**Конспект лекцій. Програмування комп’ютерних систем і ятерів**

Оглавление

**Лекція 1** [Мережі і мережеві протоколи 2](#_Toc372625450)

**Лекція 2** [Робота з ресурсами мережевого оточення 8](#_Toc372625451)

**Лекція 3** [Структура мережі 11](#_Toc372625452)

**Лекція 4** [Робота з мережею за допомогою об'єктів Visual C++ 19](#_Toc372625453)

**Лекція 5** [Робота безпосередньо з WinSock 24](#_Toc372625454)

**Лекція 6** [Обробка даних, що приймаються 39](#_Toc372625455)

**Лекція 7** [Прийом і передача даних 41](#_Toc372625456)

**Лекція 8** [Робота з СОМ портом 54](#_Toc372625457)

# Мережі і мережеві протоколи

 Перш ніж я покажу перший приклад, доведеться трохи зайнятися теорією. Це не займе багато часу, але потім нам буде легше розуміти один одного. Для кращого сприйняття матеріалу цього розділу вам бажано знати основи мереж і протоколів.

 Кожного разу, коли ви передаєте дані по мережі, вони якось перетікають від вашого комп'ютера до сервера або іншого комп'ютера. Як це відбувається? Ви, напевно, скажете, що за допомогою спеціального мережевого протоколу, і будете мають рацію. Але існує безліч різновидів протоколів.

 Який і коли використовується? Навіщо вони потрібні? Як вони працюють? Ось на ці питання я зараз постараюся дати відповідь.

 Перш ніж розбиратися з протоколами, нам необхідно дізнатися, що таке модель взаємодії відкритих систем (OSI - OpenSystemsInterconnection), яка була розроблена Міжнародною організацією за стандартами (ISO - InternationalOrganizationforStandardization). Відповідно до цієї моделі, мережева взаємодія ділиться на сім рівнів.

 *1. Фізичний уровень-* передача бітів по фізичних каналах (коаксіальний кабель, вита пара, оптоволоконний кабель). Тут визначаються характеристики фізичних середовищ і параметри електричних сигналів.

 *2. Канальний рівень* - передача кадру даних між будь-якими вузлами в мережах типової топології або сусідніми вузлами в мережах довільної топології. Як адреси на канальному рівні використовуються МАС-адреси.

 *3. Мережевий рівень* - доставка пакету будь-якому вузлу в мережах довільної топології. На цьому рівні немає ніяких гарантій доставки пакету.

 *4. Транспортний рівень* - доставка пакету будь-якому вузлу з будь-якою топологією мережі і заданим рівнем надійності доставки. На цьому рівні є засобу для встановлення з'єднання, буферизації, нумерації і впорядковування пакетів.

 *5. Рівень сеансу* - управління діалогом між вузлами. Забезпечена можливість фіксації активної на даний момент сторони.

 *6. Рівень уявлення* - надається можливість перетворення даних (шифрування, стиснення).

 *7. Прикладний рівень* - набір мережевих сервісів (FTP, E-mail і ін.) для користувача і додатку.

 Якщо ви уважно прочитали про всі рівні, то, напевно, відмітили, що перші три рівні забезпечуються устаткуванням, таким як мережеві карти, маршрутизатори, концентратори, мости і ін. Останні три - операційною системою або додатком. Четвертий рівень є проміжним.

 Як працює протокол по цій моделі? Все починається з прикладного рівня. Пакет потрапляє на цей рівень, і до нього додається заголовок. Після цього прикладний рівень відправляє цей пакет на наступний рівень (рівень уявлення). Тут йому також додається свій власний заголовок, і пакет відправляється далі. І так до фізичного рівня, який займається безпосередньо передачею даних і відправляє пакет в мережу.

 Інша машина, одержавши пакет, починає зворотний відлік. Пакет з фізичного рівня потрапляє на канальний. Канальний рівень прибирає свій заголовок і піднімає пакет вище (на рівень мережі). Рівень мережі прибирає свій заголовок і піднімає пакет вище. Так пакет піднімається до рівня додатку, де залишається чистий пакет без службової інформації, яка була прикріплена на початковому комп'ютері перед відправкою пакету.

 Передача даних не обов'язково повинна починатися з сьомого рівня. Якщо використовуваний протокол працює на четвертому рівні, то процес передачі почнеться з нього, і пакет підніматиметься вгору до фізичного рівня для відправки. Кількість рівнів в протоколі визначає його потреби і можливості при передачі даних.

 Чим нижче знаходиться протокол (ближче до прикладного рівня), тим більше у нього можливостей і більше накладних витрат при передачі даних (довше і складніше заголовок). Протоколи, що розглядаються в даній книзі, знаходитимуться на різних рівнях, тому матимуть різні можливості.

 Корпорація Microsoft реалізувала протокол TCP/IP в моделі OSI по-своєму (з невеликими відхиленнями від стандарту). Я розумію, що модель OSI довідкова, і призначена тільки як рекомендація, але не можна ж було так її змінювати, адже принцип залишений той же, хоча змінилися назви і кількість рівнів.

 У MS TCP/IP замість семи рівнів є тільки чотири. Але це не означає, що решта рівнів забута і позакидана, просто один рівень може виконувати все, що в OSI роблять три рівні. Наприклад, рівень додатку у Microsoft виконує все, що роблять рівень додатки, рівень уявлення і рівень сеансу, разом узяті.

 **Мережеві протоколи**

 Перш ніж починати писати мережеві програми, необхідно розібратися з мережевими протоколами, зрозуміти основу і принципи їх роботи. У цьому розділі я зупинюся на найважливіших моментах, які необхідно знати програмісту для правильного ухвалення рішення. Ви побачите основні відмінності і зможете зрозуміти, що не можна просто узяти перший протокол, що попався, і написати з його допомогою будь-яку програму. Іноді вибір буває дуже складним, але від нього залежить майбутнє програми.

 **Протокол IP**

 Якщо подивитися на схему мережевої моделі, то можна побачити, що протокол IP знаходиться на мережевому рівні. З цього можна зробити висновок, що IP виконує мережеві функції - доставка пакету будь-якому вузлу в мережах довільної топології.

 Протокол IP при передачі даних не встановлює віртуального з'єднання і використовує датаграми (пакети даних) для відправки інформації від одного комп'ютера до іншого. Це означає, що по протоколу IP пакети просто відправляються в мережу без очікування підтвердження про отримання даних (АСЬК Acknowledgment), а значить, без гарантії доставки пакетів і, відповідно, без гарантії цілісності даних. Якщо хоч би один пакет з 100 необхідних не дійде до адресата, то дані порушаться, і зібрати їх в єдине ціле буде неможливо.

 Всі необхідні дії по підтвердженню і забезпеченню цілісності даних повинні забезпечувати протоколи, що працюють на більш високому рівні.

 Кожен IP-пакет містить адреси відправника і одержувача, ідентифікатор протоколу, TTL (час життя пакету) і контрольну суму для перевірки цілісності пакету. Як бачите, тут є контрольна сума, яка все ж таки дозволяє дізнатися цілісність пакету. Але про це дізнається тільки одержувач. Коли комп'ютер-одержувач прийняв пакет, то він перевіряє контрольну суму тільки для себе. Якщо сума сходиться, то пакет обробляється, інакше просто ігнорується. А комп'ютер-відправник не зможе дізнатися про помилку, що виникла в пакеті, і повторити посилку. Саме тому з'єднання по протоколу IP не можна вважати надійним.

