?
11. Які заголовні файли потрібно підключити для доступу до файлів за допомогою перенаправлювача?

4 ПОШТОВІ СКРИНЬКИ

У *Windows* реалізовано простий однонаправлений механізм міжпроцесного зв'язку (*interpocess communication, IPC*), названий поштовими скриньками (*mailslots*). Поштові скриньки розташовуються в розділі оперативної пам'яті, іменованому як *Mailslot*. Основне обмеження поштових скриньок – вони допускають тільки ненадійне односпрямоване передавання повідомлень від клієнта до сервера. Основна перевага – клієнтські програми можуть легко посилати широкомовні повідомлення одному чи декільком серверним програмам.

4.1 Подробиці впровадження поштових скриньок

Клієнтська і серверна програми використовують стандартні функції введення–виведення файлової системи *Windows* (такі як *ReadFile* і *WriteFile*) для відправлення й одержання даних поштовою скринькою, а також правила іменування файлової системи *Windows*. Для створення й ідентифікації поштових скриньок перенаправлювач Windows використовує файлову систему *Mailslot File System (MSFS)*.

4.2 Імена поштових скриньок

Поштові скриньки іменуються за таким правилом:

\\сервер\Mailslot\[шлях]ім'я

\\сервер – ім'я сервера, на якому створюється поштова скринька і виконується серверна програма;

\\Mailslot – фіксоване обов'язкове ім'я; \[шлях]ім'я – дозволяє програмам унікальним чином визначати й ідентифікувати ім'я поштової скриньки.

\[шлях]ім'я – дозволяється задавати кілька рівнів каталогів. Наприклад, дозволяються такі імена поштових скриньок:

\\Oreo\Mailslot\Mymailslot

\\Testserver\Mailslot\Cooldirectory\Funtest\Anothermailslot

\\.\Mailslot\Easymailslot

**\Mailslot\Myslot*

Поле *server* може являти собою крапку (.), зірочку (*), ім'я домену чи сервера.

4.3 Розміри повідомлень

Для передавання повідомлень між комп'ютерами через мережу поштові скриньки, як правило, використовують дайтаграми (*datagram*), що передаються без встановлення з'єднання у тому випадку, якщо розмір повідомлення перевищує 424 байти. Повідомлення розміром більше 426 байтів передаються поштовою скринькою із встановленням з'єднання. Однак допускається з'єднання тільки одного клієнта з одним сервером.

Навіть під час встановлення з'єднання між комп'ютерами інтерфейс поштової скриньки не гарантує, що повідомлення буде дійсно записано у поштову скриньку. Наприклад, у разі відправлення повідомлення від клієнта неіснуючому серверу інтерфейс поштової скриньки не повідомить клієнтській програмі, що не зміг передати дані. За необхідності встановлення з'єднання замість поштових скриньок використовують іменовані канали.

Щоб забезпечити повну сумісність усіх платформ *Windows*, потрібно обмежити розмір повідомлень 424 байтами.

4.4 Компіляція програми у Microsoft Visual C++

В процесі розробки програми клієнта чи сервера поштових скриньок у *Microsoft Visual C++* необхідно під'єднати до програмних файлів заголовний файл *Winbase.h*. Якщо до програми під'єднано файл *Windows.h*, то *Winbase.h* можна опустити. Програма також має компонуватися з бібліотекою *Kernel32.1ib*. Під час створення консольної програми ця бібліотека підключається за замовчуванням.

4.5 Коди помилок

Усі *API*-функції *Windows*, що використовуються під час розробки клієнтів і серверів поштових скриньок (за винятком *CreateFile* і *CreateMailslot*), у випадку невдалого виконання повертають нуль. *API*-функції *CreateFile* і *CreateMailslot* повертають значення

INVALID_HANDLE_VALUE. Для одержання кодів помилок цих функцій програма має викликати функцію *GetLastError*. Повний список кодів помилок наведено у файлі *Winerror.h*.

4.6 Архітектура поштових скриньок

Поштові скриньки використовують архітектуру клієнт–сервер, у якій дані передаються тільки від клієнта до сервера. Сервер створює поштову скриньку і може читати з неї дані. Клієнт відкриває існуючу поштову скриньку і може записувати у неї дані.

4.7 Сервер поштових скриньок

Для реалізації поштової скриньки потрібно написати серверну програму, в якій виконати такі дії.

1. Створити описувач поштової скриньки за допомогою *API*-функції *CreateMailslot*.
2. Одержати дані від будь-якого клієнта шляхом виклику *API*-функції *ReadFile* з описувачем поштової скриньки як параметром.
3. Закрити описувач поштової скриньки за допомогою *API*-функції *CloseHandle*.

Серверні процеси створюють поштові скриньки за допомогою виклику *API*-функції ***CreateMailslot***, що визначена так:

```
HANDLE CreateMailslot(
LPCTSTR lpName,
DWORD nMaxMessageSize,
DWORD lReadTimeout,
LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

lpName – задає ім'я поштової скриньки. Потрібно, щоб воно мало такий вигляд:

\\.\Mai1slot\[шлях]ім'я

Тут ім'я сервера замінене крапкою, що означає локальний комп'ютер. Це є обов'язковим, оскільки не можна створити поштову скриньку на віддаленому комп'ютері. Ім'я може бути простим чи містити шлях, але має бути унікальним.

nMaxMessageSize – задає максимальний розмір (у байтах) повідомлення, що може бути записано у поштову скриньку. Якщо клієнт записує

повідомлення більшого розміру, сервер не бачить це повідомлення. Якщо задати значення нуль, то сервер буде приймати повідомлення будь-якого розміру.

lReadTimeout – задає кількість часу (у мілісекундах), протягом якого операції читання чекають вхідних повідомлень. Значення *MAILSLOT_WAIT_FOREVER* блокує читання, доки вхідні дані не стануть доступні. Якщо задати нульове значення, операції читання завершуються негайно.

lpSecurityAttibutes – задає права доступу до поштової скриньки. Необхідно, щоб він мав значення *NULL*, тому що у *Windows* система безпеки до поштових скриньок не застосовується.

Поштова скринька відносно безпечна тільки за локального введення-виведення. Коли клієнт намагається відкрити цю скриньку, він використовує точку (.) як ім'я сервера. Проте клієнт може обійти систему безпеки, задавши замість точки фактичне ім'я сервера, як за віддаленого виклику введення–виведення. Параметр *lpSecurityAttibutes* не реалізований для віддаленого введення–виведення. Таким чином, поштові скриньки тільки частково відповідають моделі безпеки *Windows*, реалізованій у стандартних файлових системах. Тому будь-який клієнт поштової скриньки у мережі може відправляти дані серверу.

Сервер – єдиний процес, що може читати дані з поштової скриньки. Для цього він має викликати функцію *ReadFile* з такими параметрами:

lpBuffer і *nNumberOfBytesToRead* – визначають, скільки даних може бути зчитано з поштової скриньки. Важливо задати розмір буфера більшим, ніж параметр *nMaxMessageSize* з *API*-виклику *CreateMailslot.* Крім того, розмір буфера має перевищувати розміри вхідних повідомлень, інакше *ReadFile* видасть код помилки *ERROR_INSUFFICIET_BUFFER*.

lpNumberOfBytes – повертає кількість зчитаних байтів після завершення роботи ReadFile.

lpOverlapped – дозволяє зчитувати дані з поштової скриньки асинхронно за допомогою перекритого введення–виведення. За замовчуванням, функція *ReadFile* блокує введення–виведення і очікує, доки дані не стануть доступні для читання.

У лістингу 4.1 подано приклад написання простого сервера поштових скриньок.

Лістинг 4.1 – Приклад сервера поштових скриньок

```
// Server.cpp
#include <windows.h>
#include <iostream>
using namespace std;
void main(void)
{
    HANDLE Mailslot;
    char buffer[256];
    DWORD NumberOfBytesRead;
    // Створення поштової скриньки
    if ((Mailslot = CreateMailslot(L"\\\\.\\Mailslot\\Myslot", 0,
    MAILSLOT_WAIT_FOREVER, NULL)) ==
    INVALID_HANDLE_VALUE)
    {
        cout<<"Failed to create a mailslot" << GetLastError() << endl;
        return;
    }
    // Нескінченне читання даних з поштової скриньки
    while(ReadFile(Mailslot, buffer, 256, &NumberOfBytesRead, NULL)
    != 0)
    {
        buffer[NumberOfBytesRead]=0;
        cout << buffer << endl;
    }
    CloseHandle(Mailslot);
}
```

4.8 Клієнт поштових скриньок

Клієнт поштової скриньки відкриває існуючу поштову скриньку і записує у неї дані. Для цього програма клієнта має виконати такі кроки.

1. Відкрити описувач поштової скриньки за допомогою *API*-функції *CreateFile*.
2. Записати дані у поштову скриньку, викликавши *API*-функцію *WriteFile*.
3. Закрити описувач поштової скриньки за допомогою *API*-функції

CloseHandle.

Клієнт відкриває описувач поштової скриньки без встановлення зв'язку. На поштові скриньки посилаються шляхом виклику *API*-функції *CreateFile* з такими параметрами:

lpFileName – ім'я поштової скриньки. Правила іменування поштових скриньок:

\\.\mailslot\ім'я – визначає локальну поштову скриньку на тому ж комп'ютері;

\\ім'я_сервера\mailslot\ім'я – визначає віддалений сервер поштової скриньки з ім'ям ім'я_сервера;

\\ім'я_домену\mailslot\ім'я – визначає всі поштові скриньки з іменем ім'я у домені ім'я_домену;

**\mailslot\ім'я* – визначає всі поштові скриньки з іменем ім'я в основному домені системи.

dwDesiredAccess – потрібно, щоб мав значення *GENERIC_WRITE*, тому що клієнт може тільки записувати дані на сервер.

dwShareMode – потрібно, щоб мав значення *FILE_SHARE_READ*, дозволяючи серверу відкривати і виконувати операції читання з поштової скриньки.

lpSecurityAttibutes – не впливає на поштові скриньки, йому потрібно задати значення *NULL*.

dwCreationDisposition – має дорівнювати *OPEN_EXISTING*. Цей параметр не має значення, якщо сервер працює віддалено.

dwFlagsAndAttributes – потрібно, щоб мав значення *FILE_ATTRIBUTE_NORMAL*, а *hTemplateFile* – значення *NULL*. Якщо сервер не створив поштову скриньку, *API*-функція *CreateFile* поверне помилку.

Після успішного відкриття описувача клієнт може записувати дані у поштову скриньку за допомогою функції *WriteFile* з такими параметрами:

hFile – описувач, що повертається функцією *CreateFile*.

lpBuffer – вказівник на буфер відправлення.

nNumberOfBytesToWrite – визначає, скільки байтів буде відправлено від клієнта серверу. Максимальний розмір повідомлення становить 64 байти. Якщо описувач поштової скриньки створений з іменем домену чи з зірочкою, розмір повідомлення не має перевищувати 424 байти, інакше функція *WriteFile* поверне помилку *ERROR_BAD_NETPATH* (щоб дізнатися код

помилки, потрібно викликати функцію *GetLastError*).

lpNumberOfBytesWritten – вказівник на змінну, в яку функція повертає кількість байтів, відправлених серверу після завершення.

lpOverlapped – має дорівнювати *NULL*. Оскільки поштові скриньки обмінюються даними без встановлення з'єднання, функція *WriteFile* не блокує введення–виведення.

У лістингу 4.2 наведено приклад простого клієнта поштової скриньки.

Лістинг 4.2 – Приклад клієнта поштової скриньки

```
// Client.cpp
#include <windows.h>
#include <iostream>
using namespace std;
void main()
{
    HANDLE Mailslot;
    DWORD BytesWritten;
    if ((Mailslot = CreateFile(L"\\\\.\\Mailslot\\Myslot", GENERIC_WRITE,
    FILE_SHARE_READ, NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL,NULL)) == INVALID_HANDLE_VALUE)
    {
        cout<<"CreateFile failed with error " << GetLastError()<<endl;
        return;
    }
    if (WriteFile(Mailslot, "This is a text from client ", 14, &BytesWritten, NULL)
== 0)
    {
        cout<<"WriteFile failed with error " << GetLastError() << endl;
        return;
    }
    cout << "Wrote " << BytesWritten << " bytes" << endl;
    CloseHandle(Mailslot);
}
```

4.9 Додаткові API-функції поштових скриньок

У програмі сервера поштової скриньки можна використовувати дві додаткові функції для взаємодії з поштовою скринькою: *GetMailslotInfo* і *SetMailslotInfo*. Функція ***GetMailslotInfo*** надає інформацію про розмір повідомлення, яке надійшло у поштову скриньку. Це може застосовуватися

для опитування наявності вхідних даних, а також дозволяє налаштовувати буфери для вхідних повідомлень різної довжини:

BOOL GetMailslotInfo(
HANDLE hMailslot,
LPDWORD lpMaxMessageSize,
LPDWORD lpNextSize,
LPDWORD lpMessageCount,
LPDWORD lpReadTimeout);

hMailslot – вказівник на поштову скриньку, повернутий викликом функції *CreateMailslot.*

lpMaxMessageSize – задає максимальний розмір повідомлення (у байтах), що може бути у поштовій скриньці.

lpNextSize – вказує на реальний розмір повідомлення (у байтах) у поштовій скриньці.

lpMesssageCount – вказівник на буфер, куди записується загальна кількість повідомлень, що очікують прочитання. Цей параметр також використовують для перевірки наявності даних.

lpReadTimeout – вказує на значення тайм-ауту (у мілісекундах), протягом якого операція читання чекає записування повідомлення у поштову скриньку.

Повернення функцією значення *MAILSLOT_NO_MESSAGE* вказує, що поштова скринька не має даних. Сервер використовує цей параметр для неблокувальної перевірки наявності вхідних даних. Але краще для цього використовувати перекрите введення–виведення *Windows*.

Функція ***SetMailslotInfo*** задає значення тайм-ауту для поштової скриньки, протягом якого операція читання очікує вхідних повідомлень. У такий спосіб програма може змінити режим читання з блокувального на неблокувальний чи навпаки. Функція *SetMailslotInfo* визначена так:

BOOL SetMailslotInfo(HANDLE hMailslot, DWORD ReadTimeout);

hMailslot – вказівник на поштову скриньку, повернутий викликом API-функції *CreateMailslot*.

ReadTimeout – визначає кількість часу (у мілісекундах), протягом якого операція читання очікує записування повідомлення у поштову скриньку. Якщо воно дорівнює нулю, то під час відсутності повідомлень операції читання повертаються негайно, якщо *MAILSLOT_WAIT_FOREVER* – будуть чекати нескінченно довго.

4.10 Контрольні питання

1. Що таке поштова скринька? Які переваги і недоліки поштових скриньок?
2. Опишіть обмеження на розмір повідомлень для поштових скриньок та напрямок обміну інформацією за допомогою поштових скриньок.
3. Які дії має виконувати програма, що є сервером поштової скриньки?
4. Наведіть правила і приклади іменування поштових скриньок.
5. Яка функція записує дані у поштову скриньку? Опишіть її параметри.
6. Які заголовні файли потрібно під'єднати під час написання програми поштової скриньки мовою *Visual C++*?
7. Напишіть програму сервера поштової скриньки.
8. За допомогою якої функції можна встановити чи є у поштовій скриньці вхідні повідомлення? Опишіть параметри цієї функції.
9. Які дії має виконувати програма, що є клієнтом поштової скриньки?
10. Напишіть програму клієнта поштової скриньки.
11. За допомогою якої функції можна встановити для поштової скриньки час очікування вхідних повідомлень? Опишіть параметри даної функції.
12. За допомогою якої функції сервер створює поштову скриньку? Опишіть параметри даної функції.
13. За допомогою якої функції клієнт отримує доступ до поштової скриньки? Опишіть параметри цієї функції.
14. За допомогою яких функцій клієнт і сервер поштової скриньки обмінюються інформацією? Опишіть параметри цих функцій.

5 ІМЕНОВАНІ КАНАЛИ

Іменовані канали – це простий механізм зв'язку між процесами у *Windows*, що використовує файлову систему іменованих каналів *Windows* (*Named Pipe File System, NPFS*) для забезпечення одностороннього або двостороннього передавання даних між процесами на одному або різних комп'ютерах. Їхнє використання не потребує знань механізму роботи мережевих і транспортних протоколів. Іменовані канали основані на архітектурі клієнт-сервер та дозволяють скористатися вбудованими можливостями захисту *Windows*. Основна відмінність сервера від клієнта полягає в тому, що тільки сервер може створити іменований канал і прийняти з'єднання з клієнтом. Клієнтська програма ініціює з'єднання з існуючим сервером. Для обміну даними сервер і клієнт викликають *API*-функції *ReadFile()* і *WriteFile()*.

Іменовані канали використовують два режими передавання даних – побайтовий режим і режим повідомлень. У першому режимі повідомлення передаються неперервним потоком байтів між клієнтом і сервером. Це означає, що клієнт і сервер точно не знають, скільки байтів прочитується або записується в канал у певний момент часу. Тому запис однієї кількості байтів за одну операцію з одного боку каналу не означає читання тієї самої кількості байтів за одну операцію з іншого боку. У другому режимі клієнт і сервер відправляють і приймають дані дискретними блоками, в цьому випадку кожне повідомлення прочитується повністю.

Прикладом використання іменованих каналів може бути розробка програми керування базою даних, що дозволяє виконувати транзакції тільки певній групі користувачів.

5.1 Правила іменування каналів

Обмін даними в іменованих каналах здійснюється за допомогою перенаправлювача. Назви іменованих каналів мають відповідати формату *Universal Naming Convention* (*UNC*). Імена каналів мають такий формат

\\server\Pipe\ [шлях/]ім'я,

де *\\сервер* – ім'я сервера, на якому створений іменований канал,

\Pipe – фіксований обов'язковий рядок,

[шлях]ім'я – унікальне ім'я каналу. Ім'я сервера може бути подане точкою.

Ось приклади правильних назв іменованих каналів:

\\myserver\PIPE\mypipe

\\Testserver\pipe\cooldirectory\funtest\jim

\\.\Pipe\Easynamedpipe

5.2 Особливості програмування

Під час створення клієнта або сервера іменованого каналу необхідно під'єднати до програмних файлів заголовний файл *Winbase.h*. або *Windows.h*. Крім того, має бути під'єднана бібліотека *Kernel32.1ib*.

Всі API-функції *Windows* (окрім *CreateFile* і *CreateNamedPipe*), що використовуються для розробки сервера і клієнта іменованого каналу, у разі виникнення помилки повертають нуль. Функції *CreateFile* і *CreateNamedPipe* у разі помилки повертають *INVALID_HANDLE_VALUE*.

5.3 Деталі реалізації сервера

Сервер створює екземпляр іменованого каналу і отримує його описувач, який використовує для встановлення з'єднання з локальними або віддаленими клієнтами і обміну з ними даними. Процес розробки простого сервера полягає у послідовному використанні *API*-функцій:

- *CreateNamedPipe* – для створення екземпляра іменованого каналу;
- *ConnectNamedPipe* – для прослуховування клієнтських з'єднань;
- *ReadFile* і *WriteFile* – для отримання і відправлення даних;
- *DisconnectNamedPipe* – для завершення з'єднання;
- *CloseHandle* – для закриття описувача іменованого каналу.

Спочатку сервер має створити екземпляр іменованого каналу за допомогою *API*-функції ***CreateNamedPipe***:

```
HANDLE CreateNamedPipe(
LPCTSTR lpName,
DWORD dwOpenMode,
DWORD dwPipeMode,
DWORD nMaxInstances,
DWORD nOutBufferSize,
DWORD nInBufferSize,
DWORD nDefaultTimeOut,
LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

lpName – визначає назву іменованого каналу, що відповідає формату *UNC*. Якщо ім'я сервера подане точкою, це означає, що сервером є локальний комп'ютер. Іменований канал не можна створити на віддаленому комп'ютері.

dwOpenMode – визначає напрямок передавання, керування введенням–виведенням і безпеку каналу. У табл. 5.1 описано прапорці, комбінації яких використовують в процесі створення каналу.

Таблиця 5.1 – Прапорці режимів створення іменованого каналу

Режим відкриття	Прапорець	Опис
Напрямок	*PIPE_ACCESS_DUPLEX*	Канал двонаправлений: серверні і клієнтські процеси можуть приймати і відправляти дані через канал
	PIPE_ACCESS_OUTBOUND	Дані передаються тільки від сервера до клієнта
	PIPE_ACCESS_INBOUND	Дані передаються тільки від клієнта до сервера
Керування введенням–виведенням	*FILE_FLAG_WRITE_THROUGH*	Функції записування не повертають значення, доки дані передаються мережею або знаходяться у буфері віддаленого комп'ютера. Застосовується тільки у побайтовому режимі
	FILE_FLAG_OVERLAPPED	Дозволяє використовувати перекрите введення-виведення під час виконання операцій читання, записування і з'єднання
Безпека	*WRITE_DACL*	Дозволяє програмі змінювати список *DACL* іменованого каналу
	ACCESS_SYSTEM_SECURITY	Дозволяє програмі змінювати список *SACL* іменованого каналу
	WRITE_OWNER	Дозволяє програмі змінювати власника іменованого каналу і групового *SID*

На рис. 5.1 зображено комбінації прапорців і напрямки передавання між сервером і клієнтом в операціях введення-виведення.

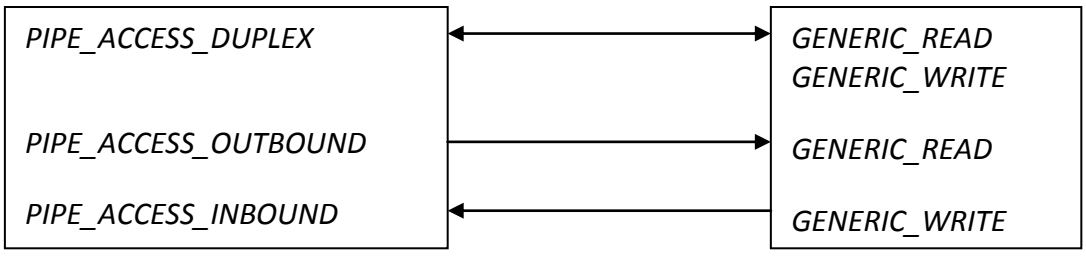


Рисунок 5.1 – Прапорці режимів і напрямків передавання

dwPipeMode – визначає режими для операцій читання, записування і очікування. В процесі створення каналу потрібно вказати по одному прапорцю з кожної категорії параметрів, об'єднавши їх операцією *OR*, як показано у табл. 5.2.

Таблиця 5.2 – Прапорці режимів читання-записування іменованого каналу

Режим	Прапорець	Опис
Записування	*PIPE_TYPE_BYTE*	Дані записуються у канал потоком байтів
	PIPE_TYPE_MESSAGE	Дані записуються у канал потоком повідомлень
Читання	*PIPE_READMODE_BYTE*	Дані зчитуються з каналу потоком байтів
	PIPE_READMODE_MESSAGE	Дані зчитуються з каналу потоком повідомлень
Очікування	*PIPE_WAIT*	Ввімкнено режим блокування
	PIPE_NOWAIT	Вимкнено режим блокування

Для створення каналу у побайтовому режимі потрібно вказати прапорці *PIPE_READMODE_BYTE* | *PIPE_TYPE_BYTE* у функції *CreateNamedPipe*. У цьому випадку не обов'язково врівноважувати кількість операцій читання і записування. Наприклад, якщо у канал записано 500 байтів, одержувач може прочитувати по 100 байтів до тих пір, доки не прочитає всі дані.

Для створення каналу у режимі повідомлень потрібно вказати прапорці *PIPE_READMODE_MESSAGE* | *PIPE_TYPE_MESSAGE*. У цьому випадку необхідно врівноважувати кількість операцій читання і записування даних. Наприклад, якщо у канал записано 500 байтів, то для читання даних буде потрібно буфер розміром 500 байтів або більше, інакше функція ReadFile видасть помилку *ERROR_MORE_DATA*. Комбінування прапорця *PIPE_TYPE_MESSAGE* з прапорцем *PIPE_READMODE_BYTE* дозволяє відправляти дані у режимі повідомлень і зчитувати довільну кількість байтів, ігноруючи роздільники повідомлень. Прапорець *PIPE_TYPE_BYTE* не можна використовувати з прапорцем *PIPE_READMODE_MESSAGE*, тому що функція *CreateNamedPipe* видасть помилку *ERROR_INVALID_ PARAMETER*. Прапорці

PIPE_WAIT і PIPE_NOWAIT переводять канал у блокувальний і неблокувальний режими. Їх можна комбінувати з прапорцями режиму читання і записування. У режимі блокування, який встановлюється за замовчуванням, операції введення–виведення зупиняють програму доки не будуть виконані повністю. Проте для асинхронного виконання функцій *ReadFile* і *WriteFile* використовується перекрите введення-виведення.

nMaxInstances – визначає максимальну кількість екземплярів (описувачів) каналу, що одночасно можуть бути створені сервером. Екземпляр каналу – це з'єднання клієнта з сервером іменованого каналу. Параметр може набувати значення від 1 до *PIPE_UNLIMITED_INSTANCES*. Якщо параметр дорівнює *PIPE_UNLIMITED_INSTANCES*, кількість екземплярів каналу обмежена тільки системними ресурсами.

nOutBufferSize і *nInBufferSize* – визначають розміри вхідного і вихідного внутрішніх буферів. Під час створення екземпляра каналу система кожного разу формує вхідний і вихідний буфери, використовуючи резидентний пул (фізична пам'ять операційної системи). Розмір буфера має бути не дуже великим, щоб система не вичерпала резидентний пул, і не дуже малим, щоб виконувати запити введення-виведення. За спроби записати дані більшого розміру система спробує автоматично розширити об'єм буфера. Оптимальні розміри – ті, які програма використовує під час виклику функцій *ReadFile* і *WriteFile*.

nDefaultTimeOut – задає час очікування з'єднання у мілісекундах. Параметр поширюється тільки на клієнтські програми, які використовують функцію *WaitNamedPipe*.

lpSecurityAttributes – вказівник на дескриптор безпеки іменованого каналу, який визначає, чи зможе дочірній процес успадковувати створений описувач. Якщо цей параметр дорівнює *NULL*, іменований канал використовує стандартний дескриптор безпеки, а описувач не може бути успадкований. Стандартний дескриптор безпеки означає, що іменований канал має ті самі права доступу, що і процес, який його створив. Програма може надати доступ до каналу певним користувачам і групам, визначивши структуру *SECURITY_DESCRIPTOR* за допомогою API-функцій. Щоб відкрити доступ до сервера будь-яким клієнтам, у структурі *SECURITY_DESCRIPTOR* потрібно задати порожній список *DACL*.

Після створення каналу сервер починає чекати з'єднання клієнтів за

допомогою функції ***ConnectNamedPipe***, що визначена так:

```
BOOL ConnectNamedPipe(
HANDLE hNamedPipe,
LPOVERLAPPED lpOverlapped);
```

hNamedPipe – описувач екземпляра іменованого каналу, повернутий функцією *CreateNamedPipe.*

lpOverlapped – дозволяє функції виконуватися асинхронно, якщо прапорець *FILE_FLAG_OVERLAPPED* використовувався під час створення каналу. Якщо цей параметр дорівнює *NULL*, функція *ConnectNamedPipe* блокується до тих пір, доки не встановиться з'єднання клієнта з сервером.

Сервер відправляє і приймає дані за допомогою функцій *WriteFile* і *ReadFile*. Після завершення обміну даними сервер закриває з'єднання за допомогою функції *DisconnectNamedPipe*. У лістингу 5.1 подано програму сервера, що обмінюється даними з одним клієнтом.

Лістинг 5.1 – Сервер іменованого каналу

```
//Server.cpp
#include <windows.h>
#include <iostream>
#define PIPE_NAME L"\\\\.\\Pipe\\Jim"
using namespace std;
void main(void)
{
    HANDLE PipeHandle;
    DWORD BytesRead;
    CHAR buffer[256];
    if ((PipeHandle = CreateNamedPipe(PIPE_NAME, PIPE_ACCESS_DUPLEX,
PIPE_TYPE_BYTE | PIPE_READMODE_BYTE, 1, 0, 0, 1000, NULL)) ==
INVALID_HANDLE_VALUE)
    {
        cout <<"CreateNamedPipe failed with error: " << GetLastError() << endl;
        return;
    }
    cout << "Server is now running" <<endl;
    if(ConnectNamedPipe(PipeHandle, NULL)==0)
```

```
    {
        cout << "Error of ConnectNamedPipe (" << GetLastError() << ")\n";
        CloseHandle(PipeHandle);
        return;
    }
    if(ReadFile(PipeHandle, buffer,sizeof(buffer),&BytesRead,NULL)<=0)
    {
        cout<<"ReadFile failed with error: " << GetLastError() << endl;
        CloseHandle(PipeHandle);
        return;
    }
    buffer[BytesRead]=0;
    cout<< buffer;
    if(DisconnectNamedPipe(PipeHandle)==0)
    {
        cout << "DisconnectNamedPipe failed with error:" <<
        GetLastError() << endl;
        CloseHandle(PipeHandle);
        return;
    }
    CloseHandle(PipeHandle);
}
```

5.4 Персоналізація

Windows підтримує так звану персоналізацію, що дозволяє серверу виконувати команди у контексті безпеки клієнта. Як правило, сервер іменованого каналу працює у контексті безпеки процесу, що його запустив. Наприклад, якщо сервер іменованого каналу запущений користувачем з привілеями адміністратора, то ці привілеї надаються будь-якому клієнту. Встановивши з'єднання з клієнтом за допомогою функції *ConnectNamedPipe*, сервер може обмежити права клієнтів, змусивши свою програму виконуватися у контексті безпеки клієнта за допомогою функції ***ImpersonateNamedPipeClient***:

BOOL ImpersonateNamedPipeClient(HANDLE hNamedPipe);

де *hNamedPipe* – це описувач екземпляра іменованого каналу,

повернутий функцією *CreateNamedPipe*.

Під час роботи у контексті безпеки клієнта сервер використовує один з чотирьох основних рівнів персоналізації:

- анонімний (*Anonymous*);
- ідентифікація (*Identification*);
- персоналізація (*Impersonation*);
- делегування (*Delegation*).

Рівні персоналізації задаються клієнтом і визначають ступінь, до якого сервер має право представляти клієнта. Завершивши сеанс зв'язку, сервер має викликати функцію ***RevertToSelf***, щоб повернутися до початкового контексту безпеки:

BOOL RevertToSelf(VOID);

5.5 Деталі реалізації клієнта

Клієнти можуть встановлювати з'єднання з існуючими на сервері каналами. Для створення простого клієнта потрібно виконати такі дії.

- Для перевірки наявності вільного екземпляра каналу викликати *API*-функцію *WaitNamedPipe*.
- Для встановлення з'єднання викликати *API*-функцію *CreateFile*.
- Для відправлення й одержання даних викликати *API*-функції *WriteFile* і *ReadFile*.
- Для завершення з'єднання викликати *API*-функцію *CloseHandle*.

Перед встановленням з'єднання клієнт має перевірити наявність на сервері вільного екземпляра іменованого каналу за допомогою функції ***WaitNamedPipe***:

BOOL WaitNamedPipe(
LPCTSTR lpNamedPipeName,
DWORD nTimeOut);

lpNamedPipeName – визначає назву іменованого каналу у форматі *UNC*.

nTimeOut – скільки часу клієнт буде чекати вільного екземпляра каналу.

У випадку успішного виконання функції *WaitNamedPipe* потрібно відкрити екземпляр іменованого каналу за допомогою *API*-функції ***CreateFile***:

HANDLE CreateFile(

LPCTSTR lpFileName,
DWORD dwDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttibutes,
DWORD dwCreationDisposition,
DWORD dwFlagsAndAttributes,
HANDLE hTemplateFile);

lpFileName – назва каналу, що відкривається, у форматі *UNC*.

dwDesiredAccess – задає режим доступу і має дорівнювати *GENERIC_READ* читання чи *GENERIC_WRITE* –записування даних у канал. Можна вказати обидва прапорці, об'єднавши їх за допомогою операції *OR*. Режим доступу має відповідати напрямку передавання (параметр *dwOpenMode*), заданому під час створення каналу. Наприклад, якщо канал створений із прапорцем *PIPE_ACCESS_INBOUND*, то клієнт вказує прапорець *GENERIC_WRITE*.

Параметр *dwShareMode* має дорівнювати нулю, оскільки у кожен момент часу тільки один клієнт може одержати доступ до екземпляра каналу. Потрібно, щоб параметр *lpSecurityAttibutes* мав значення *NULL*, якщо не потрібно, щоб дочірній процес успадковував описувач клієнта. Цей параметр не можна використовувати для керування доступом, тому що за допомогою функції *CreateFile* неможливо створити екземпляри іменованого каналу.

dwCreationDispcistion – варто визначити як *OPEN_EXISTING*, тоді функція *CreateFile* буде повертати помилку, якщо каналу не існує.

dwFlagsAndAttributes – обов'язково має містити прапорець *FILE_ATTRIBUTE_NORMAL*. Крім того, можна вказати прапорці *FILE_FLAG_WRITE_THROUGH*, *FILE_FLAG_OVERLAPPED* і *SECURITY_SQOS_PRESENT*, об'єднавши їх логічною операцією *OR*. Прапорці *FILE_FLAG_WRITE_THROUGH* і *FILE_FLAG_OVERLAPPED* використовуються для керування операціями введення–виведення. Прапорець *SECURITY_SQOS_PRESENT* визначає рівень персоналізації клієнта на сервері іменованого каналу. Клієнт вказує цю інформацію під час під'єднання до сервера. Якщо клієнт задає прапорець *SECURITY_SQOS_PRESENT*, він також має вказати один чи кілька прапорців, перерахованих у наведеному далі списку.

- *SECURITY_ANONYMOUS* – рівень анонімності. Сервер не може одержати інформацію про клієнта, але може виконуватися у контексті безпеки клієнта на локальному комп'ютері сервера.

- *SECURITY_IDENTIFICATION* – рівень ідентифікації. Сервер може одержати інформацію про клієнта, наприклад ідентифікатори і привілеї захисту, але не може виконуватися у контексті безпеки клієнта. Це корисно, коли серверу необхідно ідентифікувати клієнта, але не потрібно відігравати його роль.

- *SECURITY_IMPERSONATION* – рівень персоналізації. Сервер може одержати інформацію про клієнта, а також може виконуватися у контексті безпеки клієнта на локальному комп'ютері сервера.

- *SECURITY_DELEGATION* – рівень делегування. Сервер може одержати інформацію про клієнта, а також виконуватися у контексті безпеки клієнта на локальному комп'ютері сервера і на віддалених комп'ютерах.

- *SECURITY_CONTEXT_TRACKING* – задає динамічний режим спостерігання захисту. Якщо цей прапорець не зазначений, то режим статичний.

- *SECURITY_EFFECTIVE_ONLY* – вказує, що тільки встановлені аспекти контексту безпеки клієнта доступні серверу. Якщо цей прапорець не призначений, серверу доступні будь-які аспекти контексту безпеки клієнта.

hTemplateFile – не застосовується під час роботи з іменованими каналами і має дорівнювати *NULL*. Після успішного виконання функції *CreateFile* клієнт може відправляти і приймати дані за допомогою функцій *ReadFile* і *WriteFile.* Завершивши передавання, клієнт закриває з'єднання функцією *CloseHandle*.

У лістингу 5.2 наведено код клієнта іменованого каналу. За успішного з'єднання клієнт відправляє серверу повідомлення «*This is a test*».

Лістинг 5.2 – Клієнт іменованого каналу

```
// Client.cpp
#include <windows.h>
#include <iostream>
#define PIPE_NAME L"\\\\.\\Pipe\\Jim"
using namespace std;
void main(void)
 {
```

```
HANDLE PipeHandle;
DWORD BytesWritten;
if (WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_FOREVER) == 0)
{
    cout << "WaitNamedPipe failed with error: " << GetLastError() << endl;
    return;
}
// Створення описувача файла іменованого каналу
if ((PipeHandle = CreateFile(PIPE_NAME,GENERIC_READ | GENERIC_WRITE,
0,(LPSECURITY_ATTRIBUTES) NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL,
(HANDLE) NULL))== INVALID_HANDLE_VALUE)
{
    cout << "CreateFile failed with error: " << GetLastError() << endl;
    return;
}
if (WriteFile(PipeHandle, "This is a test", 14, &BytesWritten, NULL)== 0)
{
    cout << "WriteFile failed with error: " << GetLastError() << endl;
    CloseHandle(PipeHandle);
    return;
}
cout <<"Wrote " << BytesWritten  << " bytes\n";
CloseHandle(PipeHandle);
}
```

5.6 Інші API-виклики

Функції *CallNamedPipe* і *TransactNamedPipe* можуть спростити використання іменованих каналів, виконуючи операції читання і записування в одному виклику.

Функція ***CallNamedPipe*** дозволяє клієнту підключитися до іменованого каналу, що працює у режимі повідомлень (і зачекати, доки не звільниться екземпляр каналу), записати і зчитати дані, а потім закрити канал. Фактично, вона є повноцінним клієнтом іменованого каналу:

```
BOOL CallNamedPipe(
LPCTSTR lpNamedPipeName,
LPVOID lpInBuffer,
DWORD nInBufferSize,
```

LPVOID lpOutBuffer,

DWORD nOutBufferSize,

LPDWORD lpBytesOfRead,

DWORD nTimeOut);

Параметр *lpNamedPipeName* – назва іменованого каналу у форматі *UNC*.

lpInBuffer і *nInBufferSize* – адреса і розмір буфера для отримання даних.

lpOutBuffer і *nOutBufferSize* – адреса і розмір буфера для відправлення даних.

lpBytesOfRead – кількість байтів, зчитаних з каналу.

nTimeOut – час очікування звільнення каналу у мілісекундах.

Функція ***TransactNamedPipe*** використовується як у клієнтських, так і у серверних програмах. Вона поєднує операції читання і записування в одному *API*-виклику. Це оптимізує мережевий трафік за рахунок скорочення числа транзакцій, виконаних через перенаправлювач *MSNP*. Функція *TransactNamedPipe* визначена так:

BOOL TransactNamedPipe(

HANDLE hNamedPipe,

LPVOID lpInBuffer,

DWORD nInBufferSize,

LPVOID lpOutBuffer,

DWORD nOutBufferSize,

LPDWORD lpBytesRead,

LPOVERLAPPED lpOverlapped);

hNamedPipe – описувач іменованого каналу, повернутий функцією *CreateNamedPipe* чи *CreateFile*.

lpInBuffer і *nInBufferSize* – адреса і розмір буфера для отримання даних.

lpOutBuffer і *nOutBufferSize* – адреса і розмір буфера для відправлення даних.

lpBytesRead – кількість байтів, зчитаних з каналу.

lpOverlapped – дозволяє функції *TransactNamedPipe* виконуватися асинхронно шляхом перекритого введення–виведення.

Група функцій: *GetNamedPipeHandleState*, *SetNamedPipeHandleState, GetNamedPipeInfo* і *PeekNamedPipe* забезпечує більш гнучку взаємодію сервера та клієнта під час виконання. Наприклад, за допомогою цих функцій можна змінити режим роботи іменованого каналу з побайтового на режим повідомлень.

Функція ***GetNamedPipeHandleState*** повертає у параметри інформацію про канал, зокрема режим роботи, кількість екземплярів каналу та інформацію про стан буферів. Функція важлива, оскільки така інформація може змінюватися у процесі роботи іменованого каналу. Функція визначена так

BOOL GetNamedPipeHandleState(
HANDLE hNamedPipe,
LPDWORD lpState,
LPDWORD lpCurInstances,
LPDWORD lpMaxCollectionCount,
LPDWORD lpCollectDataTimeout,
LPTSTR lpUserName,
DWORD nMaxUserNameSize);

hNamedPipe – задає описувач іменованого каналу, повернутий функцією *CreateNamedPipe* чи *CreateFile*.

lpState – отримує поточний режим роботи каналу і набуває значення *PIPE_NOWAIT* чи *PIPE_READMODE_MESSAGE*.

lpCurInstances – отримує поточне число екземплярів каналу.

lpMaxCollectionCount – отримує максимальну кількість байтів, що будуть накопичені на комп'ютері клієнта перед передаванням на сервер.

lpCollectDataTimeout – отримує максимальний час очікування передавання у мілісекундах.

lpUserName і *nMaxUserNameSize* – визначають буфер, що містить ім'я користувача клієнтської програми, яке закінчується нулем.

Функція ***SetNamedPipeHandleState*** дозволяє змінити характеристики каналу, повернуті функцією *GetNamedPipeHandleState*:

BOOL SetNamedPipeHandleState(
HANDLE hNamedPipe,
LPDWORD lpMode,
LPDWORD lpMaxCollectionCount,
LPDWORD lpCollectDataTimeout);

hNamedPipe – описувач іменованого каналу, повернутий функцією *CreateNamedPipe* чи *CreateFile*.

lpMode – задає режим роботи іменованого каналу.

lpMaxCollectionCount – містить максимальну кількість байтів, що будуть накопичені на комп'ютері клієнта перед передаванням на сервер.

lpCollectDataTimeout – визначає максимальний час у мілісекундах, що може пройти до того, як віддалений клієнт передасть інформацію з мережі.

Функція ***GetNamedPipeInfo*** повертає розмір буферів і максимальну кількість екземплярів каналу:

BOOL GetNamedPipeInfo(
HANDLE hNamedPipe,
LPDWORD lpFlags,
LPDWORD lpOutBufferSize,
LPDWORD lpInBufferSize,
LPDWORD lpMaxInstances);

hNamedPipe – описувач іменованого каналу, повернутий функцією *CreateNamedPipe* чи *CreateFile*.

lpFlags – вказує вигляд і режим роботи іменованого каналу і визначає, є він сервером чи клієнтом.

lpOutBufferSize і *lpInBufferSize* – містять розмір вихідного і вхідного внутрішніх буферів.

lpMaxInstances – максимальна кількість екземплярів каналу.

Остання функція – *PeekNamedPipe* – дозволяє переглянути дані, що знаходяться у каналі, не стираючи їх із внутрішнього буфера. Наприклад, перевірити наявність вхідних даних і уникнути блокування функції *ReadFile*. Крім того, ця функція корисна, якщо необхідно перевірити дані перед одержанням: програма може скоригувати розмір свого буфера залежно від розміру вхідного повідомлення. Функція ***PeekNamedPipe*** визначена так :

BOOL PeekNamedPipe (
HANDLE hNamedPipe,
LPVOID lpBuffer,
DWORD nBufferSize,
LPDWORD lpBytesRead,
LPDWORD lpTotalBytesAvial,
LPDWORD lpBytesLeftThisMessege);

hNamedPipe – описувач іменованого каналу, повернутий функцією *CreateNamedPipe* чи *CreateFile*.

lpBuffer і *nBufferSize* – визначають адресу і розмір приймального буфера.