 **Зіставлення адреси ARP і RARP**

 Протокол ARP (AddressResolutionProtocol, протокол визначення адреси) призначений для визначення апаратної (MAC) адреси комп'ютера в мережі поєго IP-адресі. Перш ніж дані зможуть бути послані на який-небудь комп'ютер, відправник повинен знати апаратну адресу одержувача. Саме для цього і призначений ARP.

 Коли комп'ютер посилає ARP-запит на пошук апаратної адреси, то протокол спочатку шукає цю адресу в локальному кеші. Якщо вже були звернення по даній IP-адресі, то інформація про МАС-адресу повинна зберегтися в кеші. Якщо нічого не знайдено, то в мережу посилається широкомовний запит, який одержать всі комп'ютери мережі. Вони одержать цей пакет і перевірять адресу. Той, кому належить шуканий IP, відповість на запит, вказавши свою МАС-адресу. Оскільки ця адреса повинна бути унікальною (прошивається в мережевому пристрої на заводі-виготівнику), то і відповідь повинна бути один. Але ви повинні враховувати, що є засоби підробки МАС-адрес (хакери іноді використовують цей прийом в своїх цілях), і може виникнути ситуація, коли відповідь прийде від двох машин.

 Протокол RARP (ReversAddressResolutionProtocol, зворотний протокол визначення адреси) визначає IP-адресу по відомій МАС-адресі. Процес пошуку адрес абсолютно такий же.

 **Транспортні протоколи**

 На транспортному рівні ми маємо два протоколи: UDP і TCP. Обидва вони працюють поверх IP. Це означає, що, коли пакет TCP або UDP опускається на рівень нижче для відправки в мережу, він потрапляє на рівень мережі прямо в лапи протоколу IP. Тут пакету додається мережева адреса, TTL і інші атрибути протоколу IP. Після цього пакет йде далі вниз для фізичної відправки в мережу. Голий пакет TCP не може бути відправлений до мережі, тому що він не має інформації про одержувача, ця інформація додається до пакету з IP-заголовком на рівні мережі.

 Давайте тепер розглянемо кожен протокол окремо.

 **Швидкий UDP**

 Як і IP, протокол UDP для передачі даних не встановлює з'єднання з сервером. Дані просто викидаються в мережу, і протокол навіть не піклується про доставку пакету. Якщо дані на шляху до сервера зіпсуються або взагалі не дійдуть, то відправляюча сторона про це не дізнається. Отже, по цьому протоколу, як і по голому IP, не бажано передавати дуже важливі дані.

 Завдяки тому, що протокол UDP не встановлює з'єднання, він працює дуже швидко (у декілька разів швидше за TCP, про яке трохи нижче). Із-за високої швидкості його дуже зручно використовувати там, де не потрібно піклуватися про цілісність даних. Таким прикладом можуть служити радіостанції в Інтернеті. Звукові дані просто випліскуються в глобальну мережу, і якщо слухач не одержить одного пакету, то максимум, що він відмітить - невелике заїкання в місці втрати. Але якщо врахувати, що мережеві пакети мають невеликий розмір, то ця затримка буде практично непомітна.

 Велика швидкість - великі проблеми з безпекою. Оскільки немає з'єднання між сервером і клієнтом, то немає ніякої гарантії в достовірності даних. Протокол UDP більше схильний спуфінгу (spoofing, підміна адреси відправника), тому побудова на ньому захищених мереж утруднена.

 Отже, UDP дуже швидкий, але його можна використовувати тільки там, де дані не мають високої цінності (можлива втрата окремих пакетів) і не секретні (UDP більше схильний до злому).

 **Повільний, але надійний TCP**

 Як я вже сказав, протокол TCP лежить на одному рівні з UDP і працює поверх IP, який використовується для відправки даних. Саме тому протоколи TCP і IP нерозривно зв'язані і їх часто об'єднують однією назвою TCP/IP.

 На відміну від UDP-протокол TCP усуває недоліки свого транспорту (IP). У цьому протоколі закладені засоби встановлення зв'язку між приймачем і передавачем, забезпечення цілісності даних і гарантії їх доставки.

 Коли дані відправляються в мережу по TCP, то на відправляючій стороні включається таймер. Якщо протягом певного часу приймач не підтвердить отримання даних, то буде зроблена ще одна спроби відправки даних. Якщо приймач одержить зіпсовані дані, то він повідомить про це джерелу і попросить знову відправити зіпсовані пакети. Завдяки цьому забезпечується гарантована доставка даних.

 Коли потрібно відправити відразу велику порцію даних, що не вміщаються в один пакет, то вони розбиваються на декілька TCP-пакетів. Пакети відправляються порціями по декілька штук (залежить від настройок стека). Коли сервер одержує порцію пакетів, то він відновлює їх черговість і збирає дані разом (навіть якщо пакети прибули не в тому порядку, в якому вони відправлялися).

 Із-за зайвих накладних витрат на установку з'єднання підтвердження доставки і повторну пересилку зіпсованих даних протокол TCP набагато повільніше за UDP. Зате TCP можна використовувати там, де потрібна гарантія доставки і велика надійність. Хоча надійність не можна назвати сильною (немає шифрування, зберігається можливість злому), але вона прийнятна і набагато більше, ніж у UDP. Принаймні, тут спуфінг не може бути реалізований так просто, як у UDP, і в цьому ви переконаєтеся, коли прочитаєте про процес установки з'єднання. Хоча можливо все, і хакери уміють зламувати і ТСР-протокол.

 **Небезпечні зв'язки TCP**

 Давайте подивимося, як протокол TCP забезпечує надійність з'єднання. Все починається ще на етапі спроби з'єднання двох комп'ютерів в наступній послідовності:

 1. Клієнт, який хоче з'єднатися з сервером, відправляє SYN-запит на сервер, указуючи номер порту, до якого він хоче під'єднатися, і спеціальне число (найчастіше випадкове).

 2. Сервер відповідає своїм сегментом SYN, що містить спеціальне число сервера. Він також підтверджує прихід SYN-пакету з боку клієнта з використанням аск-відповіді, де спеціальне число, відправлене клієнтом, збільшене на 1.

 3. Клієнт повинен підтвердити прихід SYN від сервера з використанням АСЬК - спеціальне число сервера плюс 1.

 Виходить, що при з'єднанні клієнта з сервером вони обмінюються спеціальними числами. Ці числа і використовуються надалі для забезпечення цілісності і захищеності зв'язку. Якщо хтось інший захоче уклинитися у встановлений зв'язок (за допомогою спуфінга), то йому треба буде підроблювати ці числа. Але оскільки вони великі і вибираються випадковим чином, то таке завдання достатньо складне, хоча Кевін Мітник свого часу зміг вирішити її. Але це вже інша історія, і не йтимемо далеко убік.

 Варто ще відзначити, що прихід будь-якого пакету підтверджується аск-відповіддю, що гарантує доставку даних.

 **Прикладні протоколи - загадочныйNetBIOS**

 NetBIOS (NetworkBasicInputOutputSystem, базова система мережевого введення/висновку) - це стандартний інтерфейс прикладного програмування. А простіше кажучи, це всього лише набір API-функцій для роботи з мережею (хоча весь NetBIOS складається тільки з однієї функції, та зате який...). NetBIOS був розроблений в 1983 році компанією SytekCorporation спеціально для IBM.