1pBytesRead – містить кількість байтів, зчитаних з каналу у буфер *lpBuffer*.

lpTotalBytesAvail – загальна кількість байтів, що можуть бути зчитані з каналу.

lpBytesLeftThisMessege – кількість байтів, що залишилися, у повідомленні, якщо канал працює у режимі повідомлень. Якщо повідомлення не міститься у буфері *lpBuffer*, цей параметр містить кількість байтів, що залишилася. Якщо іменований канал працює у побайтовому режимі, параметр завжди містить нуль.

5.7 Контрольні питання

1. Що таке іменовані канали?
2. Які правила іменування каналів?
3. У яких режимах можуть працювати іменовані канали?
4. Які бібліотеки і файли заголовків потрібно під'єднати для роботи з іменованими каналами?
5. Як можна отримати інформацію про коди помилок під час роботи іменованих каналів?
6. Опишіть інтерфейс функції *CreateNamedPipe*.
7. Які константи встановлюють напрямок передавання інформації під час створення іменованого каналу? Якому параметру присвоюються значення цих констант?
8. Які константи встановлюють режим читання і записування в іменований канал?
9. Якою змінною вказується, скільки іменованих каналів може одночасно створити сервер? Яких значень може набувати ця змінна?
10. Яка функція дозволяє забезпечити роботу сервера іменованого каналу у контексті безпеки клієнта? У який спосіб викликається ця функція?
11. Напишіть програму, що реалізує сервер іменованих каналів.
12. Напишіть програму клієнта іменованих каналів.
13. Які функції дозволяють отримувати та встановлювати параметри іменованого каналу? Опишіть інтерфейси цих функцій.
14. Яка функція дозволяє отримати кількість створених іменованих каналів та розміри їхніх буферів? Опишіть її інтерфейс.
15. Яка функція є повноцінним клієнтом іменованого каналу? Опишіть її інтерфейс.

16. Яка функція використовується для скорочення трафіка у серверах і клієнтах іменованих каналів? Опишіть її інтерфейс.

17. За допомогою якої функції можна переглянути дані в іменованому каналі, не витираючи буфер? Опишіть інтерфейс цієї функції.

6 ІНТЕРФЕЙС МЕРЕЖЕВОГО ПРОГРАМУВАННЯ WINSOCK

Winsock – це бібліотека, що являє собою інтерфейс мережевого програмування, незалежного від транспортного протоколу. На сучасних платформах *Windows* використовується версія 2.2. Для використання бібліотеки у програмі потрібно підключити файл *winsock2.h*, а в меню "*Project-...Propeties-ConfigurationPropeties-Linker-Input-AdditionalDependecies*" додати запис *ws2_32.lib*, або записати команду препроцесора *#pragma comment*(*lib*,"*ws2_32.lib*")

6.1 Ініціалізація Winsock

Перед викликом будь-якої функції *Winsock* необхідно завантажити правильну версію бібліотеки *Winsock* за допомогою функції ***WSAStartup***:

```
int WSAStartup(
WORD VersionRequested,
LPWSADATA lpWSAData);
```

Параметри функції

VersionRequested – версія бібліотеки *Winsock*, яку необхідно завантажити. Для завантаження версії *Winsock* 2.2 потрібно вказати значення 0x0202 або макрос *MAKEWORD(2,2)*. Верхній байт визначає додатковий номер версії, нижній – основний.

lpWSAData – структура *WSADATA*, яка повертається після завершення виклику. Вона містить інформацію про версію *Winsock*, завантажену функцією *WSAStartup*.

Структура ***WSADATA*** описується так:

```
typedef struct WSAData
{
    WORD   wVersion;
    WORD   wHighVersion;
    char      szDescription[WSADESCRIPTION_LEN +1];
    char      szSystemStatus[WSASYS_STATUS_LEN+1];
    unsigned short    iMaxSockets;
    unsigned short    iMaxUdpDg;
    char *    lpVendorInfo;
} WSADATA, * LPWSADATA;
```

Єдина корисна інформація, що повертається у структурі *WSADATA*, – поля *wVersion* і *wHighVersion*.

Опис полів структури *WSADATA*:

wVersion – версія *Winsock*, що припускає використання виклику;

wHighVersion – вища версія *Winsock*, підтримувана завантаженою бібліотекою (як правило, те саме значення, що і *wVersion*)

szDescription – текстовий опис завантаженої бібліотеки;

szSystemStatus – текстовий рядок з відповідною інформацією про стан чи конфігурації;

iMaxSockets – максимальна кількість сокетів (пропустити це поле для *Winsock* 2 і більш пізніх версій);

iMaxUdpDg – максимальний розмір дайтаграми *UDP*;

lpVendorInfo – інформація про виробника (пропустити це поле для *Winsock* 2 і більш пізніх версій).

Після завершення роботи з бібліотекою *Winsock* потрібно викликати функцію ***WSACleanup*** для вивантаження бібліотеки і звільнення ресурсів. Інтерфейс функції:

```
int WSACleanup (void);
```

Для кожного виклику *WSAStartup* необхідно відповідно викликати *WSACleanup*.

6.2 Інформація про протокол

Winsock 2 дозволяє довідатися, які протоколи і з якими характеристиками встановлено на робочій станції за допомогою функції ***WSAEnumProtocols***:

```
int WSAEnumProtocols
( LPINT lpIProtocols,
LPWSAPROTOCOL_INFO lpProtocolBuffer,
LPDWORD lpdwBufferLength);
```

lpIProtocols – кількість проттоколів;

lpProtocolBuffer – масив протоколів;

lpdwBufferLength – довжина масиву.

Ця функція замінила функцію *EnumProtocols* з *Winsock* 1.1. Єдина відмінність: *WSAEnumProtocols* повертає масив структур *WSAPROTOCOL_INFO*, а *EnumProtocols* – масив структур *PROTOCOL_INFO*, що містить менше полів, ніж структура *WSAPROTOCOL_INFO* (хоча інформація та сама). Структура

WSAPROTOCOL_INFO визначена так:

```
typedef struct _WSAPROTOCOL_INFOW
{
    DWORD dwServiceFlags1;
    DWORD dwServiceFlags2;
    DWORD dwServiceFags3;
    DWORD dwServiceFlags4;
    DWORD dwProviderFlags;
    GUID ProviderId;
    DWORD dwCatalogEntryId;
    WSAPROTOCOLCHAIN    ProtocolChain;
    int iVersion;
    int iAddressFamily;
    int iMaxSockAddr;
    int iMinSockAddr;
    int iSocketType;
    int iProtocol;
    int iProtocolMaxOffset;
    int iNetworkByteOrder;
    int iSecurityScheme;
    DWORD dwMessageSize;
    DWORD dwProviderReserved;
    WCHAR szProtocol[WSAPROTOCOL_LEN + 1];
} WSAPROTOCOL_INFOW, * LPWSAPROTOCOL_INFOW;
```

Особливо часто у структурі *WSAPROTOCOL_INFO* використовується поле *dwServiceFlags1* – бітова маска різних атрибутів протоколу. У наведеному далі списку перераховано бітові прапорці цього поля і дії, що ініціюються заданими прапорцями.

XP1_CONNECTIONLESS – протокол передає дані без встановлення з'єднання. Якщо прапорець не задано – із встановленням з'єднання.

XPI_GUARANTEED_DELIVERY – протокол гарантує доставку даних одержувачу.

XP1_GUARANTEED_ORDER – протокол гарантує доставку даних у порядку їхнього відправлення без дублювання, хоча саму доставку не гарантує.

XP1_MESSAGE_ORIENTED – протокол обробляє межі повідомлень.

XP1_PSEUDO_STREAM – протокол передає повідомлення, але межі повідомлень ігноруються приймачем.

XP1_GRACEFUL_CLOSE – протокол підтримує двофазне завершення сеансу (кожна сторона повідомляється про намір іншої завершити сеанс зв'язку). Якщо цей прапорець не задано, сеанс розривається без попередження.

XP1_EXPEDITED_DATA – протокол підтримує обмін терміновими (out-of-band) даними.

XP1_CONNECT_DATA – протокол підтримує передавання даних із запитом з'єднання.

XP1_DISCONNECT_DATA – протокол підтримує передавання даних із запитом роз'єднання.

XP1_SUPPORT_BROADCAST – протокол підтримує механізм широкомовлення.

XP1_SUPPORT_MULTIPOINT – протокол підтримує механізм багатоадресного передавання даних.

XP1_MULTIPOINT_CONTROL_PLANE – площина керування (control plane) маршрутизується, якщо прапорець не задано – цього не відбувається.

XP1_MULTIPOINT_DATA_PLANE – площина даних маршрутизується, якщо прапорець не задано – цього не відбувається.

XP1_QOS_SUPPORTED – протокол підтримує запити *QoS*.

XP1_UNI_SEND – протокол односпрямований і забезпечує лише відправлення даних.

XP1_UNI_RECV – протокол односпрямований і забезпечує лише приймання даних.

XP1_IFS_HANDLES – дескриптори сокета, повернуті відновлювачем, є описувачами файлової системи *IFS* і можуть бути використані у таких *API*-функціях, як *ReadFile* і *WriteFile*.

XPl_PARTIAL_MESSAGE – прапорець *MSG_PARTIAL*, підтримується функціями *WSASend* і *WSASendTo*.

Для перевірки наявності певної властивості потрібно виконати логічне АБО відповідного прапорця з полем *dwServiceFlags*. Якщо результат операції не нульовий, протокол має цю властивість, інакше – ні.

Поле *iProtocol* визначає, до якого протоколу належить цей запис.

Поле *iSocketType* важливе, якщо протокол здатен працювати у різних режимах, наприклад, із установленням потокового чи дайтаграмного з'єднання.

Поле *iAddressFamily* дозволяє з'ясувати коректну структуру адресації, застосовувану цим протоколом. Ці три поля дуже важливі під час створення сокета для конкретного протоколу.

Найпростіше у *WSAEnumProtocols* задати *lpProtocolBuffer=NULL* і *lpdwBufferLength*=0. Викликавши функцію з правильним розміром буфера, можна отримати кілька структур *WSAPROTOCOL_INFO*.

6.3 Сокети Windows

Сокет – це описувач постачальника транспорту. У *Windows* сокет подано типом *SOCKET*. Сокет створюється однією з двох функцій:

SOCKET socket (int af, int type, int protocol);
SOCKET WSASocket (
int af,
int type,
int protocol,
LPWSAPROTOCOL_INFO lpProtocolInfo,
GROUP g,
DWORD dwFlags);

Параметри функції такі:

af – визначає сім'ю адрес протоколу.

type – тип сокета для цього протоколу.

protocol – вказує конкретний транспорт.

У табл. 6.1 вказано можливі значення параметрів *af*, *type* та *protocol*.

Таблиця 6.1 – Основні параметри сокетів

Протокол		Сокет		
Мережевий	Транспортний	Сім'я адрес (*af*)	Тип сокета (*type*)	Тип протоколу (*protocol*)
IP	*TCP*	*AF_INET*	*SOCK_STREAM*	*IPPROTO_TCP*
	UDP		*SOCK_DGRAM*	*IPPROTO_UDP*
	Прості сокети		*SOCK_RAW*	*IPPROTO_RAW* *IPPROTO_IGMP*

Найважливішим параметром є *af*, який обмежує значення інших параметрів. Якщо задано параметри *af* та *type*, то у *protocol* можна задати нуль. Це можливо лише для протоколів, у яких функція *WSAEnumProtocols* повертає структуру *WSAPROTOCOL_INFO*, що у полі *dwProviderFlags* має значення *PFL_MATCHES_PROTOCOL_ZERO*.

lpProtocolInfo – вказівник на структуру *WSAPROTOCOL_INFO*, яка повертається функцією *WSAEnumProtocols*.

g – дорівнює нулю, оскільки *Windows* не підтримує групи сокетів.

dwFlags – може задаватись одним або декількома прапоцями: *WSA_MULTIPOINT_C_LEAF, WSA_FLAG_OVERLAPPED, WSA_MULTIPOINT_D_LEAF, WSA_MULTIPOINT_D_ROOT, WSA_MULTIPOINT_C_ROOT*.

WSA_FLAG_OVERLAPPED встановлює перекрите введення–виведення і рекомендується задавади завжди у функції *WSASocket*. Решта прапорців відносяться до багатоадресного передавання.

Створення *IP*-сокета дозволяє програмам здійснювати під'єднання через *TCP*, *UDP* і протоколи *IP*. Для створення *IP*-сокета з використанням протоколу *TCP* викликаються функції *socket* чи *WSASocket* із сім'єю адрес *AF_NET*, типом сокета *SOCK_STREAM* та значенням «нуль» поля протоколу, наприклад:

```
SOCKET s;
s=socket(AF_INET, SOCK_STREAM, 0);
s=WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0, WSA_FLAG_OVERLAPPED);
```

Для створення *IP*-сокета з використанням протоколу *UDP* вказують тип сокета *SOCK_DGRAM*. Якщо потрібно створити сокет для зв'язку безпосередньо через *IP*, то вказують тип *SOCK_RAW*.

6.4 Winsock і модель OSI

До *Winsock* відносять три верхні рівні протоколів *OSI*: прикладний, представницький і сеансовий. Тобто, програма *Winsock* керує всіма аспектами сеансу зв'язку та може форматувати дані.

6.5 Використання адрес і перетворення імен

6.5.1 Протокол IP

Internet Protocol (*IP*) підтримується більшістю ОС як у локальних (*local area networks, LAN*), так і у глобальних мережах (*wide area networks, WAN*). *IP* не потребує встановлення з'єднання і не гарантує доставку даних. Для передавання даних використовуються два протоколи більш високого рівня *TCP* і *UDP*, тому говорять про використання *TCP*/*IP* чи *UDP*/*IP*. У *Winsock* для *IP*-з'єднань передбачена сім'я адрес *AF_INET*, визначена у файлах *Winsock.h* і *Winsock2.h*.

6.5.2 Протокол TCP

Transmission Control Protocol (*TCP*) реалізує обмін із встановленням віртуального з'єднання локального комп'ютера з віддаленим та забезпечує

надійне передавання даних в обох напрямках. У випадку виявлення помилок у переданих даних приймальна сторона надсилає запит на їх повторне передавання.

6.5.3 Протокол UDP

Зв'язок без встановлення з'єднання виконується за допомогою *User Datagram Protocol* (*UDP*). *UDP* може виконувати передавання даних безлічі адресатів і приймання даних від безлічі джерел, але не гарантує надійності. Дані, що відправляються клієнтом, передаються негайно, незалежно від того, чи готовий сервер до приймання. Під час одержання даних від клієнта сервер не підтверджує їх приймання. Дані передаються у вигляді дайтаграм.

6.5.4 Адресація

Під час використання *IP* комп'ютерам призначаються *IP*-адреси, що складаються з 32 бітів та офіційно названі *IP*-адресами версії 4 (*IPv4*). Для взаємодії із сервером через *TCP* чи *UDP* клієнт має вказати *IP*-адресу сервера і номер порту служби. Щоб прослуховувати вхідні запити клієнта, сервер також має вказати *IP*-адресу і номер порту. У *Winsock IP*-адреса і порт служби задають у структурі ***sockaddr_in***.

```
struct sockaddr_in
{
    short sin_family;
    u_short sin_port;
    in_addr sin_addr;
    char sin_zero[8];
};
```

Поля структури:

sin_family – має дорівнювати *AF_INET*. Це означає, що використовується сім'я адрес *IP*;

sin_port – задає комунікаційний порт;

sin_addr – зберігає *IP*-адреси у 4-байтовому вигляді і є типом *unsigned long int*;

sin_zero – відіграє роль простого заповнювача, щоб структура *sockaddr_in* за розміром дорівнювала структурі *sockaddr*.

IP-адреси звичайно задають у точковій нотації *a.b.c.d* у вигляді рядка символів. Корисна допоміжна функція ***inet_addr*** перетворить *IP*-адресу з

точкової нотації у 32-бітове довге ціле без знака:

*unsigned long inet_addr(const char *cp);*

Параметр *cp* є рядком, що закінчується нульовим символом, тут задається *IP*-адреса у крапковій нотації. Ця функція як результат повертає *IP*-адресу, подану 32-бітовим числом з мережевим порядком проходження байтів (*network-byte order*).

6.5.5 Спеціальні адреси

INADDR_ANY – дозволяє серверній програмі слухати клієнта через будь-який мережевий інтерфейс на своєму комп'ютері. Звичайно програми сервера використовують цю адресу, щоб прив'язати сокет до локального інтерфейсу для прослуховування з'єднань.

INADDR_BROADCAST – дозволяє широкомовно розсилати *UDP*-дайтаграми *IP*-мережею. Для її використання необхідно у програмі задати параметр сокета *SO_BROADCAST*.

INADDR_LOOPBACK – адреса локального комп'ютера.

6.5.6 Номери портів

У разі використання *TCP* і *UDP* програма вказує, через який порт зв'язатися. Існують стандартні номери портів, зарезервовані для служб сервера, що підтримують протоколи більш високого рівня, ніж *TCP*. Наприклад, порт 21 зарезервований для *FTP*, 80 – для *HTTP*. Для визначення номерів портів стандартних служб використовуються функції *getservbyname* чи *WSAAsyncGetServByName*. Ці функції просто читають інформацію з файла з ім'ям *services*. Файл *services* розташований у папці *%WINDOWS%\System32\Drivers\Etc*.

Номери портів поділяють на три категорії:

- 0–1023 – стандартні, контролюються *IANA* і зарезервовані для стандартних служб;
- 1024–49151 – зареєстровані *IANA* і можуть використовуватися процесами і програмами;
- 49152–65535 – динамічні (приватні).

Якщо під час використання функції *bind* програма спробує вибрати порт, уже зайнятий іншою програмою, то система поверне помилку *WSAEADDRINUSE*.

6.5.7 Порядок байтів

Різні процесори залежно від своєї архітектури зберігають і обробляють

числа в одному з двох порядків байтів: *big-endian* чи *little-endian*. Наприклад, процесори *Intel x86* зберігають і обробляють багатобайтові числа у порядку від менш значущого до більш значущого байта (*little-endian*). Порядок, у якому зберігаються числа у пам'яті комп'ютера, називається системним (*host-byte-order*). *IP*-адреса та номер порту передаються мережею у порядку від старшого байта до молодшого (*big-endian*), що називається мережевим порядком (*network-byte-order*). Є цілий ряд функцій для перетворення чисел із системного порядку у мережевий і навпаки. Чотири таких *API*-функції перетворять числа із системного порядку у мережевий:

```
u_long htonl(u_long hostlong);
int WSAHtonl ( SOCKET s, u_longl hostlong, u_long * lpnetlong);
u_short htons(u_short hostshort);
int WSAHtons(SOCKET s, u_short hostshort, u_short * lpnetshort );
```

Параметри *hostlong* функцій *htonl* і *WSAHtonl* – чотирибайтові числа типу *long* із системним порядком. Функція *htonl* повертає число типу *long* з мережевим порядком, а *WSAHtonl* – число типу *long* з мережевим порядком через параметр *lpnetlong*.

Параметр *hostshort* функцій *htons* і *WSAHtons* є двобайтовим числом типу *short* із системним порядком. Функція *htons* повертає значення типу *short* з мережевим порядком. Функція *WSAHtons* повертає число типу *short* через параметр *lpnetshort*.

Перетворення порядку байтів з мережевого у системний виконують такі чотири функції:

```
u_long ntohl(u_long netlong);
int WSANtohl (SOCKET s, u_long netlong, u_long * lphostlong);
u_short ntohs(u_short netshort);
int WSANtohs(SOCKET s, u_short netshort, u_short * lphostshort );
```

Продемонструємо, як задати структуру *sockaddr_in* за допомогою вже описаних функцій *inet_addr* і *htons*:

```
sockaddr_in InternetAddr;
int nPort = 5150;
InternetAddr.sin_family = AF_INET;
InternetAddr.sin_addr.s_addr = inet_addr("136.149.3.29");
InternetAddr.sin_port = htons(nPort);
```

6.5.8 Перетворення імен

Для під'єднання до вузла через *IP Winsock*-програма має знати *IP*-адресу цього вузла. Користувачі більш охоче звертаються до комп'ютерів за допомогою імен вузлів, що запам'ятовуються легко. У *Winsock* для перетворення імені в *IP*-адресу передбачено дві функції *gethostbyname* і *WSAAsyncGetHostByName*, які повертають структуру ***hostent***.

```
struct hostent
{
    char * h_name;
    char ** h_aliases;
    short h_addrtype;
    short h_length;
    char ** h_addr_list;
};
```

Поля структури:

h_name – офіційне ім'я вузла;

h_aliases – масив, що завершується нулем (*null-terminated array*) додаткових імен вузла;

h_addrtype – сім'я адрес;

h_length – визначає довжину у байтах кожної адреси з поля *h_addr_list*;

h_addr_list – масив, що завершується нулем і містить *IP*-адреси вузла (вузол може мати кілька *IP*-адрес). Кожна адреса у цьому масиві подана у мережевому порядку.

API-функція ***gethostbyname*** визначена так:

```
hostent * gethostbyname (const char * name );
```

name – дружнє ім'я вузла. За успішного виконання функції повертається вказівник на структуру *hostent*, що зберігається у системній пам'яті. У процесі роботи програми вміст структури може змінюватись системою. Оскільки структура *hostent* використовується системою, то програма не має її знищувати.

WSAAsyncGetHostByName – асинхронна версія функції *gethostbyname*. Оповіщає програму про завершення свого виконання за допомогою повідомлень *Windows*:

```
HANDLE WSAAsyncGetHostByName(
HWND hWnd,
unsigned int wMsg,
```

*const char * name,*
*char * buf,*
int buflen);

Параметри функції:

hWnd – дескриптор вікна, що одержить повідомлення після завершення виконання асинхронного запиту;

wMsg – *Windows*-повідомлення, що буде повернуто після завершення виконання асинхронного запиту;

name – дружнє ім'я вузла;

buf – вказівник на область даних, куди поміщається hostent. Цей буфер має бути більшим за структуру *hostent* і мати розмір, визначений у *MAXGETHOSTSTRUCT*.

Існують також дві функції пошуку інформації про вузол: *gethostbyaddr* і *WSAAsyncGetHostByAddr*. Вони дозволяють за *IP*-адресою вузла знайти його зрозуміле користувачу ім'я.

Функція ***gethostbyaddr*** визначена так:

*hostent * gethostbyaddr(const char * addr, int len, int type);*

Параметр *addr* – вказівник на *IP*-адресу у мережевому порядку.

Параметр *len* задає довжину параметра *addr* у байтах.

Параметр *type* – потрібно, щоб мав значення *AF_NET* (*IP*-адреса).

WSAAsyncGetHostByAddr – асинхронна версія функції *gethostbyaddr*:

HANDLE WSAAsyncGetHostByAddr(
HWND hWnd,
unsigned int wMsg,
*const char *addr,*
int len,
int type,
*char *buf,*
int buflen);

Параметри *hWnd*, *wMsg*, *buf*, *buflen* такі самі, як у функції *WSAAsyncGetHostByName*.

Крім того:

addr – адреса вузла у мережевому порядку;

len – розмір структури *addr*;

type – дорівнює *AF_INET*;

Функція ***getservbyname***. Інтерфейс функції:

*servent * getservbyname(const char * name, const char * proto);*

Параметри функції:

name – ім'я шуканої служби. Наприклад, в процесі визначення порту, що використовується службою *FTP*, параметру name потрібно передати вказівник на рядок «*ftp»*;

proto – посилається на рядок, що вказує протокол, під яким зареєстрована служба з параметра name.

Функція повертає структуру ***servent***:

```
struct servent
{
    char *s_name;
    char **s_aliases;
    short s_port;
    char *s_proto;
};
```

Поля структури *s_name* і *s_aliases* такі ж, як у структури *hostent*. Крім того:

s_port – комунікаційний порт;

s_proto – ім'я протоколу.

Функція ***WSAAsyncGetServByName*** – асинхронна версія *getservbyname*.

```
HANDLE WSAAsyncGetServByName(
HWND wnd,
u_int msg,
char *name,
char *proto,
char *buf,
nt buflen);
```

6.6 API-функції сервера

Для встановлення з'єднання сервер має прослуховувати запити на з'єднання від клієнтів на стандартному імені. У *TCP/IP* таким ім'ям є *IP*-адреса локального інтерфейсу і номер порту, які вказуються у структурі *sockaddr_in*.

Перший крок встановлення з'єднання – прив'язка сокета до

стандартного імені функцією *bind*. Другий – переведення сокета у режим прослуховування функцією *listen*. І нарешті, сервер має прийняти запит на з'єднання функцією accept чи *WSAAccept*.

Основні виклики функцій, що мають зробити клієнт і сервер для встановлення каналу зв'язку між комп'ютерами у мережі, зображено на рис. 6.1.

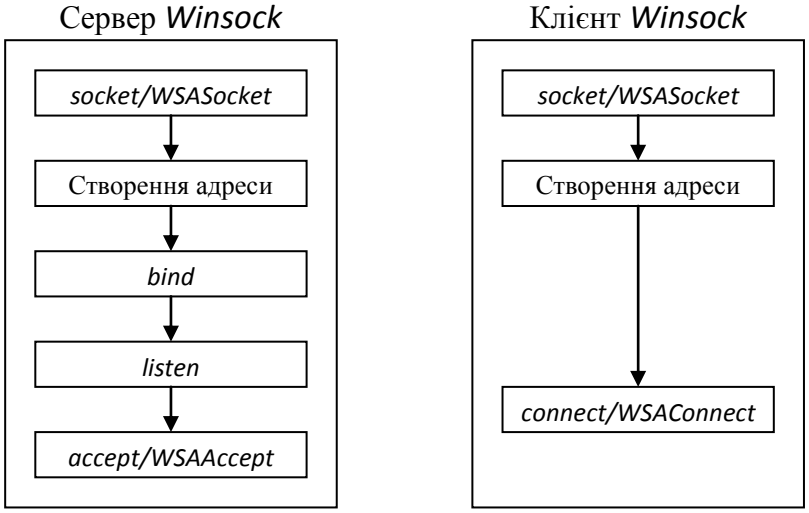


Рисунок 6.1 – Встановлення з'єднання

Розглянемо ці функції.

6.6.1 Функції зв'язування і прослуховування сокета

Функція ***bind***. Зв'язує сокет із заданою адресою. Інтерфейс функції:

*int bind (SOCKET s, const sockaddr *name, int namelen);*

Параметри функції:

s – задає сокет, на якому будуть очікуватись запити клієнтів на з'єднання;

name – вказівник на структуру, у якій знаходиться локальна адреса. Під час виклику *bind* потрібно привести цю структуру до типу *sockaddr**;

namelen – розмір передаваної структури адреси.

Далі наведено приклад, що ілюструє прив'язку сокета у разі *TCP*-з'єднання:

SOCKET s;

```
sockaddr_in addr;
int port = 5150;
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.S_un.S_addr = htonl(INADDR_ANY);
bind(s, (sockaddr *)&addr, sizeof(addr));
```

Під час виникнення помилки функція *bind* повертає значення *SOCKET_ERROR*. Найпоширеніша помилка під час виклику *bind* – *WSAEADDRINUSE*. У випадку використання *TCP/IP* це означає, що з локальним *IP*-інтерфейсом і номером порту вже зв'язаний інший процес чи вони знаходяться у стані *TIME_WAIT*. У випадку повторного виклику *bind* для вже зв'язаного сокета повертається помилка *WSAEFAULT*.

Функція ***listen***

Переводить сокет у режим прослуховування. Інтерфейс функції:

```
int listen(SOCKET s, int backlog);
```

Параметри функції:

s – зв'язаний сокет;

backlog – максимальна довжина черги з'єднань, що очікують оброблення. Нехай значення цього параметра дорівнює 2, тоді у разі одночасного приймання трьох клієнтських запитів перші два з'єднання будуть поміщені у чергу очікування, і програма зможе їх обробити. Третій запит поверне помилку *WSAECONNREFUSED*. Після того як сервер прийме з'єднання, запит видаляється з черги, а інший займає його місце. Значення *backlog* залежить від постачальника протоколу. Неприпустиме *backlog* значення замінюється найближчим дозволеним.

Помилки, пов'язані з *listen*, досить прості. Найчастіша з них – *WSAEINVAL*. Звичайно це означає, що перед *listen* не була викликана функція *bind*. Іноді під час виклику *listen* виникає помилка *WSAEADDRINUSE*, але частіше вона виникає під час виклику *bind*.

6.6.2 Функції приєднання сокета

Приєднання нового сокета до прослуховувального сокета виконується за допомогою функцій *accept* і *WSAAccept*.

Функція ***accept***:

```
SOCKET accept( SOCKET s, sockaddr *addr, int *addrlen);
```

Параметри функції:

s – зв'язаний сокет у стані прослуховування;

addr – вказівник на адресу існуючої структури *sockaddr_in*;

addrlen – вказівник на довжину структури *sockaddr_in*.

Виклик accept обслуговує перший запит, що знаходиться у черзі на з'єднання. Після його завершення структура *addr* буде містити інформацію про *IP*-адресу клієнта, що відправив запит, а параметр *addrlen* – розмір структури. Крім того, *accept* повертає новий дескриптор сокета, що відповідає прийнятому клієнтському з'єднанню. Для всіх таких операцій з цим клієнтом має застосовуватися новий сокет. Прослуховувальний сокет використовується для приймання інших клієнтських з'єднань і продовжує знаходитися у режимі прослуховування.

Функція ***WSAAccept*** здатна встановлювати з'єднання залежно від результату обчислення умови:

```
SOCKET WSAAccept(
SOCKET s,
sockaddr * addr,
LPINT addrlen,
LPCONDITIONPROC lpfnCondition,
DWORD dwCallbackData );
```

Перші три параметри – ті самі, що і в *accept* для *Winsock* 1.

lpfnCondition – вказівник на функцію, що викликається під час запиту клієнта. Вона визначає можливість приймання з'єднання і має такий прототип:

```
int CALLBACK ConditionFunc(
LPWSABUF lpCallerId.
LPWSABUF lpCallerData,
LPQOS lpSQOS.
LPQOS lpGQOS,
LPWSABUF lpCalleeId,
LPWSABUF lpCalleeData,
GROUP * g.
DWORD dwCallbackData );
```

lpCallerId – містить адресу об'єкта, що з'єднується.

Структура *WSABUF* використовується багатьма функціями *Wipsock* 2 і визначена так:

```
typedef struct __WSABUF
{
```

```
    u_long len;
    char * buf;
} WSABUF,* LPWSABUF;
```

Залежно від її використання поле *len* визначає розмір буфера, на який посилається поле *buf*, чи кількість даних у буфері *buf*.

lpCallerId – вказує на структуру *WSABUF*, поле *buf* якої вказує на структуру *sockaddr*, що містить адресу клієнта. Щоб одержати коректний доступ до інформації, потрібно привести вказівник *buf* до відповідного типу *sockaddr*. Більшість мережевих протоколів віддаленого доступу підтримують ідентифікацію абонента на етапі запиту.

lpCallerData – вказує на структуру, що містить дані, відправлені клієнтом під час запиту з'єднання. Якщо ці дані не зазначені, *lpCallerData* дорівнює *NULL*. Більшість мережевих протоколів, таких як *TCP/IP*, не використовують дані про з'єднання. Щоб довідатися, чи підтримує протокол цю можливість, потрібно викликати функцію *WSAEnumProtocols*.

lpSQOS і *lpGQOS* – задають рівень якості обслуговування, запитуваний клієнтом. Обидва параметри посилаються на структуру, що містить інформацію про вимоги пропускної здатності для приймання і передавання. Якщо клієнт не запитує параметри якості обслуговування (*quality of service, QoS*), то вони дорівнюють *NULL*. Різниця між ними у тому, що lpSQOS використовується для одного сокета, а *lpGQOS* – для груп сокетів. Групи сокетів не реалізовані і не підтримуються *Winsock* 1 і *Winsock* 2.

lpCalleeId – вказівник на структуру *WSABUF*, поле *buf* якої вказує на структуру *sockaddr*, що містить локальну адресу, до якої під'єднаний клієнт. Ця інформація корисна, якщо сервер запущений на багатоадресній машині. Якщо сервер зв'язаний з адресою *INADDR_ANY*, запити з'єднання будуть обслуговуватися у будь-якому мережевому інтерфейсі, а параметр *lpCalleeId* буде містити адресу інтерфейсу, що прийняв з'єднання.

lpCalleeData – посилається на структуру *WSABUF*, яку сервер може використовувати для відправлення даних клієнту в процесі встановлення з'єднання. Якщо постачальник послуг підтримує цю можливість, поле *len* вказує максимальне число відправлення байтів. У такому випадку сервер копіює деяку, не більшу від цього значення, кількість байтів у блок *buf* структури *WSABUF* і обновляє поле *len*, щоб показати, скільки байтів передається. Якщо сервер не має повертати дані про з'єднання, то перед поверненням умовна функція приймання з'єднання присвоїть полю *len* значення «нуль». Якщо постачальник не підтримує передавання даних про

з'єднання, поле *len* буде дорівнює нулю. Знову ж, більшість протоколів, що підтримуються платформами *Windows*, не підтримують обмін даними під час встановлення з'єднання.

Обробивши передані у функцію *ConditionFunc* параметри, сервер має вирішити – приймати, відхиляти чи затримати запит з'єднання. Якщо з'єднання приймається, функція *ConditionFunc* поверне значення *CF_ACCEPT*, якщо відхиляється – *CF_REJECT*. Якщо за якихось причин на цей момент рішення не може бути прийнято, повертається *CF_DEFER*.

Як тільки сервер готовий обробити запит, він викликає функцію *WSAAccept*. Функція *ConditionFunc* виконується в одному процесі з *WSAAccept* і має працювати якнайшвидше. У протоколах, підтримуваних платформами *Windows*, клієнтський запит затримується, доки не буде обчислено значення функції *ConditionFunc*. У більшості випадків базовий мережевий стек до моменту виклику умовної функції вже може прийняти з'єднання. А під час повернення значення *CF_REJECT* стек просто закриває його.

У разі виникнення помилки повертається значення *INVALID_SOCKET*, найчастіше – *WSAEWOULDBLOCK*. Воно повертається, якщо сокет знаходиться у неблокувальному асинхронному режимі і якщо немає з'єднання для приймання. Якщо умовна функція поверне *CF_DEFER*, *WSAAccept* генерує помилку *WSATRY_AGAIN*, якщо *CF_REJECT* – то помилку *WSAECONNREFUSED*.

6.7 API-функції клієнта

Клієнтська частина значно простіша і для встановлення з'єднання потрібно всього три етапи: створити сокет функцією *socket* чи *WSASocket*, перетворити ім'я сервера (залежить від використовуваного протоколу) та ініціювати з'єднання функцією *connect* чи *WSAConnect*.

6.7.1 Функції встановлення з'єднання

Встановлення з'єднання здійснюється викликом *connect* чи *WSAConnect*.

Функція ***connect***:

```
int connect(
SOCKET s,
const sockaddr * name,
int namelen);
```

Параметри:
s – існуючий *TCP*-сокет для встановлення з'єднання;
name – структура, що містить адресу і порт віддаленого сокета;
namelen – довжина змінної *name*.

Функція ***WSAConnect***:
int WSAConnect(
SOCKET s,
*const sockaddr * name,*
int namelen,
LPWSABUF lpCallerData,
LPWSABUF lpCalleeData,
LPQOS lpSQOS,
LPQOS lpGQOS);
Перші три параметри такі самі, як і у функції *connect*.

lpCallerData – вказівник на буфер з даними, що відправляються клієнтом серверу разом із запитом на з'єднання;

lpCalleeData – вказівник на буфер з даними, що повертаються клієнту сервером під час встановлення з'єднання.

Обидві змінні є структурами *WSABUF*, і для *lpCallerData* поле *len* має вказувати довжину буфера *buf* для передавання даних. У випадку *lpCalleeData* поле *len* визначає розмір буфера *buf* для приймання даних.

lpSQOS і *lpGQOS* – вказівники на структури *QoS*, що визначають вимоги до пропускної здатності відправлення і приймання даних встановлюваного з'єднання. Параметр *lpSQOS* задає вимогу до пропускної здатності сокета *s*, а *lpGQOS* – до всієї групи сокетів, в яку входить *s*. Однак на цей момент групи сокетів у *Windows* не підтримуються. Нульове значення параметра *lpSQOS* означає, що програма не висуває вимог до якості обслуговування.

Якщо на віддаленому комп'ютері не виконується програма, що прослухує цей порт, функція *connect* поверне помилку *WSAECONREFUSED*. Інша помилка – *WSAETIMEDOUT* – виникає у тому випадку, коли адресат недоступний, наприклад, через відмову комунікаційного обладнання на шляху до вузла чи відсутності вказаного вузла у мережі.

6.8 Передавання даних

Для пересилання даних через сокет використовуються функції *send* та *WSASend*. Аналогічно, для приймання даних існують функцій *recv* і *WSARecv*.

Функції, що використовуються під час відправлення і приймання даних, не призначені для роботи з кодуванням *UNICODE*. Відправити рядок

символів *UNICODE* можна, використовуючи тип *UNICODE* або привівши до типу *char*. У першому випадку за вказання кількості символів, що відправляються, чи прийнятих символів значення параметра, який вказує кількість переданих або прийнятих символів, має бути у два рази більшим, тому що кожен *UNICODE*-символ займає 2 байти масиву. У другому випадку потрібно спочатку перевести рядок з *UNICODE* у *ASCII* функцією *WideCharToMultiByte*.

Усі функції приймання і відправлення даних у разі виникнення помилки повертають код *SOCKET_ERROR.* Для одержання більш докладної інформації про помилку потрібно викликати функцію *WSAGetLastError*. Найпоширеніші помилки – *WSAECONNABORTED* і *WSAECONNRESET*. Обидві помилки виникають в процесі закриття з'єднання: або після закінчення часу очікування або під час закриття з'єднання партнерським вузлом. Ще одна типова помилка – *WSAEWOULDBLOCK*, як правило, виникає у разі використання асинхронних сокетів. Ця помилка означає, що функція не може бути виконана у цей момент. Функції *send* і *WSASend* призначені для відправлення даних через сокет.

Функція ***send*** визначена так

*int send(SOCKET S, const char * buf, int len, int flags);*

Параметри:

s – описувач сокета для відправлення даних;

buf – вказівник на символьний буфер, що містить дані для відправлення;

len – містить кількість символів у буфері;

flags – встановлює режим передавання. Може набувати значень 0, *MSG_DONTROUTE*, *MSG_OOB* чи результату логічного АБО над двома останніми параметрами. За вказання прапорця *MSG_DONTROUTE* транспорти не будуть маршрутизувати пакети, що відправляються. Прапорець *MSG_OOB* вказує, що дані мають бути відправлені поза смугою (*out of band*), тобто терміново. Обробка цього запиту залишається на розсуд базового протоколу (наприклад, якщо транспорт не підтримує цей параметр, запит ігнорується).

У випадку успішного виконання функція *send* повертає кількість переданих байтів, інакше – помилку *SOCKET_ERROR*. Одна з типових помилок – *WSAECONNABORTED*, відбувається під час розриву віртуального з'єднання через помилку протоколу чи закінчення часу очікування. У

цьому випадку сокет має бути закритий, тому що він більше не може використовуватися. Помилка *WSAECONNRESET* відбувається, якщо програма на віддаленому вузлі, виконавши апаратне закриття, розриває віртуальне з'єднання чи зненацька завершується, чи відбувається перезавантаження віддаленого вузла. У такій ситуації сокет також має бути закритий. Ще одна помилка – *WSAETIMEDOUT*, найчастіше відбувається у разі обриву з'єднання через збої мережі чи відмови віддаленої системи без попередження.

Функція ***WSASend*** визначена так:

int WSASend (
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesSent,
DWORD dwFlags,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

Параметри:

s – описувач сокета для відправлення даних;

lpBuffers – вказівник на структуру *WSABUF* чи на масив цих структур;

dwBufferCount – кількість переданих структур *WSABUF*. Потрібно пам'ятати, що структура *WSABUF* містить сам символьний буфер і його довжину. Це називається комплексним введенням–виведенням (*scatter-gather I/O*). Під час використання декількох буферів для відправлення даних через сокет з'єднання масив буферів відправляється, починаючи з першої і закінчуючи останньою структурою *WSABUF*;

lpNumberOfBytesSent – вказівник на тип *DWORD*, що після виклику *WSASend* містить загальне число переданих байтів;

dwFlags – такий самий, що й у функції *send*;

1pOverlapped і *lpCompletionRoutine* – використовуються для перекритого введення–виведення (*overlapped I/O*) – однієї з моделей асинхронного введення–виведення, підтримуваних *Winsock*.

WSASend присвоює параметру *lpNumberOfBytesSent* кількість записаних байтів. За успішного виконання функція повертає нуль, інакше – *SOCKET_ERROR*. Помилки ті самі, що й у функції *send*.

Функція ***WSASendDisconnect***

Ця спеціалізована функція використовується рідко. Вона визначена так:

int WSASendDisconnect (

SOCKET s,

LPWSABUF lpOUTboundDisconnectData);

Функція *WSASendDisconnect* починає процес закриття сокета і відправляє відповідні дані. Зрозуміло, вона доступна тільки для протоколів, що підтримують поступове закриття і передавання даних під час його здійснення. Жоден з існуючих постачальників транспорту на цей момент не підтримує передавання даних про закриття з'єднання. Функція *WSASendDisconnect* діє аналогічно *shutdown* з параметром *SD_SEND*, але також відправляє дані, що містяться у параметрі *boundDisconnectData*. Після її виклику відправляти дані через сокет неможливо. У випадку невдалого завершення *WSASendDisconnect* повертає значення *SOCKET_ERROR*. Помилки, що зустрічаються під час роботи функції, аналогічні помилкам *send*.

6.8.1 Термінові дані

Якщо програмі потрібно відправити через потоковий сокет інформацію більш високого пріоритету, вона може позначити цю інформацію як термінові дані *(out-of-band, OOB*). Програма з іншої сторони з'єднання одержує й обробляє *OOB*-дані через окремий логічний канал, концептуально незалежний від потоку даних.

У *TCP* передавання *OOB*-даних реалізовано шляхом додання однобітового маркера (*URG*) і 16-бітового вказівника у заголовку сегмента *TCP*, що дозволяють виділити важливі байти в основному трафіку. На цей момент для *TCP* існують два способи виділення термінових даних. У *RFC 793*, що описує *TCP* і концепцію термінових даних, говориться, що вказівник терміновості у заголовку *TCP* є додатним зсувом байта, що розташований за байтом термінових даних. Однак у *RFC 1122* цей зсув трактується як вказівник на сам байт терміновості.

У специфікації *Winsock* під терміном *OOB* розуміють як незалежні від протоколу *OOB* дані, так і реалізацію механізму передавання термінових даних у *TCP*. Для перевірки, чи є у черзі термінові дані, потрібно викликати функцію *ioctlsocket* з параметром *SIOCATMARK*.

У *Winsock* передбачено кілька способів передавання термінових даних. Можна вставити їх у звичайний потік або, відключивши цю можливість, викликати окрему функцію, що повертає тільки термінові дані. Параметр *SO_OOBINLINE* керує поведінкою *OOB*-даних.

У ряді випадків термінові дані використовують програми *Telnet* і *Rlogin*. Утім краще уникати застосування термінових даних – вони не стандартизовані і можуть мати інші реалізації на відмінних від *Windows* платформах. Якщо потрібно час від часу передавати терміново якусь інформацію, то потрібно створити окремий керівний сокет для термінових даних, а основне з'єднання надати для звичайного передавання даних.