 Система NetBIOS визначає тільки програмну частину передачі даних, тобто як повинна працювати програма для передачі даних по мережі. А ось як фізично передаватимуться дані, в цьому документі не мовиться ні слова, та і в реалізації відсутнє що-небудь подібне.

 NetBIOS тільки формує дані для передачі, а фізично передаватися вони можуть тільки за допомогою інших протоколів, наприклад, TCP/IP, IPX/SPX і т.д. Це означає, що NetBIOS є незалежним від транспорту. Якщо інші варіанти протоколів верхнього рівня (що тільки формують пакети, але не передавальні) прив'язані до певного транспортного протоколу, який повинен передавати сформовані дані, то пакети NetBIOS може передавати будь-який інший протокол. Відчув силу? Уявіть, що ви написали мережеву програму, що працює через NetBIOS. А якщо ви ще не знаєте, то вона буде прекрасна працювати як в UNIX/Windows-сетях через TCP, так і в Novell-мережах через IPX.

 З іншого боку, для того, щоб два комп'ютери змогли з'єднатися один з одним по NetBIOS, необхідно, щоб на обох стояв хоч би один загальний транспортний протокол. Якщо один комп'ютер посилатиме NetBIOS -пакеты через TCP, а інший - за допомогою IPX, то ці комп'ютери один одного не зрозуміють. Транспорт повинен бути однаковий.

 Варто відразу ж відзначити, що не всі варіанти транспортних протоколів за умовчанням можуть передавати по мережі NetBIOS-пакети. Наприклад, IPX/SPX сам по собі цього не уміє. Щоб його навчити, потрібно мати "NWLink IPX/SPX/NetBIOSCompatibleTransportProtocol".

 Оскільки NetBIOS найчастіше використовує як транспорт протокол TCP, який працює з установкою віртуального з'єднання між клієнтом і сервером, то по цьому протоколу можна передавати достатньо важливі дані. Цілісність і надійність передачі здійснюватиме TCP/IP, а NetBIOS дає тільки зручне середовище для роботи з пакетами і програмування мережевих додатків. Отже якщо вам потрібно відправити до мережі які-небудь файли, то можна сміливо покластися на NetBIOS.

 **NetBEUI**

 У 1985 році вже сама IBM зробила спробу перетворити NetBIOS на повноцінний протокол, який уміє не тільки формувати дані для передачі, але і фізично передавати їх по мережі. Для цього був розроблений NetBEUI (NetBIOSExtendedUserInterface, розширений призначений для користувача інтерфейс NetBIOS). Він призначений саме для опису фізичної частини передачі даних протоколу NetBIOS.

 Відразу хочу відзначити, що NetBEUI є немаршрутізіруємим протоколом, і перший же маршрутизатор відбиватиметься від таких пакетів як тенісистка від м'ячиків :). Це означає, що якщо між двома комп'ютерами стоїть маршрутизатор і немає іншого шляху для зв'язку, то їм не вдасться встановити з'єднання через NetBEUI.

 **Сокети Windows**

 Сокети (Sockets) - це всього лише програмний інтерфейс, який полегшує взаємодію між різними додатками. Сучасні сокети народилися з програмного мережевого інтерфейсу, реалізованого в ОС BSD UNIX. Тоді цей інтерфейс створювався для полегшення роботи з TCP/IP на верхньому рівні.

 За допомогою сокетів легко реалізувати більшість відомих протоколів, які використовуються щодня при виході в Інтернет. Достатньо тільки назвати HTTP, FTP POP3, SMTP і далі в тому ж дусі. Всі вони використовують для відправки своїх даних або TCP, або UDP і легко програмуються за допомогою бібліотеки sockets/winsock.

 **Протоколи IPX/SPX**

 Залишилося тільки розповісти ще про декілька протоколів, які зустрічаються в повсякденному житті трохи рідше, та зате вони не менш корисні. Перші на черзі - це IPX/SPX.

 Протокол IPX (InternetworkPacketExchange, міжмережевий обмін пакетами) зараз використовується, напевно, тільки в мережах фірми Novell. У наших улюблених віконцях є спеціальна служба **Клієнт для мереж Novell**, за допомогою якої ви зможете працювати в таких мережах. IPX працює подібно до IP і UDP - без встановлення зв'язку, а значить, без гарантії доставки і всіх подальших достоїнств і недоліків.

 SPX (SequencePacketExchange, послідовний обмін пакетами) - це транспорт для IPX, який працює зі встановленням зв'язку і забезпечує цілісність даних. Отже якщо вам знадобиться надійність при використанні IPX, то використовуй зв'язку IPX/SPX або IPX/SPX11.

 Зараз IPX вже втрачає свою популярність, але ще пам'ятаються часи DOS, коли всі мережеві ігри працювали через цей протокол.

 Як бачите, в Інтернеті протоколів ціле море, але більшість з них взаємозв'язана, як, наприклад, HTTP-TCP-IP. Протокол, призначений для однієї мети, може виявитися абсолютно непридатним для іншої, тому що створити щось ідеальне неможливе. У кожного будуть свої достоїнства і недоліки.

 Та все ж модель OSI, прийнята ще на зорі Інтернету, не втратила своєї актуальності дотепер. По ній працює все і вся. Головне її достоїнство - приховувати складність мережевого спілкування між комп'ютерами, з чим старенька OSI справляється без особливих проблем.

 **Мережеві порти**

 Перш ніж ви почнете писати власні програми, треба розібратися з ще одним поняттям - **мережевий порт**. Допустимо, що вашому комп'ютеру на мережеву карту прийшов пакет даних. Як операційна система повинна визначити, для якої програми пришли дані: для InternetExplorer, для поштового клієнта або для вашої програми? Щоб визначити це, використовуються порти.

 Коли програма з'єднується з сервером, то вона відкриває на вашому комп'ютері який-небудь мережевий порт і повідомляє сервер, що саме з цим портом вона працює. Після цього сервер посилатиме на ваш комп'ютер пакети даних, в яких буде вказана мережева адреса комп'ютера і номер порту. По IP-адресі пакет буде доставлений до вашого комп'ютера, а по номеру порту операційна система визначить, що саме для вашої програми призначається пакет, що прийшов.

 Для з'єднання з сервером вам треба знати не тільки IP-адресу сервера, але і порт, на якому працює програма, тому що на сервері може працювати безліч мережевих програм, і всі вони використовують свої порти.

 Зі всього вищесказаного виходить, що тільки одна програма може відкрити певний порт. Якби дві програми могли відкривати, наприклад, 21-й порт, то Windows (або будь-яка інша операційна система) вже не змогла б визначити, якій з двох програм пришли дані.

 Номер порту - це число від 1 до 65 535. Для передачі такого числа по мережі достатньо всього лише двох байт, тому це не буде невигідно для мережі. Я рекомендую використовувати для своїх цілей порти з номерами більше 1024, тому що серед менших значень дуже багато зареєстрованих номерів, і у вашої програми збільшується вірогідність конфлікту з іншими мережевими програмами.

 Тепер пора переходити до докладнішого розгляду деяких протоколів і мережевих можливостей Windows. Я не зможу пояснити абсолютно все, але постараюся розглянути найцікавіше в мережевому програмуванні і показати декілька корисних прикладів.

#  Робота з ресурсами мережевого оточення

 У ОС Windows є дуже зручна можливість - обмінюватися інформацією між комп'ютерами через відкриті ресурси. Ви можете зробити яку-небудь теку відкритою для мережевого доступу, і будь-який користувач у вашій мережі, у якого є відповідні права, зможе звертатися до файлів цієї теки. Можна також підключити відкриту теку як локальний диск. У будь-якому випадку для доступу до таких ресурсів можна використовувати стандартні функції для доступу до файлів.

 Коли додаток використовує файл, то ОС спочатку визначає пристрій, на якому знаходиться необхідний ресурс. Якщо ресурс розташований на видаленому комп'ютері, то запит на введення/висновок передається по мережі цьому пристрою. Таким чином, ОС при зверненні до мережевого ресурсу займається перенаправленням введення/висновку (I/O redirection).

 Допустимо, що у вас диск Z: - це підключена по мережі тека з видаленого комп'ютера. Кожного разу, коли ви звертаєтеся до неї, ОС переадресовуватиме запити введення/висновку перенаправітелю (redirector), яке створить мережевий канал зв'язку з видаленим комп'ютером для доступу до його ресурсів. Таким чином, можна використовувати ті ж засоби, що і для доступу до локальних ресурсів. Це сильно полегшує створення додатків, призначених для роботи в локальній мережі. Точніше сказати, ніяких змін вносити не треба. Якщо програма уміє працювати з локальним диском, то зможе працювати і з видаленими дисковими ресурсами.

 Для докладнішої інформації по роботі редіректора можете звернутися до документації по Windows або спеціалізованій літературі. Для простого користувача і навіть програміста ця інформація не дуже важлива, тому що весь процес перенаправлення прихований.

 Щоб забезпечити доступ до ресурсів іншого комп'ютера у вашій мережі, не обов'язково підключати відкриту теку як локальний диск. Достатньо правильно написати мережевий шлях. Для цього треба знати універсальні правила іменування (Universal Naming Conversion, UNC) - спосіб доступу до файлів і пристроїв (наприклад, до принтерів) без призначення їм букви локального диска. Тоді ви не залежатимете від імен дисків, але потрібно буде чітко визначити ім'я комп'ютера, на якому знаходиться потрібний об'єкт.

 Загальний вид UNC-імені виглядає таким чином:

 \ компьютер\ имя\ шлях

 Ім'я починається з подвійною косою межі (\). Потім йде ім'я комп'ютера або сервера, на якому розташований об'єкт, ім'я - це ім'я мережевої теки. Після цього потрібно вказати шлях до об'єкту.

 Допустимо, що у вас є комп'ютер *Тоm,* на якому відкрита для загального доступу тека *Sound.* У цій теці є файл *MySound.wav.* Для доступу до цього файлу необхідно використовувати UNC-ім'я: \Tom\Sound\MySound.wav.

 У лістингу нижче приведений приклад створення файлу у відкритій теці комп'ютера з ім'ям **Notebook**.

 **Приклад створення файлу у відкритій теці іншого комп'ютера**

 void CreateNetFile()

 {

 HANDLE FileHandle;

 DWORD BWritten;

 // Create file \notebook\temp\myfile.txt

 // Створення файлу \notebook\temp\myfile.txt

 if ((FileHandle = CreateFile("\notebook\temp\myfile.txt",

 GENERIC\_WRITE | GENERIC\_READ,

 FILE\_SHARE\_READ | FILE\_SHARE\_WRITE, NULL

 CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, NULL

 )) == INVALID\_HANDLE\_VALUE)

 MessageBox(0, "Create file error", "Error",0);

 return;

 // Write to file 9 symbols

 // (Записати у файл 9 символів)

 if (WriteFile(FileHandle, "Test line", 9 &BWritten, NULL)== 0)

 MessageBox(0, "Write to file error", "Error",0);

 return;

 // Close file (Закрити файл)

 CloseHandle(FileHandle);

 }

 Спершу створюється файл за допомогою стандартної WinAPI -функции CreateFile . У цієї функції наступні параметри:

* шлях до створюваного файлу;
* режим доступу - файл відкритий для читання (GENERIC\_READ) і запису (GENERIC\_WRITE);
* режим доступу до відкритого файлу іншим профаммам - іншим додаткам дозволено читання (FILE\_SHARE\_READ) і запис (FILE\_SHARE\_WRITE);
* атрибути безпеки - не використані (NULL);
* спосіб відкриття файлу - завжди створювати (CREATE\_ALWAYS), якщо файл вже існує, то дані будуть перезаписані;
* атрибути створюваного файлу - нормальний стан файлу (FILE\_ATTRIBUTE\_NORMAL);
* покажчик на шаблон, який використовуватиметься при створенні файлу.

 Функція CreateFile повертає покажчик на відкритий файл. Якщо результат рівний INVALID\_HANDLE\_VALUE, то файл не був створений з яких-небудь причин.

 Для запису використовується функція WriteFile, у якої наступні параметри:

* покажчик на відкритий файл;
* дані, які треба записати;
* кількість байт даних для запису;
* кількість записаних байт (змінну типу DWORD);
* структура, яка необхідна тільки при відкритті файлу в режимі накладення (overlapped I/O).

 Якщо запис пройшов успішно, то функція повинна повернути ненульове значення.

 Після всіх маніпуляцій з файлом його необхідно закрити. Для цього викликається функція CloseHandle, який потрібно тільки передати покажчик на файл, який треба закрити.

#  Структура мережі

 Для того, щоб проглянути доступні у вашій мережі комп'ютери, потрібно скористатися мережевим оточенням. Але що, якщо вам потрібно в своїй програмі зробити проглядання мережі? Це дуже просто. Зараз я продемонструю програму, за допомогою якої можна буде у вигляді дерева побачити всі комп'ютери в мережі і їх відкриті ресурси.

 Створіть новий MFC-додаток в Visual C++ і назвіть проект NetNeighbour. У Майстрі створення додатків, в розділі **Application Type** виберіть **Dialog based**, а в розділі **Advanced Features - Windows sockets**. На тисніть кнопку **Finish**, щоб середовище розробки створило необхідний шаблон додатку.

 Перш ніж приступати до програмування, необхідно оформити вікно майбутньої програми. Відкрийте в редакторі ресурсів діалогове вікно IDD\_NETNEIGHBOUR\_DIALOG. Розтягніть по всій вільній поверхні компонент Tree Control.

 Щоб можна було працювати з цим компонентом, клацніть по ньому правою кнопкою мишки. У меню, що з'явилося, виберіть пункт **Add variable**, а в полі **Variable name** вкажіть m\_NetTree. Ця змінна знадобиться для додавання в меню нових пунктів.

 Тепер все готово для розгляду початкового коду. Перейдіть у файл NetNeighbourDlg.cpp. Тут знайдіть функцію OnInitDialog, яка викликається під час ініціалізації вікна. У цей момент необхідно створити кореневий елемент дерева. Це повинно відбуватися таким чином:

 m\_hNetworkRoot = InsertTreeItem(TVI\_ROOT, NULL, "My Net", DRIVE\_RAMDISK +1);

 У змінній m\_hNetworkRoot зберігається результат роботи функції InsertTreeItem.

 Доведеться кілька разів використовувати такий же код для додавання елементів, і щоб в одному модулі не повторювати одні і ті ж дії, я все оформив окремою функцією.