6.8.2 Функції recv і WSARecv

Функція ***recv*** – основний інструмент приймання даних через сокет. Вона визначена так:

```
int recv(
SOCKET s,
char * buf,
int len,
int flags );
```

Параметри:

s – визначає сокет для приймання даних;

buf – символьний буфер, призначений для отриманих даних;

len – кількість прийнятих байтів чи розмір буфера *buf*;

flags – може набувати значення *0, MSG_PEEK, MSG_OOB* чи результат логічного АБО над двома останніми параметрами. Нуль означає відсутність дій. Прапорець *MSG_PEEK* вказує, що доступні дані мають копіюватися у приймальний буфер і водночас залишатися у системному буфері. Після завершення функція повертає кількість байтів у системному буфері. Зчитувати повідомлення з використанням прапорця *MSG_PEEK* не рекомендується. Мало того, що через наявність двох системних викликів (один – для зчитування даних, і інший, без прапорця *MSG_PEEK*, – для знищення даних) знижується продуктивність. У ряді випадків цей спосіб просто ненадійний. Обсяг даних, що повертаються, може не відповідати їхній сумарній доступній кількості. До того ж, зберігаючи дані у системних буферах, система залишає все менше пам'яті для розміщення наступних вхідних даних. Як наслідок, зменшується розмір вікна *TCP* для всіх

процесів-відправників, що не дозволяє програмі досягти максимальної продуктивності. Найкраще скопіювати всі дані у власний буфер і обробляти їх там. Прапорець *MSG_OOB* вже обговорювався раніше в процесі розгляду відправлення даних.

Використання *recv* у сокетах, орієнтованих на передавання повідомлень чи дайтаграм, має кілька особливостей. Якщо під час виклику *recv* розмір даних, що очікують оброблення, більший від наданого буфера, то після його повного заповнення виникає помилка *WSAEMSGSIZE*. Помилка перевищення розміру повідомлення відбувається тільки у випадку використання протоколів, орієнтованих на передавання повідомлень. Потокові протоколи буферизують дані, що надходять, і під час запиту програмою надають їх у повному обсязі, навіть якщо кількість даних, що очікують оброблення, більша розміру буфера. Таким чином, помилка *WSAEMSGS1ZE* не може виникнути за роботи з потоковими протоколами.

Функція ***WSARecv*** має додаткові, порівняно з *recv*, можливості: підтримує перекрите введення–виведення і фрагментарні дайтаграмні повідомлення.

```
int WSARecv(
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesRecvd,
LPDWORD lpFlags,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

Параметри:

s – сокет з'єднання;

lpBuffers і *dwBufferCount* – визначають буфери для приймання даних;

lpBuffers – вказівник на масив структур *WSABUF*;

dwBufferCount – визначає кількість таких структур у масиві;

lpNumberOfBytesReceived – у випадку завершення операції одержання даних вказує на кількість прийнятих цим викликом байтів;

lpFlags – може набувати значення *MSG_PEEK*, *MSG_OOB*, *MSG_PARTIAL* чи результату логічного АБО над будь-якими з цих параметрів.

У прапорця *MSG_PARTIAL*, залежно від способу використання, можуть бути різні значення і зміст. Для протоколів, орієнтованих на передавання

повідомлень, цей прапорець встановлюється після виклику *WSARecv* (якщо все повідомлення не може бути повернуте через нестачу місця у буфері). У цьому випадку кожен такий виклик *WSARecv* встановлює прапорець *MSG_PARTIAL*, доки повідомлення не буде прочитане повністю. Якщо цей прапорець передається як вхідний параметр, операція приймання даних має завершитися як тільки дані будуть доступні, навіть якщо це лише частина повідомлення. Прапорець *MSG_PARTIAL* використовується тільки з протоколами, орієнтованими на передавання повідомлень. Запис кожного протоколу у каталозі *Winsock* містить прапорець, що вказує на підтримку цієї можливості. Параметри *lpOverlapped* і *lpCompletionROUTINE* застосовуються в операціях перекритого введення–виведення.

Функція ***WSARecvDisconnect*** обернена до *WSASendDisconnect* і визначена так:

int WSARecvDisconnect(
SOCKET s,
LPWSABUF lpInboundDisconnectData);

Як і у *WSASendDisconnect*, першим її параметром є описувач сокета з'єднання. Другий параметр – недійсна структура *WSABUF* для приймання даних.

Функція отримує тільки дані про закриття з'єднання, відправлені з іншої сторони функцією *WSASendDisconnect*, її не можна використовувати для приймання звичайних даних. До того ж, відразу після прийняття даних вона припиняє приймання з віддаленої сторони, що еквівалентно виклику shutdown з параметром *SD_RECV*.

Функція ***WSARecvEx*** є спеціальним розширенням *Microsoft* для *Winsock 1*. Вона ідентична *recv* у всьому, крім того, що параметр *flags* передається за посиланням. Це дозволяє процесу приймача задавати прапорець *MSG_PARTIAL*.

*int PASCAL WSARecvEx(SOCKET s, char * buf, int len, int *flags);*

Якщо отримані дані є лише частиною повідомлення, то у параметрі *flags* повертається прапорець *MSG_PARTIAL*. Він використовується тільки з тими протоколами, що орієнтовані на передавання повідомлень. Коли під час прийняття неповного повідомлення прапорець *MSG_PARTIAL* передається як частина параметра *flags*, функція *WSARecvEx* завершується негайно, повертаючи прийняті дані. Якщо у буфері не вистачає місця, щоб прийняти повідомлення повністю, *WSARecvEx* поверне помилку

WSAEMSGSIZE, а решта даних, що не змогли бути прийняті, будуть відкинуті. Зверніть увагу на різницю між прапорцем *MSG_PARTIAL* і помилкою *WSAEMSGSIZE*: якщо виникла помилка, то це означає, що повідомлення надійшло повністю, однак вхідний буфер занадто малий для того, щоб його прийняти.

У *WSARecvEx* також можна використовувати прапорці *MSG_PEEK* і *MSG_OOB*.

6.9 Особливості передавання потоків даних

Більшість протоколів зі встановленням з'єднання між предавачем і приймачем є потоковими. Важливо враховувати, що за використання будь-якої функції відправлення чи приймання даних через потоковий сокет немає гарантії, що буде прочитано чи записано весь замовлений обсяг даних. Нехай потрібно відправити 2048 байтів із символьного буфера функцією *send*:

```
char sendbuff[2048];
int nBytes = 2048;
ret = send(s, sendbuff, nBytes, 0);
```

Функція *send* повідомить про відправлення менше 2048 байтів. Змінна *ret* буде містити кількість переданих байтів. Під час відправлення даних внутрішні буфери утримують їх до моменту відправлення безпосередньо через дріт. Причиною неповного відправлення може бути, наприклад, передавання великої кількості даних, водночас всі буфери занадто швидко заповняться. У *TCP/IP* існує так званий розмір вікна. Приймальна сторона регулює його, вказуючи кількість даних, які здатна прийняти. У разі переповнення даними одержувач може задати нульовий розмір вікна, щоб справитися з даними, які надійшли. Це призведе до призупинення відправлення даних, доки розмір вікна не стане більшим нуля.

Нехай на приймальному кінці розмір буфера буде 1024 байти. Отже, потрібно повторно відправити решту 1024 байтів. Відправлення всього вмісту буфера забезпечить такий фрагмент програми:

```
char sendbuff[2048];
int nBytes = 2048, nLeft, idx;
nLeft = nBytes;
idx = 0;
```

```
int ret;
while (nLeft > 0)
{
    ret = send(s, &sendbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Помилка
    }
    nLeft -= ret;
    idx += ret;
}
```

Це справедливо і у разі потокового отримання даних, але не так важливо. Наприклад, якщо в усіх повідомлень однаковий розмір (512 байтів), то прочитати їх можна так:

```
char recvbuff[1024];
int ret, nLeft, idx;
nLeft = 512;
idx = 0;
while (nLeft > 0)
{
    ret = recv(s, &recvbuff[idx], nLeft, 0);
    if (ret == SOCKET_ERROR)
    {
        // Помилка .
    }
    idx += ret;
    nLeft-=ret;
}
```

Ситуація ускладниться, якщо розмір повідомлень буде змінюватись. Тоді потрібно реалізувати власний протокол, що повідомляє одержувачу про розмір отриманого повідомлення. Наприклад, перші 4 байти повідомлення можуть містити його розмір.

6.9.1 Комплексне введення-виведення

Вперше принцип комплексного введення–виведення (*Scatter-Gather I/O*) був застосований у функціях *recv* і *write* сокетів Берклі (*Berkeley Sockets*). У *Winsock* 2 його підтримують функції *WSARecv*, *WSARecvFrom*, *WSASend*, *WSASendTo*. Комплексне введення–виведення найчастіше використовують програми, що відправляють і приймають дані у специфічному форматі. Наприклад, якщо передані клієнтом серверу повідомлення мають

складатися з фіксованого 32-байтового заголовка, що визначає деякі дії, 64-байтового блока даних і закінчуватися 16-байтовою контрольною сумою. У цьому випадку функція *WSASend* може бути викликана для масиву відповідних трьох структур *WSABUF*. На приймальній стороні одним із вхідних параметрів викликуваної функції *WSARecv* також мають бути три структури *WSABUF*, що містять 32, 64 і 16 байтів.

Під час використання потокових сокетів операції комплексного введення–виведення інтерпретують кілька буферів даних як один неперервний. Функція прийняття даних може завершитися раніше, ніж будуть заповнені всі буфери. У сокетах, орієнтованих на передавання повідомлень, кожна операція одержання даних приймає одне повідомлення, довжина якого не більша розміру буфера. Якщо у буфері недостатньо місця, виклик закінчується помилкою *WSAEMSGSIZE* і дані урізаються до розміру доступного простору. У протоколах, що підтримують фрагментарні повідомлення, для запобігання втрати даних можна використовувати прапорець *MSG_PARTIAL*.

6.9.2 Завершення сеансу

Після закінчення роботи із сокетом необхідно викликати функцію *closesocket* для закриття з'єднання і звільнення всіх ресурсів, зв'язаних з описувачем сокета, однак її неправильне використання може призвести до втрати даних. Тому перед викликом *closesocket* потрібно коректно завершити сеанс. Правильно написана програма повідомляє одержувача про закінчення відправлення даних. Так само має вчинити і віддалений комп'ютер. Така поведінка називається коректним завершенням сеансу і здійснюється за допомогою функції *shutdown*.

Функція ***shutdown***:

int shutdown(SOCKET s, int how);

Параметр *how* може набувати значення *SD_RECEIVE*, *SD_SEND* чи *SD_BOTH*. Значення *SD_RECEIVE* забороняє виклики функцій приймання даних; на протоколи нижнього рівня це не діє. Якщо у черзі *TCP*-сокета є дані або вони надходять пізніше, з'єднання все одно закривається. *UDP*-сокети в аналогічній ситуації продовжують приймати дані і ставити їх у чергу. *SD_SEND* забороняє виклики функції відправлення даних. У випадку *TCP*-сокетів після підтвердження одержувачем приймання усіх відправлених даних передається пакет *FIN*. Нарешті, *SD_BOTH* забороняє як

приймання, так і відправлення.

Функція ***closesocket*** закриває сокет. Вона визначена так:

int closesocket (SOCKET s);

Після виклику *closesocket* всі подальші операції із сокетом закінчаться помилкою *WSAENOTSOCK*. Якщо не існує інших посилань на сокет, то усі зв'язані з дескриптором ресурси будуть звільнені, зокрема й дані у черзі.

Асинхронні виклики, які очікують надходження даних, скасовуються без повідомлення. Операції, які очікують перекритого введення–виведення, також анулюються. Усі події, які виконуються у цей момент, процедура і порт завершення, зв'язані з перекритим введенням–виведенням, завершаться помилкою з повідомленням *WSA_OPERATION_ABORTED*.

6.9.3 Приклади

У більшості програм для приймання інформації використовують тільки функції recv чи *WSARecv*, а для відправлення – функції send чи *WSASend*. Інші функції забезпечують додаткові можливості і рідко використовуються (чи підтримуються тільки транспортними протоколами).

Розглянемо приклад клієнт-серверної взаємодії з урахуванням описаних принципів і функцій. У лістингу 6.1 наведено код простого сервера. Програма створює сокет, прив'язує його до локального IP-інтерфейсу і порту та слухає з'єднання клієнта. Після приймання від клієнта запиту на з'єднання створюється новий сокет, через який відбувається обмін інформацією.

Лістинг 6.1 – Приклад сервера *TCP*, що приймає з'єднання від одного клієнта

```
// Ім'я модуля: Server1.cpp
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include "windows.h"
#include <stdio.h>
#include <conio.h>
#pragma comment(lib, "ws2_32.lib")
#define PORT 5555
#define BUFFLEN 4096
```

```
char Msg[]="Hi from Server1!";
int main()
{
    WSADATA Wsadata;
    SOCKET ListenSock, ClientSock;
    sockaddr_in ClientAddr, LocalAddr;
    int AddrLen, RetCod;
    char Buffer[BUFFLEN];
    WSAStartup(0x0202, &Wsadata);
    ListenSock=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    LocalAddr.sin_family=AF_INET;
    LocalAddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
    LocalAddr.sin_port=htons(PORT);
    bind(ListenSock, (sockaddr*)&LocalAddr, sizeof(LocalAddr));
    listen(ListenSock, 8);
    AddrLen=sizeof(ClientAddr);
    ClientSock=accept(ListenSock, (sockaddr*)&ClientAddr, &AddrLen);
    printf("Connect client. Addr: %s.\n", inet_ntoa(ClientAddr.sin_addr));
    RetCod=recv(ClientSock, Buffer, BUFFLEN,0);
    Buffer[RetCod]=0;
    puts("Message from client: ");
    puts(Buffer);
    send(ClientSock, Msg, sizeof(Msg), 0);
    _getch();
    closesocket(ListenSock);
    closesocket(ClientSock);
    WSACleanup();
    return 0;
}
```

У лістингу 6.2 наведено приклад сервера, що виконує під'єднання багатьох клієнтів завдяки використанню потоків.

Лістинг 6.2 – Приклад сервера *TCP*, що приймає з'єднання від багатьох клієнтів

```
// Ім'я модуля: Server2.cpp
#define _WINSOCK_DEPRECATED_NO_WARNINGS
```

```
#include "winsock2.h"
#include "windows.h"
#include <stdio.h>
#include <conio.h>
#pragma comment(lib, "ws2_32.lib")
#define PORT 55555
#define BUFFLEN 4096
char Msg[]="Hi from Server2!";
int ClientNom=0;
sockaddr_in ClientAddr;
DWORD WINAPI ClientThread(LPVOID lpParam);
int main()
{
    HANDLE hThread;
    int ThreadID;
    WSADATA Wsadata;
    SOCKET ListenSock, ClientSock;
    sockaddr_in LocalAddr;
    WSAStartup(0x0202, &Wsadata);
    ListenSock=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    LocalAddr.sin_family=AF_INET;
    LocalAddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
    LocalAddr.sin_port=htons(PORT);
    bind(ListenSock, (sockaddr*)&LocalAddr, sizeof(LocalAddr));
    listen(ListenSock, 8);
    while(!_kbhit())
    {
        int AddrLen;
        AddrLen=sizeof(ClientAddr);
        ClientSock=accept(ListenSock, (sockaddr*)&ClientAddr,
        &AddrLen);
        hThread=CreateThread(NULL, 0, ClientThread, (LPVOID)ClientSock,
    0,(LPDWORD)&ThreadID);
        CloseHandle(hThread);
    }
    WSACleanup();
```

```
    return 0;
}
DWORD WINAPI ClientThread(LPVOID lpParam)
{
    int Nom=ClientNom++;
    printf("CONNECT to client %d with addr:\n%s\n", Nom,
inet_ntoa(ClientAddr.sin_addr));
    DWORD RetCod;
    SOCKET s=(SOCKET)lpParam;
    char Buffer[BUFFLEN];
    RetCod=recv(s, Buffer, BUFFLEN,0);
    Buffer[RetCod]=0;
    printf("MESSAGE from client %d:\n", Nom);
    puts(Buffer);
    _getch();
    send(s, Msg, sizeof(Msg),0);
    closesocket(s);
    return 0;
}
```

Програму клієнта *TCP* із встановленням з'єднання подано у лістингу 6.3.

Спочатку клієнт створює сокет за допомогою виклику функції *socket*. Потім за допомогою виклику функції *connect* клієнт з'єднується із сервером. Як тільки з'єднання встановлено, клієнт за допомогою виклику функції *send* відправляє серверу повідомлення.

Після відправлення повідомлення клієнт очікує відповіді сервера і отримує її за допомогою виклику функції *recv*. Всі отримані через сокет дані виводяться на екран.

Лістинг 6.3 – Приклад клієнта TCP

```
//Ім'я модуля: Client.cpp
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include "windows.h"
#include <stdio.h>
#include <conio.h>
#pragma comment(lib, "ws2_32.lib")
```

```
#define PORT 55555
#define BUFFLEN 4096
char IPAddr[]="127.0.0.1";
char Msg[]="Hi from Client!";
int main()
{
    WSADATA Wsadata;
    SOCKET s;
    sockaddr_in ServerAddr;
    int RetCod;
    char Buffer[BUFFLEN]="";
    WSAStartup(0x0202, &Wsadata);
    s=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    ServerAddr.sin_family=AF_INET;
    ServerAddr.sin_port=htons(PORT);
    ServerAddr.sin_addr.S_un.S_addr=inet_addr(IPAddr);
    if(connect(s, (sockaddr*)&ServerAddr,
     sizeof(ServerAddr))==SOCKET_ERROR)
    {
        puts("ERROR! Can't connect to Server.");
        _getch();
        return 1;
    }
    printf("CONNECT to server with addr:\n%s.\n",
    inet_ntoa(ServerAddr.sin_addr));
    _getch();
    send(s, Msg, sizeof(Msg),0);
    RetCod=recv(s, Buffer, sizeof(Buffer),0);
    Buffer[RetCod]=0;
    puts("MESSAGE from Server: ");
    puts(Buffer);
    _getch();
    closesocket(s);
    WSACleanup();
    return 0;
}
```

6.10 Протоколи, що не потребують з'єднання

Принцип дії таких протоколів інший, тому що у них використовуються інші методи відправлення і приймання даних. У таких протоколах архітектурою обміну є не клієнт-сервер, а приймач-передавач. Різниця між приймачем і передавачем полягає тільки у тому, хто викликає функції приймання, а хто – передавання інформації. Функція приймання завжди має викликатись першою. Будь-який з варіантів функції приймання виконується у блокувальному режимі і очікує отримання даних.

6.10.1 Приймач

Спочатку створюють сокет функцією *socket* чи *WSASocket*. Потім виконують прив'язку сокета до локальної адреси функцією *bind* (як і у випадку протоколів, орієнтованих на сеанси). Різниця у тому, що не викликають функції *listen* та *accept*. Замість цього просто передають або отримують дані. Оскільки у цьому випадку з'єднання немає, приймальний сокет може одержувати дайтаграми від будь-якої машини у мережі.

Найпростіша функція приймання – ***recvfrom***:

```
int recvfrom(
SOCKET s,
char * buf,
int len,
int flags,
sockaddr * from,
int * fromlen );
```

Перші чотири параметри такі самі, як і для функції recv, включно з допустимими значеннями для *flags*: *0, MSG_OOB і MSG_PEEK*;

from – структура *sockaddr* передавального сокета, на розмір якої посилається fromlen.

Після повернення виклику структура *sockaddr* буде містити адресу робочої станції, що відправляє дані.

У *Winsock* 2 для приймання інформації без встановлення з'єднання застосовується ***WSARecvFrom***:

```
int WSARecvFrom(
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
```

LPDWORD lpNumberOfBytesRecvd,
LPDWORD lpFlags,
*sockaddr * lpFrom,*
LPINT lpFromlen,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

Функції можна надати один чи кілька буферів *WSABUF*, вказавши їх кількість у *dwBufferCount*, – у цьому випадку можливе комплексне введення–виведення. Сумарна кількість зчитаних байтів передається у *lpNumberOfBytesRecvd*. Під час виклику функції *WSARecvFrom lpFlags* може приймати такі значення: *0* (за відсутності параметрів), *MSG_OOB*, *MSG_PEEK* чи *MSG_PARTIAL*. Ці прапорці можна комбінувати логічною операцією АБО. Якщо у разі виклику функції задано прапорець *MSG_PARTIAL*, постачальник перешле дані навіть у випадку приймання лише частини повідомлення. Після повернення прапорець задається у *MSG_PARTIAL*, тільки якщо повідомлення прийняте частково. Після повернення *WSARecvFrom* присвоїть параметру *lpFrom* (вказівник на структуру *sockaddr*) адресу комп'ютера-відправника. Знову ж, параметр *lpFromLen* вказує на розмір структури *SOCKADDR*, однак у цій функції він є вказівником на *DWORD*. Два останніх параметри – *lpOverlapped* і *lpCompletionRoutine* – використовуються для перекритого введення–виведення.

Інший спосіб приймання–відправлення даних у сокетах, що не потребують з'єднання, – встановлення з'єднання (хоч це і звучить дивно). Після створення сокета і зв'язування з локальною адресою можна викликати connect чи *WSAConnect*, присвоївши параметру *sockaddr* адресу віддаленого комп'ютера, з яким необхідно зв'язатися. Фактично ніякого з'єднання не відбувається. Адреса сокета, переданого у функцію з'єднання, асоціюється із сокетом, щоб було можна використовувати функції *recv* і *WSARecv* замість *recvfrom* чи *WSARecvFrom* (оскільки джерело даних відоме). Якщо програмі потрібно одночасно зв'язуватися лише з однією кінцевою точкою, використовується можливість під'єднання сокета дайтаграм.

6.10.2 Передавач

Є два способи відправлення даних через сокет, за яких не потрібне з'єднання. Перший і найпростіший – створити сокет і викликати функцію *sendto* чи *WSASendTo*.

Розглянемо функцію ***sendto***:

int sendto(
SOCKET s,
*const char * buf,*
int len,
int flags,
*const sockaddr * to,*
int tolen);

Параметри цієї функції такі самі, як і у *recvfrom*, за винятком *buf* – буфера даних для відправлення і *tolen* – розміра структури *sockaddr*.

Параметр *to* – вказівник на структуру *sockaddr* з адресою приймальної робочої станції.

Також можна використовувати функцію ***WSASendTo*** з *Winsock 2*:

int WSASendTo(
SOCKET s,
LPWSABUF lpBuffers,
DWORD dwBufferCount,
LPDWORD lpNumberOfBytesSent,
DWORD dwFlags,
*const sockaddr * lpTo,*
int iToLen,
LPWSAOVERLAPPED lpOverlapped,
LPWSAOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);

Функція *WSASendTo* аналогічна своїй попередниці. Вона приймає вказівник на одну чи кілька структур *WSABUF* з даними для відправлення одержувачу у вигляді параметра *lpBuffers*, а *dwBufferCount* задає кількість структур.

Для комплексного введення–виведення можна відправити кілька структур *WSABUF*. Перед виходом *WSASendTo* присвоює четвертому параметру – *lpNumberOfBytesSent* – кількість реально відправлених одержувачу байтів. Параметр *lpTo* – структура *sockaddr* для цього протоколу з адресою приймача. Параметр *iToLen* – довжина структури *sockaddr*. Два останніх параметри – *lpOverlapped* і *lpCompletionRoutine* – використовуються для перекритого введення–виведення.

Як і під час отримання даних, сокет, що не потребує з'єднання, можна підключати до адреси кінцевої точки і відправляти дані функціями *send* і

WSASend. Після створення цього прив'язування не можна використовувати *sendto* і *WSASendTo* з іншою адресою – буде видано помилку *WSAEISCONN*.

6.10.3 Особливості протоколів, що не потребують з'єднання

Протоколи, що потребують з'єднання, орієнтовані на передавання повідомлень. Тому потрібно враховувати такі фактори.

По-перше, оскільки орієнтовані на передавання повідомлень протоколи зберігають межі повідомлень, то дані, поставлені у чергу для відправлення, блокуються до завершення виконання функції відправлення. Якщо відправлення не може бути завершене, то за асинхронного або неблокувального режиму введення–виведення функція відправлення поверне помилку *WSAEWOULDBLOCK*. Це означає, що операційна система не змогла обробити дані і потрібно викликати функцію відправлення повторно. Важливим є те, що в орієнтованих на повідомлення протоколах запис відбувається тільки внаслідок самостійної дії.

По-друге, під час виклику функції приймання потрібно надати місткий буфер, інакше функція видасть помилку *WSAEMSGSIZE*: буфер заповнений, і дані, що залишилися, відкидаються. Винятком є протоколи, що підтримують обмін фрагментарними повідомленнями, наприклад *AppleTalk PAP*. Якщо було прийнято лише частину повідомлення, функція *WSARecvEx* присвоює параметру *flag* значення *MSG_PARTIAL*.

Для передавання дайтаграм на основі протоколів, що підтримують фрагментарні повідомлення, використовують одну з функцій *WSARecv* чи *recv*. Після чергового виклику *recv* може бути отримано лише частину дайтаграми. Проте під час виклику *recv* не можна відстежити повноту зчитування повідомлення (це задача програміста). Через таке обмеження зручніше використовувати функцію *WSARecvEx*, що дозволяє задавати і зчитувати прапорець *MSG_PARTIAL*. Функції *Winsock 2 WSARecv* і *WSARecvFrom* також підтримують роботу з цим прапорцем.

Якщо замість *bind* викликається *sendto* чи *WSASendTo*, чи спочатку встановлюється з'єднання, мережевий стек автоматично вибирає локальну *IP*-адресу з таблиці маршрутизації. Таким чином, якщо спочатку було виконано прив'язку, початкова *IP*-адреса може бути неправильною і не відповідати інтерфейсу, з якого фактично була відправлена дайтаграма.

6.10.4 Звільнення ресурсів сокета

Оскільки з'єднання не встановлюється, його формального розриву чи

коректного закриття не потрібно. Після припинення відправлення чи одержання даних просто викликається функція *closesocket* з описувачем необхідного сокета, внаслідок чого звільняються усі виділені йому ресурси.

6.10.5 Приклади

Розглянемо приклади відправлення і приймання дайтаграм.

Приймання дайтаграм просте. Спочатку необхідно створити сокет, потім прив'язати його до локального інтерфейсу. Для прив'язки до інтерфейсу за замовчуванням потрібно визначити його *IP*-адресу функцією *getsockname*. Як параметр їй передається сокет, а вона повертає структуру *sockaddr_in*, яка вказує зв'язаний із сокетом інтерфейс. Потім для читання вхідних даних викликається функція *recvfrom*. Ми використовуємо *recvfrom*, тому що нас не цікавлять фрагментарні повідомлення, оскільки протокол *UDP* не підтримує їх передавання. Фактично, стек *TCP/IP* намагається зібрати велике повідомлення з отриманих фрагментів. Якщо один чи кілька фрагментів відсутні або порушено порядок їхнього проходження, стек відкидає все повідомлення. У лістингу 6.4 наведено код приймача.

Лістинг 6.4 – Приймач, що не потребує встановлення з'єднання

```
// Ім'я модуля: Receiver.cpp
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include <windows.h>
#include "stdio.h"
#include <conio.h>
#pragma comment(lib, "ws2_32.lib")
#define R_PORT 55555
#define S_PORT 55556
#define ADDR "127.0.0.1"
#define BUFFLEN 256
char Buffer[BUFFLEN];
int main()
{
    WSADATA wsd;
    SOCKET s;
    int RetCod;
    int AddrLen;
    sockaddr_in SenderAddr,ReceiverAddr;
```

```
    WSAStartup(0x0202,&wsd);
    s=socket(AF_INET,SOCK_DGRAM,0);
    SenderAddr.sin_addr.S_un.S_addr=inet_addr(ADDR);
    SenderAddr.sin_port=htons(S_PORT);
    SenderAddr.sin_family=AF_INET;
    ReceiverAddr.sin_addr.S_un.S_addr=inet_addr(ADDR);
    ReceiverAddr.sin_port=htons(R_PORT);
    ReceiverAddr.sin_family=AF_INET;
    bind(s,(sockaddr*)&ReceiverAddr,sizeof(ReceiverAddr));
    AddrLen=sizeof(SenderAddr);
    RetCod=recvfrom(s,Buffer,BUFFLEN,0,(sockaddr*)&SenderAddr, &AddrLen);
    Buffer[RetCod]=0;
    puts(Buffer);
    _getch();
    closesocket(s);
    return 0;
}
```

Замість функції *recvfrom*, яка вказує адресу відправника, можна використовувати функцію *recv*, якщо перед нею викликати функцію connect:

```
connect(s,(sockaddr*)&SenderAddr,sizeof(SenderAddr));
RetCod=recv(s,Buffer,BUFFLEN,0);
```

В цьому випадку з'єднання між клієнтом і сервером не встановлюється, а лише виконується прив'язування до сокета адреси, за якою у подальшому відбувається відправлення. В обох випадках можна приймати повідомлення від будь-якого комп'ютера, якщо у структурі *SenderAddr* у полі *S_addr* задати спеціальну адресу:

```
SenderAddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
```

У лістингу 6.5 наведено код відправника, який не потребує з'єднання.

Лістинг 6.5 – Відправник, що не потребує встановлення з'єднання

```
// Ім'я модуля: Sender.c
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include <windows.h>
#include "stdio.h"
#include <conio.h>
#pragma comment(lib, "ws2_32.lib")
```

```
#define R_PORT 55555
#define S_PORT 55556
#define ADDR "127.0.0.1"
char Msg[]="Hello from Sender";
int main()
{
    WSADATA wsd;
    SOCKET s;
    sockaddr_in SenderAddr,ReceiverAddr;
    WSAStartup(0x0202,&wsd);
    s=socket(AF_INET,SOCK_DGRAM,0);
    SenderAddr.sin_addr.S_un.S_addr=inet_addr(ADDR);
    SenderAddr.sin_port=htons(S_PORT);
    SenderAddr.sin_family=AF_INET;
    ReceiverAddr.sin_addr.S_un.S_addr=inet_addr(ADDR);
    ReceiverAddr.sin_port=htons(R_PORT);
    ReceiverAddr.sin_family=AF_INET;
    bind(s,(sockaddr*)&SenderAddr,sizeof(SenderAddr));
    sendto(s,Msg,sizeof(Msg),0,(sockaddr*)&ReceiverAddr,sizeof(ReceiverAddr));
    _getch();
    closesocket(s);
    return 0;
}
```

У випадку, якщо одна і та сама програма може бути приймачем або передавачем, то в режимі передавання потрібно використовувати один локальний порт, а в режимі приймання – інший, оскільки передавання буде некоректним, якщо його виконувати на одному комп'ютері. У лістингу 6.6 наведено приклад приймача-передавача, що не потребує з'єднання і може здійснювати обмін інформацією на одному комп'ютері. Тут використовуються функції *sendto* і *recvfrom*.

Лістинг 6.6 – Приймач-передавач, що не потребує встановлення з'єднання

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include"windows.h"
```

```
#include <iostream>
#pragma comment(lib, "ws2_32.lib")
using namespace std;

int main()
{
    WSADATA wsd;
    SOCKET s;
    sockaddr_in recvAddr, sendAddr;
    char buf[256];
    int addrLen;
    WSAStartup(0x0202, &wsd);
    s = socket(AF_INET, SOCK_DGRAM, 0);
    recvAddr.sin_family = AF_INET;
    recvAddr.sin_port = htons(44444);
    recvAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    sendAddr.sin_family = AF_INET;
    sendAddr.sin_port = htons(55555);
    sendAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    char mode;
    cout << "mode(s,r): ";
    cin >> mode;
    switch(mode)
    {
    case 's':
        bind(s, (sockaddr*)&sendAddr, sizeof(sendAddr));
        sendto(s, "This is message!", 17, 0, (sockaddr*)&recvAddr, sizeof(recvAddr));
        break;
    case 'r':
        addrLen = sizeof(sendAddr);
        bind(s, (sockaddr*)&recvAddr, sizeof(recvAddr));
        recvfrom(s, buf, 255, 0, (sockaddr*)&sendAddr, &addrLen);
        cout << buf << endl;
        break;
    }
    system("pause");
    closesocket(s);
    return 0;
}
```

У іншому випадку замість функції *sendto* можна використовувати

функцію *send*, якщо перед нею викликати функцію *connect*:

```
connect(s,(sockaddr*)&ReceiverAddr,sizeof(ReceiverAddr));
send(s,Msg,sizeof(Msg),0);
```

У лістингу 6.7 наведено приклад приймача-передавача, що не потребує з'єднання і може здійснювати обмін інформацією на одному комп'ютері. Тут використовуються функції *connect*, *send* і *recv*.

Лістинг 6.7 – Приймач-передавач, що не потребує встановлення з'єднання, проте використовує встановлення з'єднання

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include"windows.h"
#include <iostream>
#pragma comment(lib, "ws2_32.lib")
using namespace std;

int main()
{
    WSADATA wsd;
    SOCKET s;
    sockaddr_in recvAddr, sendAddr;
    char buf[256];
    int addrLen;
    WSAStartup(0x0202, &wsd);
    s = socket(AF_INET, SOCK_DGRAM, 0);
    recvAddr.sin_family = AF_INET;
    recvAddr.sin_port = htons(44444);
    recvAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    sendAddr.sin_family = AF_INET;
    sendAddr.sin_port = htons(55555);
    sendAddr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    char mode;
    cout << "mode(s,r): ";
    cin >> mode;
    switch(mode)
    {
    case 's':
        bind(s, (sockaddr*)&sendAddr, sizeof(sendAddr));
        connect(s, (sockaddr*)&recvAddr, sizeof(recvAddr));
        send(s, "This is message!", 17, 0);
        break;
```

```
    case 'r':
        addrLen = sizeof(sendAddr);
        bind(s, (sockaddr*)&recvAddr, sizeof(recvAddr));
        connect(s, (sockaddr*)&sendAddr, sizeof(sendAddr));
        recv(s, buf, 255, 0);
        cout << buf << endl;
        break;
    }
    system("pause");
    closesocket(s);
    _getch();
    return 0;
}
```

6.11 Додаткові функції Winsock 2

Функції *WSAAddressToString* і *WSAStringToAddress* у *Winsock 2* забезпечують незалежний від протоколу спосіб перетворення структури *sockaddr* протоколу у форматований рядок символів і навпаки. Оскільки ці функції не залежать від протоколу, потрібно, щоб транспортний протокол підтримував перетворення рядків. У цей час означені функції працюють тільки для сімейств адрес *AF_INET* і *AF_ATM*.

Функція ***WSAAddressToString*** визначена так:

```
int WSAAddressToString(
LPSOCKADDR lpsaAddress,
DWORD dwAddressLength,
LPWSAPROTOCOL_INFO lpProtocolInfo,
OUT LPTSTR lpszAddressString,
IN OUT LPDWORD lpdwAddressStringLength);
```

Параметри:

lpsaAddress – вказівник на структуру *sockaddr* для конкретного протоколу, що містить адресу, яку потрібно перетворити на рядок;

dwAddressLength – розмір структури першого параметра для кожного протоколу;

lpProtocolInfo – постачальник протоколу. Постачальників протоколу можна знайти функцією *WSAEnumProtocols*. Якщо параметру присвоюється значення *NULL*, виклик використовує постачальника першого протоколу,

який підтримує сім'ю адрес, вказану у параметрі *lpsaAddress*;

lpszAddressString – буфер, де зберігається зручна для читання адреса;

lpdwAddressStringLength – розмір *lpszAddressString*. Під час виведення у ньому повертається довжина рядка, фактично скопійована у *lpszAddressString*. Якщо наданий буфер малий, функція видає помилку *WSAEFAULT*, а у параметрі *lpdwAddressStringLength* повернеться необхідний розмір у байтах.

Функція ***WSAStringToAddress*** перетворить зручну для читання адресу у структуру *sockaddr*:

```
int WSAStringToAddress(
LPTSTR AddressString,
int AddressFamily,
LPWSAPROTOCOl_INFO lpProtocolInfo,
LPSOCKADDR lpAddress,
LPINT lpAddressLength);
```

Параметри функції:

AddressString – адреса у такому форматі:

• для *IP – XXX.XXX.XXX.XXX:Y*, де *X* є октетом у рядку *IP*-адреси, а *Y* – номер порту;

• для *ATM – NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN NNNNNNNN*, де 40 символів *N* є 20-байтовою *ATM*-адресою у шістнадцятковому форматі;

AddressFamily – тип сім'ї адрес для параметра *AddressString*;

lpProtocolInfo – постачальник протоколу. Якщо присвоїти йому *NULL*, то *Winsock* буде шукати перший доступний постачальник протоколу відповідно до значення *AddressFamily*. Для вказання конкретного постачальника можна спочатку за допомогою *WSAEnumProtocols* отримати список доступних постачальників протоколу, встановлених у системі;

lpAddress – структура *sockaddr*, що приймає інформацію в адресному рядку;

lpAddressLength – розмір структури *sockaddr*, яка містить результат.

Функція ***getpeername***. Повертає інформацію про адресу під'єднаного партнера:

```
int getpeername(
SOCKET s,
sockaddr * name,
int FAR* namelen);
```

Параметри функції:

s – сокет для з'єднання;

name – вказівник на пусту структуру *sockaddr*;

namelen – довжина структури *sockaddr*.

Для сокетів дайтаграм ця функція записує адресу, передану виклику з'єднання (за винятком адрес, переданих у виклик *sendto* чи *WSASendTo*).

Функція ***getsockname***. Ця функція обернена до *getpeername* і повертає адресну інформацію локального інтерфейсу певного сокета:

```
int getsockname(
SOCKET s,
sockaddr * name,
int FAR* namelen );
```

Використовуються ті самі параметри, що і для *getpeername*, однак повертається інформація про локальну адресу. У випадку *TCP* адреса збігається з сокетом сервера, що слухає на заданому порту і *IP*-інтерфейсі.

Функція ***WSADuplicateSocket***. Ця функція застосовується для створення структури *WSAPROTOCOL_INFO*, яку можна передати іншому процесу, що дозволить йому відкрити описувач того самого базового сокета й оперувати цим ресурсом. Така необхідність виникає тільки між процесами. Потоки в одному й тому самому процесі можуть вільно передавати описувачі сокета. Функція визначена так:

```
int WSADuplicateSocket(
SOCKET s,
DWORD dwProcessId,
LPWSAPROTOCOL_INFO lpProtocolInfo);
```

Параметри функції:

s – описувач сокета для копіювання;

dwProcessId – код процесу, що буде використовувати скопійований сокет;

lpProtocolInfo – вказівник на структуру *WSAPROTOCOL_INFO*, що містить інформацію, необхідну цільовому процесу для відкриття копії описувача сокета. У деяких випадках процеси мають взаємодіяти між собою таким чином, щоб поточний процес мав можливість передавати структуру *WSAPROTOCOL_INFO* цільовому процесу, який потім міг би використати її для створення описувача сокета (за допомогою функції *WSASockef*).

Описувачі в обох сокетах можна використовувати для введення-виведення незалежно, однак *Winsock* не забезпечує контроль за доступом.

Тому програмісту необхідно передбачити деякі способи синхронізації. Вся інформація про стан будь-якого сокета зберігається в одному місці для всіх його описувачів, тому що у функцію копіюються описувачі сокета, а не сам сокет. Наприклад, значення будь-якого параметра сокета, заданого для одного з описувачів викликом функції *setsockopt*, можна потім отримати, викликавши функцію *getsockopt* для будь-якого іншого його описувача. Якщо процес викликає *closesocket* для закривання копії сокета, описувач копії у цьому процесі звільняється, однак сокет залишиться відкритим, доки функція *closesocket* не буде викликана для останнього його описувача.

Існують певні особливості сокетів під час використання функцій *WSAAsyncSelect* і *WSAEventSelect*. Ці функції використовуються у випадку асинхронного введення-виведення. В процесі їхнього виклику з кожним із спільних описувачів скасовується реєстрація будь-якої попередньої події для сокета, незалежно від того, який із спільних описувачів використовувався для реєстрації. Тому, наприклад, через спільний для процесів *A* і *B* сокет не можна передавати події *FD_READ* процесу *A* і події *FD_WRITE* процесу *B*. Якщо необхідно передавати повідомлення про події через обидва описувачі, потрібно використовувати потоки замість процесів.

Функція ***TransmitFile***. Це розширення *Microsoft* дозволяє швидко передавати дані з файла. Висока ефективність обумовлена тим, що все передавання даних відбувається у режимі ядра. Якщо зчитується блок даних з файла, а потім викликається *send* чи *WSASend*, то відбуваються багаторазові переключення між режимами ядра і користувача. Під час виклику *TransmitFile* весь процес читання і відправлення виконується у режимі ядра.

Функція визначена так:

```
BOOL TransmitFile(
SOCKET hSocket,
HANDLE hFile,
DWORD nNumberOfBytesToWrite,
DWORD nNumberOfBytesPerSend,
LPOVERLAPPED lpOverlapped,
LPTRANSMIT_FILE_BUFFERS lpTransmitBuffers,
DWORD dwFlags);
```

Параметри функції:

hSocket – під'єднаний сокет, яким буде передано файл;

hFile – описувач відкритого файла;

nNumberOfBytesToWrite – кількість записуваних з файла байтів. Якщо цей параметр дорівнює нулю, файл відправляється повністю;

nNumberOfBytesPerSend – розмір блоків даних, які відправляються, для операцій записування. Наприклад, якщо розмір дорівнює 2048, *TransmitFile* передасть файл порціями по 2 Кбайт, якщо нуль – використовується стандартний розмір відправлення;

lpOverlapped – вказівник на структуру *OVERLAPPED*, що використовується у разі перекритого введення-виведення;

lpTransmitBuffers – вказівник на структуру *TRANSMIT_FILE_BUFFERS* з даними, що потрібно відправити до і після передавання файла:

```
typedef struct _TRANSMIT_FILE_BUFFERS
{
    PVOID Head;
    DWORD HeadLength;
    PVOID Tail;
    DWORD TailLength;
} TRANSMIT_FILE_BUFFERS;
```

Поля структури:

Head – вказівник на дані, що відправляються перед передаванням файла;

HeadLength – кількість заздалегідь переданих даних;

Tail – посилається на дані, що відправляються після передавання файла;

TailLength – кількість переданих потім байтів;

dwFlags – режими роботи *TransmitFile*.

Прапорці можуть мати такі значення:

TF_DISCONNECT – ініціює закриття сокета після передавання даних;

TF_REUSE_SOCKET – дозволяє повторно використовувати у функції *AcceptEx* описувач сокета як клієнтського сокета;

TF_USE_DEFAULT_WORKER і *TF_USE_SYSTEM_THREAD* – вказують, що передавання має виконуватись у контексті стандартного системного процесу. Ці прапорці корисні у випадку передавання великих файлів;

TF_USE_KERNEL_APC – вказує, що передавання має виконуватись ядром за допомогою асинхронних викликів процедур (*Asynchronous Procedure Call, APC*). Це істотно збільшує продуктивність, якщо для зчитування файла у кеш потрібна лише одна операція читання;

TF_WRITE_BEHIND – вказує, що *TransmitFile* може завершитися, не одержавши підтверджень про приймання даних від віддаленої системи.

6.12 Контрольні питання

1. Що таке сокет? Якими параметрами характеризується сокет? Які переваги має використання сокетів порівняно з іншими мережевими технологіями?
2. Яка функція ініціалізує бібліотеку сокетів?
3. Опишіть поля структури *WSADATA*.
4. Яка функція створює сокет? Опишіть її параметри.
5. Як відбувається адресація протоколу *IP*? Будова *IP*-адреси. Спеціальні адреси.
6. Опишіть поля структури *sockaddr_in*.
7. Що таке порт? На які групи поділяються номери портів?
8. Який порядок байтів в адресах і портах? Назвіть функції, які задають порядок байтів, та опишіть їх параметри.
9. Які функції слугують для перетворення адрес? Які параметри мають ці функції?
10. Які послідовності функцій мають викликати клієнт і сервер?
11. Опишіть функції, що використовуються для зв'язування і прослуховування сокета. Опишіть їхні параметри.
12. Опишіть функції, що використовуються для приєднання сокета, і їхні параметри.
13. Опишіть функції, що використовуються для встановлення з'єднання, і їхні параметри.
14. Опишіть функції, що використовуються для потокового передавання даних, та їхні параметри.
15. Опишіть функції, що виконують термінове передавання даних мережею.
16. Опишіть функції, що використовуються для потокового приймання даних, і їхні параметри.
17. Як можна запобігти переповненню буферів у разі потокового передавання даних?
18. Що таке комплексне введення-виведення даних і як воно реалізовано у сокетах?
19. Які функції потрібно викликати для завершення сеансу зі встановленням з'єднання? Опишіть їхні параметри.
20. Напишіть програму *TCP*-сервера, що обмінюється інформацією з одним клієнтом.

21. Напишіть програму *TCP*-сервера, що обмінюється інформацією з багатьма клієнтами.
22. Напишіть програму *TCP*-клієнта.
23. Який принцип дії програм, що використовують протоколи без встановлення з'єднання?
24. Які функції використовуються для дайтаграмного передавання інформації? Опишіть їхні параметри.
25. Які функції використовуються для дайтаграмного отримання інформації? Опишіть їхні параметри.
26. Для чого під час передавання і отримання дайтаграм використовують встановлення з'єднання? Які функції в цьому випадку використовуються для передавання?
27. Як забезпечується звільнення ресурсів після завершення обміну інформацією без встановлення з'єднання?
28. Напишіть програму передавача, що не встановлює з'єднання.
29. Напишіть програму приймача, що не встановлює з'єднання.
30. Напишіть програму приймача-передавача без встановлення з'єднання, що може здійснювати обмін на одному комп'ютері, використовуючи функції *sendto* і *recvfrom*.
31. Напишіть програму приймача-передавача без встановлення з'єднання, що може здійснювати обмін на одному комп'ютері, використовуючи функції *send* і *recv*.
32. Які функції забезпечують незалежний від протоколу спосіб перетворення структури *sockaddr* у форматований рядок символів і навпаки? Опишіть їх параметри.
33. Які функції повертають адресу локального чи віддаленого комп'ютера? Опишіть їхні параметри.
34. Яка функція використовується для створення структури *WSAPROTOCOL_INFO*? Як у подальшому використовується дана структура?
35. Яка функція дозволяє швидко передавати дані з файла на інший комп'ютер? За рахунок чого підвищується швидкість передавання? Опишіть параметри даної функції.
36. Які характеристики мають сокети сім'ї *AF_INET*?

7 ОРГАНІЗАЦІЯ ВВЕДЕННЯ-ВИВЕДЕННЯ У СОКЕТАХ

7.1 Блокувальний режим введення

У блокувальному режимі введення функція *recv* блокує програму до отримання інформації з мережі. Проблема у тому, що функція *recv* може ніколи не завершитись, оскільки для цього потрібно зчитати якісь дані з буфера системи. Щоб запобігти такій ситуації потрібно створити окремо потік зчитування і потік обробки даних, які спільно використовують один буфер. Доступ кожного потоку до буфера встановлюється за допомогою таких об'єктів синхронізації, як критична секція (*critical section*) або мютекс (*mutex*). Потік зчитування зчитує дані, записує їх у буфер і встановлює подію. Після цього потік обробки обробляє дані. Наприклад:

Сервер

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include "windows.h"
#include "conio.h"
#include "stdio.h"
#include "string.h"
#pragma comment(lib, "ws2_32.lib")
#define PORT 5555
DWORD WINAPI FRead(LPVOID lpData);
DWORD WINAPI FRun(LPVOID lpData);
char buff[1024];
HANDLE evRead;
HANDLE hRun;
SOCKET Sock;
CRITICAL_SECTION cs;
int main(int argc, TCHAR* argv[])
{
    WSADATA WsaData;
    WSAStartup(0x0202,&WsaData);
    evRead=WSACreateEvent();
    InitializeCriticalSection(&cs);
    SOCKET ListenSock;
```

```
    sockaddr_in Addr;
    ListenSock=socket(AF_INET,SOCK_STREAM,0);
    Addr.sin_port=htons(PORT);
    Addr.sin_family=AF_INET;
    Addr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
    bind(ListenSock,(sockaddr*)&Addr,sizeof(Addr));
    listen(ListenSock,5);
    Sock=accept(ListenSock,NULL,NULL);
    ResetEvent(evRead);
    CreateThread(NULL,0,FRead,(LPVOID)Sock,0,NULL);
    hRun=CreateThread(NULL,0,FRun,NULL,0,NULL);
    _getch();
    closesocket(Sock);
    closesocket(ListenSock);
    return 0;
}

DWORD WINAPI FRead(LPVOID lpData)
{
    while(1)
    {
        SOCKET sock=(SOCKET)lpData;
        int n;
        EnterCriticalSection(&cs);
        n=recv(sock,buff,1024,0);
        if(n==0)
        {
            TerminateThread(hRun,0);
            ExitThread(0);
        }
        buff[n]=0;
        LeaveCriticalSection(&cs);
        SetEvent(evRead);
    }
    return 0;
}
DWORD WINAPI FRun(LPVOID lpData)
{
```

```
    while(1)
    {
        WaitForSingleObject(evRead,INFINITE);
        EnterCriticalSection(&cs);
        if(strlen(buff)>0)
            puts(buff);
        buff[0]=0;
        LeaveCriticalSection(&cs);
        ResetEvent(evRead);
    }
    return 0;
}
```

Клієнт

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include "windows.h"
#include "conio.h"
#pragma comment(lib, "ws2_32.lib")
#define PORT 5555
int main(int argc, CHAR* argv[])
{
    WSADATA WsaData;
    WSAStartup(0x0202,&WsaData);
    SOCKET Sock;
    sockaddr_in Addr;
    Sock=socket(AF_INET,SOCK_STREAM,0);
    Addr.sin_family=AF_INET;
    Addr.sin_port=htons(PORT);
    Addr.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
    connect(Sock,(sockaddr*)&Addr,sizeof(Addr));
    while(!_kbhit())
    {
        send(Sock,"Hello from client!",18,0);
        Sleep(10);
    }
    closesocket(Sock);
    _getch();
    return 0;
}
```

Для підтримки зв'язку через декілька сокетів в загальному випадку потрібно для кожного сокета окремо створити потік зчитування і потік обчислення. Тому недоліком цього варіанта є те, що він погано підтримує масштабування.

7.2 Неблокувальний режим введення

Неблокувальний режим складніший для програмування, ніж блокувальний, але він має переваги у тому, що не потрібно створювати у програмі окремий потік. Для цього потрібно після створення сокета перевести його у неблокувальний режим за допомогою виклику функції ***ioctlsocket***. Наприклад:

```
...
SOCKET sock;
unsigned long u1=1;
sock=socket(AF_INET,SOCK_STREAM,0);
if(ioctlsocket(sock,FIONBIO,&u1)==SOCKET_ERROR)
{
    printf("Error!");
    return;
}
...
```

Якщо сокет знаходиться у неблокувальному режимі, то функції *WSAAccept*, *accept*, *WSAConnect*, *connect*, *WSARecv*, *recv*, *WSARecvFrom*, *recvfrom*, *WSASend*, *send*, *WSASendTo*, *sendto*, *closesocket* завершуються одразу. У більшості випадків одразу після виклику вони повертають помилку *WSAEWOULDBLOCK*, яка означає, що функція не встигла виконатись під час виклику. Після першого виклику функції потрібно аналізувати код помилки і викликати цю функцію повторно. Однак циклічний виклик функцій, пов'язаних з обміном інформацією, є неефективним з погляду споживання ресурсів. Тому як у блокувальному, так і неблокувальному режимах ефективніше використовувати моделі введення-виведення сокетів.

7.3 Моделі введення-виведення сокетів

Існує п'ять моделей введення-виведення у *Winsock*: *select*, *WSAAsynchSelect*, *WSAEventSelect*, перекрите введення-виведення і порти завершення. Далі буде розглянуто перші три моделі.

7.3.1 Модель *select*

Це найпростіша модель введення-виведення у *Winsock*. Вона основана на використанні функції *select*, яка визначає, чи є у сокеті дані і чи можна туди записати нові. Функція *select* блокує введення-виведення до тих пір, доки не будуть виконані умови, задані у параметрах. Вона використовується для запобігання блокування функцій, пов'язаних з обміном інформацією, у синхронному режимі і для запобігання виникнення помилки *WSAEWOULDBLOCK* під час виклику цих функцій в асинхронному режимі. Функція ***select*** має вигляд:

```
int select(
    int nfds,
    fd_set *readfds,
    fd_set *writefds,
    fd_set *exceptfds,
    const struct timeval *timeout);
```

nfds – ігнорується (призначений для сумісності з сокетами Берклі);

readfds – набір сокетів, у яких є дані для читання;

writefds – набір сокетів, у яких є дані для записування;

exceptfds – набір сокетів, у яких є термінові дані.

timeval – вказівник на структуру timeval, що визначає час очікування завершення функції *select*. Якщо *timeval* = NULL, то функція буде чекати нескінченно довго.

Набір *readfds* містить сокети, що задовольняють одну з таких умов:

- дані доступні для читання;
- з'єднання закрите (*closed*), скинене (*canceled*), або завершене (*complited*);
- виклик функції *accept* після функції *listen* буде успішним.

Набір *writefds* містить сокети, що задовольняють одну з таких умов:

- дані готові для відправлення;
- вдалося встановити з'єднання у неблокувальному режимі.

Набір *exceptfds* містить сокети, що задовольняють одну з таких умов:

- не вдалося встановити з'єднання у неблокувальному режимі;
- *OOB*-дані доступні для читання.

Наприклад, для того, щоб перевірити можливість читання даних з сокета потрібно додати його до набору *readfds* і викликати функцію *select*. Після її завершення потрібно перевірити, чи входить ще цей сокет до

набору *readfds*. Якщо це так, то можна читати дані з сокета.

Під час виклику функції *select* хоча б один із наборів потрібно, щоб мав значення не *NULL*.

Структура ***timeval*** визначена так:

```
struct timeval
{
    long tv_sec;
    long tv_usec;
};
```

Поле *tv_sec* задає час чекання у секундах, а *tv_usec* – у мілісекундах. Таймаут {0,0} означає, що функція *select* має закінчуватись терміново, чого потрібно уникати. За успішного завершення функція повертає кількість описувачів сокетів, у яких є операції, що очікують введення-виведення. Якщо час *timeval* завершився, то у структурах повертається 0. Якщо функція *select* завершується невдачею, то вона повертає *SOCKET_ERROR*.

Перед викликом функціїї *select* потрібно створити одну чи декілька структур *fd_set*, обнулити їх і присвоїти їм описувачі сокетів, які потрібно перевірити. Після виклику функції потрібно перевірити, чи входять все ще ці сокети до набору. Це можна зробити за допомогою макросів:

*FD_ZERO(*set)* – обнуляє набір;

*FD_CLR(s, *set)* – вилучає сокет з набору;

*FD_SET(s,*set)* – додає сокет до набору;

*FD_ISSET(s,*set)* – повертає не 0, якщо сокет входить до набору.

Приклад сервера з використанням моделі *select* і клієнта за блокувального встановлення з'єднання:

Сервер

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include "windows.h"
#include "conio.h"
#include <stdio.h>
#pragma comment(lib, "ws2_32.lib")
#define PORT 5555
int main(int argc, TCHAR* argv[])
{
```

```
SOCKET Socket;
SOCKET ListenSock;
SOCKADDR_IN Addr;
WSADATA wsaData;
FD_SET WriteSet;
FD_SET ReadSet;
int n=0;
WSAStartup(0x0202,&wsaData);
ListenSock = socket(AF_INET,SOCK_STREAM,0);
Addr.sin_family = AF_INET;
Addr.sin_addr.s_addr = htonl(INADDR_ANY);
Addr.sin_port = htons(PORT);
bind(ListenSock, (PSOCKADDR) &Addr, sizeof(Addr));
listen(ListenSock, 5);
while(1)
{
    FD_ZERO(&ReadSet);
    FD_SET(ListenSock, &ReadSet);
    select(0, &ReadSet, NULL, NULL, NULL);
    if (FD_ISSET(ListenSock, &ReadSet))
    {
        printf("Client %d is connected.\n",n);
        Socket = accept(ListenSock, NULL, NULL);
    }
    FD_ZERO(&ReadSet);
    FD_SET(Socket, &ReadSet);
    select(0, &ReadSet, NULL, NULL, NULL);
    if (FD_ISSET(Socket, &ReadSet))
    {
        char buf[1024]="";
        int ret=recv(Socket,buf,1024,0);
        if(ret>0)
        {
            buf[ret]=0;
            printf("CLIENT %d: ",n);
            puts(buf);
```

```
                }
            }
            FD_ZERO(&WriteSet);
            FD_SET(Socket, &WriteSet);
            select(0, NULL, &WriteSet, NULL, NULL);
            if (FD_ISSET(Socket, &WriteSet))
        send(Socket,"Hello from server!",18,0);
            closesocket(Socket);
            printf("Client %d is disconnected.\n\n",n);
            n++;
        }
        _getch();
        return 0;
}
```

Клієнт

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include "winsock2.h"
#include "windows.h"
#include "conio.h"
#include "stdio.h"
#pragma comment(lib, "ws2_32.lib")
#define PORT 5555
int main(int argc, TCHAR* argv[])
{
    char buff[100];
    WSADATA Wsadata;
    WSAStartup(0x0202,&Wsadata);
    SOCKET Sock;
    Sock=socket(AF_INET,SOCK_STREAM,0);
    sockaddr_in ServerAddr;
    ServerAddr.sin_family=AF_INET;
    ServerAddr.sin_addr.S_un.S_addr=inet_addr("127.0.0.1");
    ServerAddr.sin_port=htons(PORT);
    printf("CLIENT: ");
    if(connect(Sock,(sockaddr*)&ServerAddr,sizeof(ServerAddr))==
SOCKET_ERROR)
    {
```

```
            printf("Error connection\n");
            _getch();
            return 1;
        }
        printf("Connect to server.\n");
        send(Sock,"Hello from client!",18,NULL);
        printf("SERVER: ");
        int retcod=recv(Sock,buff,100,NULL);
        buff[retcod]=0;
        puts(buff);
        closesocket(Sock);
        printf("Disconnect from server.\n");
        WSACleanup();
        _getch();
        return 0;
    }
```

7.3.2 Модель *WSAAsynchSelect*

Ця модель основана на повідомленнях *Windows*. Керування режимом введення-виведення у ній реалізовано за допомогою функції *WSAAsynchSelect*. Модель *WSAAsynchSelect* використовується у *Win32 API*-програмах, які обробляють повідомлення у віконній процедурі (*WndProc*). Крім того, вона реалізована також у програмах *MFC* для класу *CSocket*.

Функція ***WSAAsynchSelect*** має такий інтерфейс:

```
int WSAAsynchSelect (
SOCKET s,
HWND hWnd,
unsigned int nMsg,
long lEwent
);
```

Параметри функції такі: *s* – сокет; *hWnd* – описувач вікна, в яке буде відправлено повідомлення, коли відбудеться мережева подія; *nMsg* – повідомлення (зазвичай йому присвоюється номер *WM_USER*+1); *lEvent* – бітова маска, що задає комбінацію мережевих подій, повідомлення про які потрібно відслідковувати. цьому параметру можна присвоювати такі константи:

FD_ACCEPT – відслідковувати повідомлення про запит на з'єднання;

FD_CONNECT – відслідковувати повідомлення про встановлення з'єднання;

FD_READ – відслідковувати повідомлення про готовність даних для читання;

FD_WRITE – відслідковувати повідомлення про готовність даних для запису;

FD_CLOSE – відслідковувати повідомлення про закриття сокета;

FD_ADDRESS_LIST_CHANGE – відслідковувати повідомлення про зміну локальних адрес;

FD_OOB – відслідковувати повідомлення про отримання термінових даних;

FD_GROUP_QOS – відслідковувати повідомлення про зміну групового *QoS* (зарезервовано для використання у майбутньому);

FD_QOS – відслідковувати повідомлення про зміну *QoS*;

FD_ROUTING_INTERFACE_CHANGE – відслідковувати повідомлення про зміну інтерфейсу маршпутизації.

Об'єднання цих констант за допомогою порозрядного «АБО» дозволяє відслідковувати декілька повідомлень про події. Проте неможливо відслідковувати повідомлення про події, що відбуваються одночасно. Повідомлення про події будуть надсилатись, доки не буде закрито сокет або не викликано повторно функцію *WSAAsynchSelect* з іншими аргументами. Якщо під час виклику цієї функції параметра *lEwent* присвоїти 0, то ніякі повідомлення про події відслідковуватись не будуть.

У разі виклику функції *WSAAsynchSelect* сокет автоматично переходить у неблокувальний режим. Усі подальші функції викликаються під час надходження відповідних повідомлень у віконну процедуру ***WndProc***, яка має такий інтерфейс:

```
LRESULT CALLBACK WndProc(
HWND hWnd,
UINT nMsg,
WPARAM wParam,
LPARAM lParam);
```

Тут *hWnd* – описувач вікна, *nMsg* – повідомлення, *wParam* – сокет, *lParam* – складається з двох частин. Старша частина містить код помилки, а молодша – вигляд події, про яку надійшло повідомлення. Коли віконна процедура отримує повідомлення, то вона має насамперед перевірити, чи немає помилки. Для цього використовується макрос

WSAGETSELECTEDERROR(lParam), який повертає код помилки. Якщо помилки немає, то для отримання коду події викликається макрос *WSAGETSELECTEDEVENT(lParam)*. Далі наведено приклад сервера, створеного як *Win32 API*-додаток з використанням моделі введення-виведення *WSAAsynchSelect*, який взаємодіє з простим клієнтом, створеним раніше у консольній програмі для моделі *select*. Сервер приймає від клієнта запит на з'єднання, потім отримує від нього повідомлення і надсилає відповідь.