 **Додавання нового елементу в дерево мережі**

 HTREEITEM CNetNeighbourDlg::InsertTreeItem(HTREEITEM hParent

 NETRESOURCE \*const pNetResource, CString sText, int iImage)

 {

 TVINSERTSTRUCT InsertStruct;

 InsertStruct.hParent = hParent;

 InsertStruct.hInsertAfter = TVI\_LAST;

 InsertStruct.itemex.mask = TVIF\_IMAGE | TVIF\_TEXT |

 TVIF\_CHILDREN | TVIF\_PARAM;

 InsertStruct.itemex.pszText = sText.GetBuffer(sText.GetLength());

 InsertStruct.itemex.iImage = iImage;

 InsertStruct.itemex.cChildren = 1;

 InsertStruct.itemex.lParam = (LPARAM)pNetResource;

 sText.ReleaseBuffer();

 return m\_NetTree.InsertItem( &InsertStruct );

 }

 Тепер програма виглядає належним чином і створює кореневий елемент, але поки без пошуку в мережі. Коли програма запущена, і користувач клацне мишкою по елементу дерева, потрібно знайти все, що є доступного в мережі, що відноситься до цього елементу.

 Для цього треба написати обробник події ITEMEXPANDING і в ньому проводити пошук. Перейдіть в редактора ресурсів і виділіть компонент Tree Control . У вікні **Properties** клацніть по кнопці **Control Events**, і ви побачите всі події, які може генерувати виділений компонент. Клацніть напроти події TVN\_ITEMEXPANDING і у випадному списку виберіть пункт **Add**, щоб додати обробник події. Код, який повинен бути в цьому обробнику, приведений в лістингу.

 **Обробник події TVN\_ITEMEXPANDING**

 void CNetNeighbourDlg::OnTvnItemexpandingTree1(NMHDR \*pNMHDR,

 LRESULT \*pResult)

 {

 LPNMTREEVIEW pNMTreeView = reinterpret\_cast<LPNMTREEVIEW>(pNMHDR);

 // TODO: Add your control notification handler code here

 // (Додайте сюди ваш код обробки повідомлення)

 CWaitCursor CursorWaiting;

 ASSERT(pNMTreeView);

 ASSERT(pResult);

 if (pNMTreeView->action == 2)

 CString sPath = GetItemPath(pNMTreeView->itemNew.hItem);

 if(!m\_NetTree.GetChildItem(pNMTreeView->itemNew.hItem))

 EnumNetwork(pNMTreeView->itemNew.hItem);

 if( m\_NetTree.GetSelectedItem( ) != pNMTreeView->

 itemNew.hItem)

 m\_NetTree.SelectItem(pNMTreeView->itemNew.hItem);

 }

 \*pResult = 0;

 }

 Тут у елементу, який в даний момент намагаються відкрити, перевіряється наявність дочірніх елементів і організовується їх пошук. Для цього викликається функція EnumNetwork, яку можна побачити в лістингу.

 **Функція EnumNetwork для проглядання мережі**

 bool CNetNeighbourDlg::EnumNetwork(HTREEITEM hParent)

 {

 bool bGotChildren = false;

 NETRESOURCE \*const pNetResource = (NETRESOURCE \*)

 (m\_NetTree.GetItemData(hParent));

 DWORD dwResult;

 HANDLE hEnum;

 DWORD cbBuffer = 16384;

 DWORD cEntries = 0xFFFFFFFF;

 LPNETRESOURCE lpnrDrv;

 DWORD i;

 dwResult = WNetOpenEnum(pNetResource ? RESOURCE\_GLOBALNET :

 RESOURCE\_CONTEXT,

 RESOURCETYPE\_ANY, 0

 pNetResource ? pNetResource: NULL

 &hEnum );

 if (dwResult != NO\_ERROR)

 return false;

 do

 lpnrDrv = (LPNETRESOURCE) GlobalAlloc(GPTR, cbBuffer);

 dwResult = WNetEnumResource(hEnum &cEntries,

 lpnrDrv &cbBuffer);

 if (dwResult == NO\_ERROR)

 for(i = 0; i<cEntries; i++)

 CString sNameRemote = lpnrDrv[i].lpRemoteName;

 int nType = 9;

 if(sNameRemote.IsEmpty())

 sNameRemote = lpnrDrv[i].lpComment;

 nType = 8;

 if (sNameRemote.GetLength() > 0 &&

 sNameRemote[0]== \_T('\'))

 sNameRemote = sNameRemote.Mid(1);

 if (sNameRemote.GetLength() > 0 &&

 sNameRemote[0]== \_T('\'))

 sNameRemote = sNameRemote.Mid(1);

 if (lpnrDrv[i].dwDisplayType ==

 RESOURCEDISPLAYTYPE\_SHARE)

 int nPos = sNameRemote.Find( \_T('\'));

 if(nPos >= 0)

 sNameRemote = sNameRemote.Mid(nPos+1);

 InsertTreeItem(hParent, NULL

 sNameRemote DRIVE\_NO\_ROOT\_DIR);

 else

 NETRESOURCE\* pResource = new NETRESOURCE;

 ASSERT(pResource);

 \*pResource = lpnrDrv[i];

 pResource->lpLocalName =

 MakeDynamic(pResource->lpLocalName);

 pResource->lpRemoteName =

 MakeDynamic(pResource->lpRemoteName);

 pResource->lpComment =

 MakeDynamic(pResource->lpComment);

 pResource->lpProvider =

 MakeDynamic(pResource->lpProvider);

 InsertTreeItem(hParent, pResource

 sNameRemote pResource->dwDisplayType+7);

 bGotChildren = true;

 GlobalFree((HGLOBAL)lpnrDrv);

 if (dwResult != ERROR\_NO\_MORE\_ITEMS)

 break;

 while (dwResult != ERROR\_NO\_MORE\_ITEMS);

 WNetCloseEnum(hEnum);

 return bGotChildren;

 }

 Логіка пошуку мережевих ресурсів достатньо проста. Спершу потрібно відкрити пошук функцією WNetOpenEnum, яка виглядає таким чином:

 DWORD WNetOpenEnum(

 DWORD dwScope // scope of enumeration

 DWORD dwType // resource types to list

 DWORD dwUsage // resource usage to list

 LPNETRESOURCE lpNetResource // pointer to resource structure

 LPHANDLE lphEnum // pointer to enumeration handle buffer

 );

 Функція відкриває перелік мережевих пристроїв в локальній мережі. Розглянемо передавані їй параметри:

* dwScope - ресурси, що включаються в перелік. Можливі комбінації наступних значень:
* RESOURCE\_GLOBALNET - всі ресурси мережі;
* RESOURCE\_CONNECTED - підключені ресурси;
* RESOURCE\_REMEMBERED - ресурси, що запам'ятали;
* dwType - тип ресурсів, що включаються в перелік. Можливі комбінації наступних значень:
* RESOURCETYPE\_ANY - всі ресурси мережі;
* RESOURCETYPE\_DISK - мережеві диски;
* RESOURCETYPE\_PRINT - мережеві принтери;
* dwUsage - використання ресурсів, що включаються в переліки. Можливі наступні значення:
* 0 - всі ресурси мережі;
* RESOURCEUSAGE\_CONNECTABLE - що підключаються;
* RESOURCEUSAGE\_CONTAINER - контейнерні;
* lpNetResource - покажчик на структуру NETRESOURCE. Якщо цей параметр рівний нулю, то перелік починається з самого верхнього ступеня ієрархії мережевих ресурсів. Нуль ставиться для того, щоб одержати найперший ресурс. Потім я передаю як цей параметр покажчик на вже знайдений ресурс. Тоді перелік почнеться з нього і продовжиться далі. Так я повторюю, поки не знайдуться всі ресурси;
* lphEnum - покажчик, який знадобиться У функції WnetEnumResource.

 Тепер потрібно розглянути структуру NETRESOURCE:

 typedef struct \_NETRESOURCE {

 DWORD dwScope;

 DWORD dwType;

 DWORD dwDisplayType;

 DWORD dwUsage;

 LPTSTR lpLocalName;

 LPTSTR lpRemoteName;

 LPTSTR lpComment;

 LPTSTR lpProvider;

 } NETRESOURCE;

 Що таке dwScope, dwType і dwUsage, ви вже знаєте. А ось інші розглянемо докладніше:

* dwDisplayType - спосіб відображення ресурсу:
* RESOURCEDISPLAYTYPE\_DOMAIN - це домен;
* RESOURCEDISPLAYTYPE\_GENERIC - немає значення;
* RESOURCEDISPLAYTYPE\_SERVER - сервер;
* RESOURCEDISPLAYTYPE\_SHARE - ресурс, що розділяється;
* lpLocalName - локальне ім'я;
* lpRemoteName - видалене ім'я;
* lpComment - коментар;
* lpProvider - господар ресурсу. Параметр може бути рівний нулю, якщо господар невідомий.

 Тепер можна переходити до наступної функції:

 DWORD WNetEnumResource(

 HANDLE hEnum // handle to enumeration

 LPDWORD lpcCount // pointer to entries to list

 LPVOID lpBuffer // pointer to buffer for results

 LPDWORD lpBufferSize // pointer to buffer size variable

 );

 Параметри функції WnetEnumResource:

* hEnum - покажчик на повернене функцією wNetopenEnum значення;
* lpcCount - максимальна кількість повертаних значень. Не соромтеся, ставте 2000. Якщо ви задасте 0xFFFFFFFF, то перерахуються всі ресурси. Після виконання функція передасть сюди фактичне число знайдених ресурсів;
* lpBuffer - покажчик на буфер, в який буде поміщений результат;
* lpBuffersize - розмір буфера.

 Після закінчення переліку викликається функція WNetCloseEnum, яка закриває початий функцією WNetOpenEnum перелік мережевих ресурсів. Як єдиний параметр потрібно передати покажчик на повернене функцією WNetOpenEnum значення.

 Це все, що стосується пошуку відкритих мережевих ресурсів. Залишилося тільки зробити одне зауваження. Функція пошуку WNetOpenEnum і відповідні їй структури знаходяться в бібліотеці mpr.lib, яка за умовчанням не лінкуєтся до проекту. Щоб зібрати проект без помилок, необхідно підключити цю бібліотеку. Для цього клацніть правою кнопкою мишки на ім'я проекту у вікні **Solution Explorer** і в меню, що з'явилося, виберіть пункт **Properties**. Перед вами відкриється вікно властивостей, в якому треба перейти в розділ **Configuration Properties/Linker/Input**. Тут в рядку **Additional Dependencies** напишіть ім'я бібліотеки mpr.lib.

#  Робота з мережею за допомогою об'єктів Visual C++

 При роботі з мережею можна використовувати можливості, які надає середовище розробки Visual C++. Об'єкти спрощують програмування і приховують деякі особливості реалізації протоколів і мережі.

 При використанні об'єктів проекти будуть достатньо великими, тому що вже не можна використовувати додатку **Win32 Project**. Проекти треба створювати за допомогою майстра **MFC Application**. Для початку цього буде досить, тому що основна мета зараз - зрозуміти процес програмування мережевих додатків. Трохи пізніше я познайомлю вас з мережевими WinAPI-функціями, і тоді ми зможемо написати ті ж додатки, але без використання об'єктів, і одержати додатки маленького розміру.

 Для роботи з мережею в MFC є дуже зручний клас - CSocket. Як предок у нього виступає CAsyncSocket. Що це означає? Об'єкт CAsyncSocket працює з мережею асихронно. Відправивши пакет до мережі, об'єкт не чекає підтвердження, і програма може продовжувати працювати далі. Про закінчення дії ми можемо дізнатися по подіях, які для нас вже реалізовані в об'єкті, і достатньо тільки написати їх обробники.

 При синхронній роботі кожна відправка пакету або з'єднання з сервером заморожує виконання програми до закінчення виконання дії. Таким чином, процесорний час витрачається нераціонально.

 Об'єкт CSocket є нащадком об'єкту CAsyncSocket, а значить, дістає всі його можливості, властивості і методи. Його робота побудована на основі технології "сервер" клієнта. Це означає, що один об'єкт може бути сервером, який приймає з'єднання клієнтів і працює з ними. З цього виходить, що в прикладах для передачі даних знадобиться створювати два об'єкти: CServerSocet (сервер) і CClientSocket (клієнт для підключення до сервера).

 Об'єкт CServerSocet схожий з CClientSocket. Сервер чекає з'єднання на певному порту, і коли клієнт підключився, створюється об'єкт CClientSocket, за допомогою якого можна відправляти і приймати дані на сервері.

 Щоб побачити на практиці роботу з мережею, давайте напишемо програму, яка скануватиме вказаний комп'ютер і шукати на ньому відкриті порти (сканер портів). Як це працює? Для того, щоб дізнатися, які порти відкриті, достатньо тільки спробувати під'єднатися до порту. Якщо з'єднання пройде успішно, то даний порт відкрила якась програма.

 Деякі вважають, що якщо якась серверна програма вимагає авторизації при з'єднанні, то не можна буде приєднатися до її порту. Ця помилкова думка, тому що авторизація відбувається тільки після з'єднання з сервером. Саме тому є можливість визначити всі відкриті порти, але тільки якщо між нами і сканованим комп'ютером не встановлений який-небудь спеціальний захисний комплекс (Firewall).

 Тепер перейдемо до справи. Створіть новий проект **MFC Application** в Visual C++ і назвіть його MFCScan. У майстрі зміните наступні параметри:

* у розділі **Application Type** - встановите тип додатку **Dialog based**;
* у розділі **Advanced Features** - виділите пункт **Windows Sockets**.

 Тепер можна натискати кнопку **Finish**, щоб Visual C++ сформував необхідні файли.

 Вже готовий шаблон додатку, але ще не вистачає об'єктів, через які додаток працюватиме з мережею. У принципі, можна обійтися і без них, але краще додати на той випадок, якщо ви захочете в майбутньому розширити можливості сканера. Клацніть правою кнопкою на імені проекту у вікні **Solution Explorer** і виберіть в меню **Add/Add Class**, що з'явилося.... Перед вами відкриється вікно додавання класу. Виберіть пункт **MFC Class** і натисніть кнопку **Open**.

 Якщо ви всі зробили правильно, то повинне відкритися вікно в якому ви повинні вказати майбутнє ім'я класу і базовий клас, від якого походитиме створюваний. Для цього в списку **Base class** (Базовий клас) знайдіть ім'я CSocket, а в поле **Class name** (Ім'я створюваного класу) введіть CClientSocket.

 Тепер в проект додалися два нові файли: ClientSocket.cpp і ClientSocket.h. З їх допомогою можна управляти з'єднанням, але поки тільки подивитеся на них.

 Відкрийте файл ClientSocket.cpp і у вікні **Properties** натисніть кнопку **Overrides**. Вокне властивостей з'являться методи і події, які можна переписати, щоб об'єкт працював так, як необхідно вам. Для цього клацніть за випадним списком напроти потрібного методу або події і виберіть пункт меню **<Add> Ім'я методу**. Для даного методу я нічого не змінюватиму.