```
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#define WM_SOCKET WM_USER+1
#define PORT 55555
#include <winsock2.h>
#include "windows.h"
#pragma comment(lib, "ws2_32.lib")
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam,LPARAM lParam);
SOCKET ClientSock;
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE, LPSTR, int)
{
    TCHAR szTitle[]="WSAAsyncSelect Server"; // текст рядка заголовка
    TCHAR szWindowName[]="MainWindow";// ім'я класу головного вікна
    MSG msg;
    HWND hWnd;
    WNDCLASS wndClass;
    ZeroMemory(&wndClass,sizeof(wndClass));
    wndClass.lpfnWndProc=WndProc;
    wndClass.hInstance=hInstance;
    wndClass.hCursor=LoadCursor(NULL,IDC_ARROW);
    wndClass.lpszClassName=szWindowName;
    wndClass.hbrBackground=(HBRUSH)(COLOR_WINDOW+1);
    RegisterClass(&wndClass);
    hWnd=CreateWindow(szWindowName, szTitle, WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
    ShowWindow(hWnd,SW_SHOWNORMAL);
    WSADATA WsaData;
    WSAStartup(0x0202,&WsaData);
    SOCKET ListenSock;
    ListenSock=socket(AF_INET,SOCK_STREAM,0);
```

```
        sockaddr_in ListenAddr;
        ListenAddr.sin_family=AF_INET;
        ListenAddr.sin_port=htons(PORT);
        ListenAddr.sin_addr.S_un.S_addr=htonl(INADDR_ANY);
        bind(ListenSock,(sockaddr*)&ListenAddr,sizeof(ListenAddr));
        listen(ListenSock,5);
        WSAAsyncSelect(ListenSock,hWnd,WM_SOCKET,FD_ACCEPT|FD_CLOSE);
        while(GetMessage(&msg,NULL,0,0))
            DispatchMessage(&msg);
    }
    LRESULT CALLBACK WndProc(HWND hWnd, UINT msg, WPARAM wParam,
LPARAM lParam)
    {
        HDC hdc;
        PAINTSTRUCT pc;
        static sockaddr_in ClientAddr;
        char str1[100];
        char str2[100];
        static char str3[100]="";
        int retcod=0;
        int AddrLen=sizeof(ClientAddr);
        switch(msg)
        {
        case WM_DESTROY:
            WSACleanup();
            PostQuitMessage(0);
            return 0;
        case WM_PAINT:
            RECT r;
            hdc=BeginPaint(hWnd,&pc);
            GetClientRect(hWnd,&r);
            DrawText(hdc,str3,-1,&r,NULL);
            EndPaint(hWnd,&pc);
            break;
        case WM_SOCKET:
            switch(WSAGETSELECTEVENT(lParam))
            {
            case FD_ACCEPT:
                ZeroMemory(&ClientAddr,sizeof(ClientAddr));
```

```
            ClientSock=accept(wParam,(sockaddr*)&ClientAddr,&AddrLen);
                strcat(str3,"SERVER: Client connected.\n\r");
                InvalidateRect(hWnd,NULL,true);
                WSAAsyncSelect(ClientSock,hWnd,WM_SOCKET,FD_READ);
                break;
            case FD_READ:
                retcod=recv(ClientSock,str1,sizeof(str1),NULL);
                str1[retcod]=0;
                wsprintf(str2,"CLIENT: %s\n\r",str1);
                strcat(str3,str2);
                InvalidateRect(hWnd,NULL,true);
                WSAAsyncSelect(ClientSock,hWnd,WM_SOCKET,FD_WRITE);
                break;
            case FD_WRITE:
                send(ClientSock,"Hello from server!",18,NULL);
                WSAAsyncSelect(ClientSock,hWnd,WM_SOCKET, FD_CLOSE);
                break;
            case FD_CLOSE:
                closesocket(wParam);
                strcat(str3,"SERVER: Client disconnected.\n\r\n\r");
                InvalidateRect(hWnd,NULL,true);
            }
            break;
        default:
            return DefWindowProc(hWnd,msg,wParam,lParam);
        }
        return 0;
    }
```

7.3.3 Модель *WSAEventSelect*

Ця модель схожа з моделлю *WSAAsynchSelect*. Відмінність у тому, що повідомлення відправляються не процедурі вікна, а програмній події. Для цього потрібно у програмі створити об'єкт події для кожного сокета за допомогою функції ***WSACreateEvent***, що має такий інтерфейс:

WSAEVENT WSACreateEvent().

Функція створює об'єкт програмної події і повертає її описувач. Цей описувач потрібно пов'язати з сокетом і задати типи повідомлень про мережеві події, що будуть відслідковуватись (використовуються ті самі константи, що і для моделі *WSAAsynchSelect*). Це робиться за допомогою

функції ***WSAEventSelect***, що має такий інтерфейс:

int WSAEventSelect(
SOCKET s,
WSAEVENT hEventObject,
long hNetworkEvent).

Тут *s* – сокет; *hEventObject* – описувач програмної події; *hNetworkEvent* – константа, що задає бітову маску повідомлення про мережеву подію.

У програмної події є два стани: вільний (*signaled*) і зайнятий (*nonsignaled*), а також два режими скидання (*reset*): ручний (manual) і автоматичний (*auto*). За замовчуванням об'єкт програмної події створюється у зайнятому стані і у режимі ручного скидання. Коли на сокеті відбувається мережева подія, то зв'язана з ним програмна подія стає вільною. Програма після обробки повідомлення має скинути цю програмну подію у зайнятий стан за допомогою функції ***WSAResetEvent*** з описувачем програмної події як параметра:

BOOL WSAEventReset(WSAEVENT hEvent).

Після виклику функції *WSAEventSelect* програма має очікувати повідомлення про мережеву подію або закінчення встановленого часу за допомогою функції ***WSAWaitForMultipleEvent***:

DWORD WSAWaitForMultipleEvent(
DWORD cEvents,
*const WSAEVENT *lphEvents,*
BOOL fWaitAll,
DWORD dwTimeout,
BOOL fAlertable).

Тут *lphEvents* – вказівник на масив описувачів програмних подій, *cEvents* – кількість елементів цього масиву (їх має бути не більше 64 для одного потоку). *fWaitAll* – визначає чи чекати звільнення всіх програмних подій, чи лише будь-якої однієї. В другому випадку функція повертає константу, що відповідає звільненій програмній події. Як правило, цьому параметру присвоюється значення *FALSE*. *dwTimeout* – час очікування у мілісекундах, також може дорівнювати 0 (не рекомендується) або *WSA_INFINITE* – нескінченний час. *fAlertable* – у цій моделі параметру присвоюється значення *FALSE*.

Функція *WSAWaitForMultipleEvent* повертає значення, за допомогою якого можна визначити індекс в масиві описувачів *lphEvents*. Це дозволяє знайти відповідну програмну подію і зв'язаний з нею сокет, на якому відбулися якісь із заданих мережевих подій:

```
int Index=WSAWaitForMultipleEvents(...);
MyEvent=EventArray[Index-WSA_WAIT_EVENT_0];
```

Вияснивши сокет, на якому відбулася програмна подія, можна визначити мережеві події, що її викликали, за допомогою функції *WSAEnumNetworkEvents*:

```
int WSAEnumNetworkEvents(
SOCKET s,
WSAEVENT hEventObject,
LPWSANETWORKEVENTS lpNetworkEvents).
```

Тут *s* – сокет, на якому відбулась програмна подія, *hEventObject* – описувач об'єкта події, *lpNetworkEvents* – вказівник на структуру *WSANETWORKEVENTS*, в яку записується мережева подія. Така структура має такий вигляд:

```
struct WSANETWORKEVENTS
{
    long lNetworkEvents;
    int iErrorCode[FD_MAX_EVENTS];
};
```

Тут *lNetworkEvents* – змінна, що містить види мережевих подій; *iErrorCode* – масив кодів помилок, викликаних мережевими подіями. Для кожної мережевої події існує індекс у масиві помилок, який задається іменованою константою з тим самим іменем, що і мережева подія, але з суфіксом *_BIT*. Наприклад, для мережевої події *FD_READ* ім'я індексу в масиві кодів помилок буде *FD_READ_ BIT*. Далі наведено приклад аналізу коду помилки для події *FD_READ*:

```
if(NetEvents.lNetworkEvents & FD_READ)
{
    if(NetEvents.iErrorCode[FD_READ_BIT]!=0)
    {
        printf("Erro FD_READ: %d",NetEvents.iErrorCode[
        FD_READ_BIT]);
    }
}
```

Закінчивши роботу з програмною подією потрібно її закрити за допомогою функції *WSACloseEvent*:

```
BOOL WSACloseEvent(WSAEVENT hEvent).
```

Приклад сервера з використанням моделі введення-виведення *WSAEventSelect*.

```
//Ім'я модуля: Server.cpp
#define _WINSOCK_DEPRECATED_NO_WARNINGS
#include <winsock2.h>
#include "windows.h"
#include "stdio.h"
#pragma comment(lib, "ws2_32.lib")
#define PORT 5555

int _tmain(int argc, _TCHAR* argv[])
{
    sockaddr_in addr;
    WSADATA WsaData;
    int Total=1,Index;
    SOCKET Socket[64];
    WSAEVENT Event[64];
    WSAStartup(0x0202,&WsaData);
    Socket[0]=socket(AF_INET, SOCK_STREAM,0);
    addr.sin_family=AF_INET;
    addr.sin_port=htons(PORT);
    addr.sin_addr.S_un.S_addr=htons(INADDR_ANY);
    bind(Socket[0],(sockaddr*)&addr, sizeof(addr));
    Event[0]=WSACreateEvent();
    listen(Socket[0], 5);
    WSAEventSelect(Socket[0], Event[0], FD_ACCEPT | FD_CLOSE);
    WSANETWORKEVENTS NetworkEvents;
    while(true)
    {
        if(Total>63)
            break;
        Index=WSAWaitForMultipleEvents(Total,Event,false, WSA_INFINITE,
    false);
        WSAEnumNetworkEvents(Socket[Index-WSA_WAIT_EVENT_0],
    Event[Index-WSA_WAIT_EVENT_0], &NetworkEvents);
        if(NetworkEvents.lNetworkEvents & FD_ACCEPT &&
    NetworkEvents.iErrorCode[FD_ACCEPT_BIT]==0)
        {
            printf("Client connected!\n");
            Socket[Total]=accept(Socket[0], NULL,NULL);
```

```
            Event[Total]=WSACreateEvent();
            WSAEventSelect(Socket[Total], Event[Total], FD_READ | FD_WRITE |
        FD_CLOSE);
            Total++;
        }
        if(NetworkEvents.lNetworkEvents & FD_READ &&
    NetworkEvents.iErrorCode[FD_READ_BIT]==0)
        {
            char buf[1024];
            int nRecv=recv(Socket[Index-WSA_WAIT_EVENT_0], buf, 1024,0);
            buf[nRecv]=0;
            puts(buf);
        }
        if(NetworkEvents.lNetworkEvents & FD_WRITE &&
    NetworkEvents.iErrorCode[FD_WRITE_BIT]==0)
            send(Socket[Index-WSA_WAIT_EVENT_0], "Hello from server!", 18,
        0);
        if(NetworkEvents.lNetworkEvents & FD_CLOSE &&
    NetworkEvents.iErrorCode[FD_CLOSE_BIT]==0)
        {
            printf("Client closed connection.\n");
            int n=Index-WSA_WAIT_EVENT_0;
            closesocket(Socket[n]);
            WSACloseEvent(Event[n]);
            for(int i=n;i<Total;i++)
            {
                Socket[i]=Socket[i+1];
                Event[i]=Event[i+1];
            }
            Total--;
        }
    }
    return 0;
}
```

Для встановлення з'єднання й обміну інформацією з цим сервером можна використати програму клієнта з пункту 7.3.1.

7.4 Контрольні питання

1. Надати інформацію, що таке блокувальний режим введення-виведення. Які проблеми виникають у цьому режимі та які існують варіанти їх вирішення?
2. Навести приклад програми сервера, який працює у блокувальному режимі введення-виведення з використанням потоків.
3. Надати інформацію, що таке неблокувальний режим введення-виведення. Які переваги і недоліки цього режиму порівняно з блокувальним режимом? Як перевести сокет у неблокувальний режим?
4. Описати, які існують моделі введення-виведення у сокетах. Надати порівняльну характеристику таких моделей.
5. Описати особливості моделі введення-виведення *select*. Описати призначення, сигнатуру і параметри функції *select*.
6. Навести приклад програми сервера, який використовує модель введення-виведення *select*.
7. Описати особливості моделі введення-виведення *WSAAsynchSelect*. У яких видах застосувань використовується ця модель? Описати призначення, сигнатуру і параметри функції *WSAAsynchSelect*.
8. Навести імена і призначення констант, за допомогою яких можна відслідковувати повідомлення про мережеві події у функції *WSAAsynchSelect*. У який режим переходить сокет після виклику функції *WSAAsynchSelect*?
9. Навести приклад програми сервера, який використовує модель введення-виведення *WSAAsynchSelect*.
10. Описати особливості моделі введення-виведення *WSAEventSelect*. У яких видах застосувань використовується ця модель? Описати призначення, сигнатуру і параметри функції *WSAEventSelect*.
11. Описати, які стани використовуються у програмній моделі *WSAEventSelect*.
12. Описати, як організована обробка подій у програмній моделі *WSAEventSelect*.
13. Описати призначення, сигнатуру і параметри функції *WSAEnumNetworkEvents*.
14. Навести приклад сервера з використанням моделі введення-виведення *WSAEventSelect*.

ПІСЛЯМОВА

У цьому посібнику описано широкий спектр технологій прикладного мережевого програмування в операційній системі *Windows*. Розглянуто перенаправлювач, поштові скриньки, іменовані канали та інтерфейс *NetBIOS*. Найбільшу увагу приділено інтерфейсу мережевого програмування *Winsock*.

Надано інформацію про перенаправлювач *Windows*, який дозволяє програмам одержувати доступ мережею до ресурсів файлової системи *Windows*. Описано, як за допомогою перенаправлювача здійснюється обмін інформацією у мережі і як в цьому випадку використовується система безпеки *Windows*.

Розглянуто принципи мережевого програмування з використанням поштових скриньок. Розглянуто переваги і недоліки такого підходу.

Детально розглянуто використання технології іменованих каналів для мережевого програмування. Технологія іменованих каналів надає просту архітектуру «клієнт-сервер» для надійного передавання даних з використанням перенаправлювача *Windows*. Основна перевага іменованих каналів полягає у тому, що вони дозволяють скористатися вбудованими можливостями захисту операційної системи *Windows*.

Розглянуто основи *NetBIOS* – потужного прикладного інтерфейсу мережевого програмування. Одна з його сильних сторін – незалежність від протоколу: програми можуть працювати через протоколи *TCP/IP*, *NetBEUI*, *IPX/SPX* та інші. *NetBIOS* дозволяє здійснювати обмін як із встановленням логічного з'єднання, так і без нього. Значна перевага *NetBIOS* перед сокетами – єдиний спосіб вирішення і реєстрації імен. Програма *NetBIOS* потребує тільки імені *NetBIOS*, тоді як програма *Winsock*, реалізуючи різні протоколи, має знати схему адресації кожного з них.

Найбільшу частину посібника присвячено популярній технології мережевого програмування – сокетам. У рамках створення *WinAPI*-програм розглянуто технологію *Winsock*. *Winsock* – це інтерфейс прикладного програмування мережевих взаємодій, реалізований на всіх платформах *Windows*. Інтерфейс *Winsock* розроблено на основі *Berceley* (*BSD*) *Sockets* на платформах *UNIX*, що працює з багатьма мережевими протоколами. У середовищі *Windows* цей інтерфейс став повністю незалежним від мережевих протоколів. У розділі, присвяченому сокетам, описано характеристики протоколів, що підтримуються *Windows* і, зі свого боку,

підтримують інтерфейс *Winsock*. Описано режими сокетів, алгоритм встановлення з'єднання та функції, необхідні для встановлення з'єднання і виконання обміну. Під час написання цього посібника використано матеріали з [1, 2].

Microsoft продовжує розвивати технології мережевого програмування. У рамках програмного середовища *Visual C++* запропоновано класи сокетів *MFC*. Бібліотека *MFC* є об'єктно-орієнтованою надбудовою над *WinAPI*, призначеною для спрощення програмування за рахунок використання класів, що інкапсулюють функції більш високого рівня. У *MFC* сокет – це об'єкт відповідного класу. Існує два класи сокетів *MFC*: *CAsyncSocket* і похідний від нього *CSocket*. Клас *CAsyncSocket* містить функції, що виконують обмін інформацією між комп'ютерами у асинхронному режимі, а клас *CSocket* призначений для обміну у синхронному режимі. Детальнішу інформацію про технологію програмування за допомогою сокетів *MFC* можна отримати у [3, 4].

Програмування за допомогою сокетів в *UNIX*-подібних системах описано у [5]. Для професійного програмування мережевого обміну інформацією потрібно більш детальне вивчення будови мереж. Для цього можна порекомендувати [3].

Останнім часом широкої популярності набула нова платформа програмування *Microsoft*, що отримала назву "*.NET*" (*DOT NET*). У рамках цієї платформи спеціально для мережевого і розподіленого програмування розроблено мову *C#*, що є модифікацією мови *C++*. Тому для вивчення *C#* найкраще мати досвід з програмування на *C++* [3]. Тим, хто знайомий з *C++*, для вивчення мови *C#* можна порекомендувати [6]. У цій книзі описано розробку програм мовою *C#* для платформи *DOT NET*, а також аспекти мережевого програмування. Для більш детального вивчення технології програмування у мережах з використанням сокетів та інших засобів платформи *DOT NET* можна порекомендувати [7, 8].

Цей посібник є основним для вивчення дисципліни «Прикладне програмування у комп'ютерних мережах». Проте вивчення цієї дисципліни потребує наявності базових знань як з програмного, так і з апаратного забезпечення засобів сучасної цифрової техніки. Для отримання базової підготовки існує велика кількість широко відомих літературних джерел. Для отримання знань з передових досягнень у галузі побудови засобів цифрової техніки можна запропонувати наукові роботи [9, 10, 11].

БІБЛІОГРАФІЧНИЙ ОПИС

1. Jones A. Network programming for Microsoft Windows. Second edition / A. Jones, J. Ohlund – Redmond : Microsoft Press, 2002. – 608 p.

2. Авраменко В. В. Програмування наVisual C++ із застосуванням бібліотеки MFC : навчальний посібник / В. В. Авраменко, А. М. Скаковська. – Суми : Сумський державний університет, 2015. – 215 с.

3. Комп'ютерні мережі : навчальний посібник / Азаров О. Д. та ін. – Вінниця: ВНТУ, 2013. – 500 с.

4. Бєлов Ю. А. Вступ до програмування мовою C++. Організація обчислень / Ю. А. Бєлов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К. : Видавнчо-поліграфічний центр "Київський університет", 2012. – 175 с.

5. Snader J. Effective TC/IP Programming / J. Snader : Addison-Wesley Proffesional, 2000. – 320 p.

6. Troelsen A. Pro C# 9 with .NET 5. Faundational principles and practices in programming. Tenth edition / A. Troelsen, P. Japikse : Apress Media, LLC, 2021. – 770 p.

7. Crowczuk A. Professional .NET network programming / A. Crowczuk, V. Kumar, N. Laghari and others : Wrox Press Ltd, 2007. – 400 p.

8. Löwy J. Programming WCF services. / J. Löwy. : O`Reilly, 2008. – 592 p.

9. Азаров О. Д. Повнофункціональна побітова потокова арифметика зі зменшеними витратами обладнання. : монографія / О. Д. Азаров, О. І. Черняк. – Вінниця : ВНТУ, 2013. – 200 с.

10. Азаров О. Д. Структурна організація побітового додавання і віднімання кодів золотої 1-пропорції із врахуванням знаків / О. Д. Азаров, О. І. Черняк // Інформаційні технології та комп'ютерна інженерія. Вінницький національний технічний університет. – 2011. – № 3(22). С. 13–16.

11. Азаров О. Д. Метод побудови швидкодіючих фібоначчієвих лічильників / О. Д. Азаров, О. І. Черняк // Проблеми інформатизації та управління – 2014. – № 2(46). – С 5–8.

ГЛОСАРІЙ

анонімний – *anonymous*
архітектура широкомовлення – *broadcasting architecture*
блок мережевого керування – *network control block, NCB*
блок повідомлення сервера – *server message block, SMB*
віддалені пристрої – *removed devices*
відкрите з'єднання систем – *open systems interconnect, OSI*
вільний стан – *nonsignaled state*
глобальна мережа – *wide area networks, WAN*
гніздо Windows – *Windows socket*
дескриптори безпеки – *security descriptors*
доменна система імен – *domain name system, DNS*
зайнятий стан – *signaled state*
запис керування доступом – *access control entities, ACE*
запит введення-виведення – *input-output query*
захист від несанкціонованого доступу – *access protection*
зв'язок між процесами – *interprocess communication*
ідентифікатор безпеки – *security identifier, SID*
ідентифікація – *identification*
іменовані канали – *named pipes*
інтерфейс прикладного програмування – *application programming interface, API*
клієнт для мереж Microsoft – *client for microsoft networks*
комп'ютерна мережа – *computer network*
комплексне введення-виведення – *scatter-gather I/O*
консольні програми – *console applications*
локальна адреса – *local address*
максимальний час життя сегмента – *maximum segment lifetime*
маркер доступу – *access tokens*
масив, що завершується нулем, – *null-terminated array*
мережева операційна система – *network operating system, NOS*
мережеве програмування – *network programming*
мережевий порядок проходження байтів – *network-byte order*
мережевий постачальник – *network provider*
мережевий протокол – *network protocol*
мережевий транспортний протокол – *network transport protocol*
мережеві програми – *network applications*

номер мережевого адаптера – *local area network adapter number, LANA*
односторонній обмін – *one-directional exchange*
операційна система – *operation system*
основний транспортний протокол дайтаграмного обміну даними у мережі – *user datagram protocol,UDP*
основний транспортний протокол потокового обміну даними у мережі – *transmission control protocol, TCP*
основні реквізити входу – *primary login*
перекрите введення-виведення – *overlapped I/O*
перенаправлення введення-виведення – *I/O redirection*
перенаправлювач – *redirector*
персоналізація – *impersonation*
повернуте повне ім'я домену – *fully qualified domain name, FQDN*
повідомлення, дайтаграма – *datagram*
порядок від молодшого до старшого байта – *little-endian*
порядок від старшого байта до молодшого – *big-endian*
постачальник декількох *UNC* – *multiple unc provider,MUP*
потік байтів – *byte stream*
потік повідомлень – *message stream*
поштові скриньки – *mailslots*
протокол спільного використання файлів на основі блоків повідомлень сервера – *server message block file sharing protocol*
реквізити сеансу – *session login*
системний порядок – *host-byte-order*
системний список керування доступом – *system access control list, SACL*
список вибіркового керування доступом – *discretionary access control list, DACL*
термінові дані – *out-of-band, OOB*
файлова система – *file system*
файлова система іменованих каналів – *named pipe file system, NPFS*
фрагментарне повідомлення – *partial message*
час життя – *time-to-live, TTL*
якість обслуговування – *quality of service, QoS*

Навчальне електронне видання
комбінованого використання.
Можна використовувати в локальному та мережному режимах

Олексій Дмитрович Азаров
Олександр Іванович Черняк
Людмила Анатоліївна Савицька

ПРИКЛАДНЕ ПРОГРАМУВАННЯ У КОМП'ЮТЕРНИХ МЕРЕЖАХ

Навчальний посібник

Видання друге, доповнене і перероблене

Рукопис оформив *О. Черняк*

Редактор *Т. Старічек*

Оригінал-макет підготувала *Т. Старічек*

Підписано до видання 23.10.2023 р.
Гарнітура Times New Roman.
Зам. № Р2023-125.

Видавець та виготовлювач
Вінницький національний технічний університет,
Редакційно-видавничий відділ.
ВНТУ, ГНК, к. 114.
Хмельницьке шосе, 95, м. Вінниця, 21021.
Тел. (0432) 65-18-06.
press.vntu.edu.ua;
E-mail: irvc.ed.vntu@gmail.com.
Свідоцтво суб'єкта видавничої справи
серія ДК № 3516 від 01.07.2009 р.