 Тепер відкрийте файл ресурсів і знайдіть діалогове вікно IDD\_MFCSCAN\_DIALOG. Двічі клацніть по ньому, щоб відкоректувати в редакторі ресурсів. Видаліть кнопки **OК** і **Cancel**, а помістите на вікно діалогу наступні компоненти:

* Static Text - з написом "Server address";
* Edit Control - для введення адреси сканованого сервера (за умовчанням текст " Sample edit box ";
* List Box - для збереження відкритих портів;
* Button (кнопка) - з написом "Scan" для запуску сканування портів вказаного комп'ютера.

 Тепер необхідно створити змінну для списку, щоб з ним потім працювати. Для цього треба клацнути по компоненту List Box правою кнопкою мишки і у випадному меню вибрати пункт **Add Variable**. У вікні, що з'явилося, потрібно ввести в поле **Variable name** ім'я змінної. Вкажіть ім'я PortsList.

 Всі підготовчі роботи закінчені. Можна приступати до написання коду сканера портів. Необхідно створити обробник події, який спрацьовуватиме при натисненні користувачем кнопки **Scan**, і написати в ньому весь необхідний код. Для цього клацніть правою кнопкою мишки по компоненту Button і виберіть в меню, що з'явилося, пункт **Add Event Handler**. Перед вами відкриється вікно майстра **Event Handler Wizard**. Погодьтеся зі всіма установками майстра і натисніть кнопку Add **and Edit.**

 Майстер створить заготівку у вигляді порожньої функції для обробника події. У ній потрібно написати код з лістингу.

 **Код сканера портів**

 void CMFCScanDlg::OnBnClickedScanButton()

 {

 // TODO: Add your control notification handler code here

 CClientSocket \*pSocket;

 CString ip;

 CString messtr;

 int port;

 pSocket=new CClientSocket();

 pSocket->Create();

 GetDlgItemText(IDC\_EDIT1,ip);

 port=1;

 while (port<100)

 if(pSocket->Connect(ip, port))

 messtr.Format("Port=%d opened", port);

 PortsList.AddString(messtr);

 pSocket->Close();

 pSocket->Create();

 port++;

 }

 Тепер розберемося з тим, що тут відбувається. У даному коді оголошена змінна pSocket типу CClientSocket. З її допомогою ми працюватимемо з об'єктом, який уміє спілкуватися з мережею по протоколу TCP/IP. Але перш ніж почати роботу, потрібно виділити пам'ять і створити об'єкт. Це робиться в наступних двох строчках:

 pSocket=new CClientSocket();
pSocket->Create();

 Після цього слід дізнатися, яку IP-адресу вказав користувач в полі введення. Для цього використовується функція GetDlgItemText, у якої два параметри: ідентифікатор компоненту і змінна, в якій буде збережений результат.

 Можна одержати дані і за допомогою спеціальної змінної. Для цього потрібно було б клацнути по компоненту в редакторі ресурсів правою кнопкою мишки і створити змінну. Але оскільки ми одержуємо дані тільки один раз, заводити змінну не має сенсу.

 Після цього в змінну port заноситься початкове значення 1, з якого починається сканування. Потім запускається цикл, який виконуватиметься, поки змінна port не стане більше 100.

 Усередині циклу проводиться спроба з'єднання з сервером таким чином:

 pSocket->Connect(ip, port)

 Тут викликається метод Connect об'єкту, на який указує змінна pSocket. У цього методу два параметри: адреса комп'ютера, до якого треба підключитися, і порт. Якщо з'єднання пройшло вдало, то результатом буде ненульове значення. В цьому випадку треба додати інформаційний рядок в список PortsList. Дуже важливо закрити з'єднання і проїніциалізіро-вать об'єкт наново, інакше подальші спроби з'єднання з сервером будуть даремні, і ви побачите тільки перший відкритий порт. Закриття з'єднання і ініціалізація проводяться методами Close і Create відповідно:

 pSocket->Close();
pSocket->Create();

 В кінці циклу збільшується змінна port, щоб на наступному етапі циклу сканувати наступний порт.

 Тепер ви готові скомпілювати програму, але щоб все пройшло вдало, потрібно перейти в початок модуля, де перераховані заголовні файли, що підключаються, і додати наступний рядок:

 #include "ClientSocket.h"

 Був використаний об'єкт CClientSocket, який описаний у файлі ClientSocket.h, тому без підключення модуля код не скомпілюється.

 Запустіть програму і, вказавши як адресу 127.0.0.1, проскануйте порти свого комп'ютера, починаючи з 0 до 99. Чому скануємо так мало портів? У Windows процес сканування 1000 портів відбувається дуже поволі (може зайняти близько 5 хвилин), тому сканувати краще маленькими порціями.

 Трохи пізніше я покажу більш довершений приклад по скануванню портів, а дана програма є чисто пізнавальною і дуже добре підходить для розуміння алгоритму сканування. Якщо у вас великий досвід програмування в середовищі Visual C++ і ви знайомі з потоками, то я все одно не раджу вам створювати безліч потоків, щоб кожний з них сканував свій порт. У такий спосіб ви прискорите програму, але під час сканування система буде навантажена і, причому, марно. Потерпіть небагато, і ви познайомитеся з реально швидким сканером портів.

#  Робота безпосередньо з WinSock

 Як бачите, робота з мережею з використанням MFC-об'єктів, а саме CSocket, дуже проста. Але ви не зможете таким чином написати маленький додаток, тому що для цього треба відмовитися від використання стандартних бібліотек. Саме тому я розгляну роботу з мережею безпосередньо достатньо детально.

 У Windows для роботи з мережею використовується бібліотека WinSock. Існують дві версії цієї бібліотеки. Перша версія WinSock розроблялася на основі моделі сокетів Берклі, використовуваної в UNIX-системах. Починаючи з Windows 98 в ОС вже вбудована друга версія.

 Бібліотека WinSock назад сумісна. Це означає, що старі функції не змінилися, і програми, написані для першої версії, чудово працюватимуть в другій. У пізніших версіях Microsoft додала нові функції, але вони опинилися несумісні з мережевими функціями на інших платформах. Вперше нововведення з'явилися у версії 1.1, і це були WSAStartup, WSACleanup, WSAGetLastError, WSARecvEx (імена починаються з "WSA"). У наступній версії таких функцій стало набагато більше.

 Якщо вам доступна версія WinSock2, то не обов'язково її використовувати. Подивіться, може бути можливостей першої версії опиниться достатньо, і тоді вашу програму буде легко адаптувати до компіляції на платформі UNIX.

 Звичайно, комп'ютери зі встановленою Windows 95 зустріти вже достатньо складно, але вони існують. Якщо ви володар такої ОС, то ви можете викачати нову версію бібліотеки з сайту **www.microsoft.com <http://www.microsoft.com>**.

 Якщо ви вирішили використовувати в своїй програмі першу версію, то необхідно підключити заголовний файл winsock.h, інакше - winsock2.h.

 Відразу попереджу, що я використовуватиму WinSock і WinSock2.

 **Обробка помилок**

 На самому початку необхідно дізнатися, як можна визначити помилки, які виникають при виклику мережевих функцій. Правильна обробка помилок для будь-якого додатку є найважливішою. Хоча мережеві функції не критичні для ОС, але можуть вплинути на хід роботи програми. А це в свою чергу може привести до пониження безпеки системи.

 Мережеві додатки обмінюються даними з чужими комп'ютерами, а це означає, що як сторонній клієнт може виступити зловмисник. Якщо не обробити помилку, це може привести до доступу до важливих даних або функцій.

 Приведу простий приклад. У вас є функція, яка викликається кожного разу, коли програмі пришли дані, і перевіряє їх на коректність і права доступу. Якщо все нормально, то функція виконує критичний код, який не повинен бути доступний зловмиснику. Контроль повинен відбуватися на кожному етапі: отримання даних, перевірка їх коректності і доступності, а також будь-який виклик мережевої функції. Пам'ятайте, чтоето додасть стабільність і надійність вашій програмі і забезпечить безпеку всієї системи.

 Якщо під час виконання якої-небудь функції відбулася помилка, то вона поверне константу SOCKET\_ERROR або -1. Якщо ви набули такого значення, то можна скористатися функцією WSAGetLastError. Їй не треба передавати параметри, вона просто поверне код помилки, яка відбулася під час виконання останньої мережевої функції. Кодів помилок дуже багато, і **вони** залежать від функції, яка відпрацювала останній. Я розглядатиму їх в міру необхідності.

 **Запуск бібліотеки**

 Перш ніж почати роботу з мережею, потрібно завантажити необхідну версію бібліотеки. Залежно від цього змінюється набір доступних функцій. Якщо використовувати першу версію бібліотеки, а викликати функцію з другої, то відбудеться збій в роботі програми. Якщо не завантажити бібліотеку, то будь-який виклик мережевої функції поверне помилку WSANOTINITIALISED.

 Для завантаження бібліотеки використовується функція WSAStartup, яка виглядає таким чином:

 int WSAStartup (

 WORD wVersionRequested

 LPWSADATA lpWSAData

 );

 Перший параметр (wVersionRequested) - це запрошувана версія бібліотеки. Молодший байт указуваного числа визначає основний номер версії, а старший байт - додатковий номер. Щоб легше було працювати з цим параметром, я раджу використовувати макрос MAKEWORD(i, j), де i - це старший байт, а j - молодший.

 Другий параметр функції WSAStartup - це покажчик на структуру WSADATA, в якій після виконання функції знаходитиметься інформація про бібліотеку.

 Якщо завантаження пройшло успішно, то результат буде нульовим, інакше - відбулася помилка.

 Подивіться приклад використання функції WSAStartup для завантаження бібліотеки WinSock 2.0:

 WSADATA wsaData;

 int err = WSAStartup(MAKEWORD(2, 0) &wsaData);

 if (err != 0)

 {

 // Tell the user that WinSock not loaded

 // ( Сказати користувачу, що бібліотека не завантажена )

 return;

 }

 Зверніть увагу, що відразу після спроби завантажити бібліотеку йде перевірка поверненого значення. Якщо функція відпрацювала правильно, то вона повинна повернути нульове значення. Приведу основні коди помилок:

* WSASYSNOTREADY - основна мережева підсистема не готова до мережевого з'єднання;
* WSAVERNOTSUPPORTED - запрошувана версія бібліотеки не підтримується;
* WSAEPROCLIM - перевищена межа підтримуваних ОС завдань;
* WSAEFAULT - неправильний покажчик на структуру WSAData.

 Структура WSADATA виглядає таким чином:

 typedef struct WSAData {

 WORD wVersion;

 WORD wHighVersion;

 char szDescription[WSADESCRIPTION\_LEN+l];

 char szSystemStatus[WSASYS\_STATOS\_LEN+l];

 unsigned short iMaxSockets;

 unsigned short iMaxUdpDg;

 char FAR \* lpVendorInfo;

 } WSADATA, FAR \* LPWSADATA;

 Розберемо кожен параметр окремо:

* wVersion - версія завантаженої бібліотеки WinSock;
* wHighVersion - остання версія;
* szDescription - текстовий опис, який заповнюється не всіма версіями;
* szSystemStatus - текстовий опис стану, який заповнюється не всіма версіями;
* iMaxSockets - максимальна кількість з'єднань, що відкриваються. Ця інформація не відповідає дійсності, тому що максимальне число залежить тільки від доступних ресурсів. Параметр залишився тільки для сумісності з первинною специфікацією;
* iMaxUdpDg - максимальний розмір дейтаграмми (пакету). Інформація не відповідає дійсності, тому що розмір залежить від протоколу;
* lpVendorInfо - інформація про виробника.

 Давайте розглянемо невеликий приклад, за допомогою якого завантажуватиметься бібліотека WinSock із структури WSAData . Створіть новий проект **MFC Application**. У Майстрі створення додатків, в розділі **Application Type** виберіть **Dialog based**, а в розділі **Advanced Features - Windows sockets**. Це вже знайомий вам тип додатку.

 Відкрийте в редакторі ресурсів діалогове вікно. На формі повинні бути 4 поля **Edit Control** для виведення інформації про завантажену бібліотеку і кнопка **Get WinSock Info**, по натисненню якої завантажуватимуться дані.

 Для кожного поля вводяться змінні:

* + номер версії - mVersion;
	+ остання версія - mHighVersion;
	+ опис - mDescription;
	+ стан - mSystemStatus.

 Створіть обробник події для кнопки **Get WinSock Info** і напишіть в ньому код з лістингу.

 **Отримання інформації про WinSock**

 void CWinSockInfoDlg::OnBnClickedButton1()

 {

 // TODO: Add your control notification handler code here

 WSADATA wsaData;

 int err = WSAStartup(MAKEWORD(2, 0) &wsaData);

 if (err != 0)

 // Tell the user that WinSock not loaded

 return;

 char mText[255];

 mVersion.SetWindowText(itoa(wsaData.wVersion, mText, 10));

 mHighVersion.SetWindowText(itoa(wsaData.wHighVersion, mText, 10));

 if (wsaData.szDescription)

 mDescription.SetWindowText(wsaData.szDescription);

 if (wsaData.szSystemStatus)

 mSystemStatus.SetWindowText(wsaData.szSystemStatus);

 }

 На самому початку запускається WinSock (код, який я вже приводив). Після цього одержана інформація просто виводиться в поля на формі діалогу.

 У приведеному прикладі є один недолік - не вивантажується бібліотека. для звільнення бібліотеки використовується функція WSACleanup:

 int WSACleanup(void);

 Функції не потрібні параметри, вона просто звільняє бібліотеку, після чого робота з мережевими функціями стає недоступною.

 **Створення сокета**

 Після завантаження бібліотеки необхідно створити сокет, за допомогою якого відбувається робота з мережею. Для цього в першій версії бібліотеки є функція socket:

 SOCKET socket (

 int af

 int type

 int protocol

 );

 У версії WinSock2 для створення сокета можна використовувати функцію WSASocket.

 SOCKET WSASocket (

 int af

 int type

 int protocol

 LPWSAPROTOCOL\_INFO lpProtocolInfo

 GROUP g

DWORD dwFlags

 );

 Перші три параметри і повертане значення для обох функцій однакові. І в тому, і в іншому випадку функція повертає створений сокет, який використовуватиметься надалі при роботі з мережею. Давайте розглянемо загальні параметри:

* af - сімейство протоколів, які можна використовувати:
* AF\_UNSPEC - специфікація не вказана;
* AF\_INET - інтернет-протоколи TCP, UDP і т.д. У даній книзі я використовуватиму саме ці протоколи, як найпопулярніші і поширеніші;
* AF\_IPX - протоколи IPX, SPX;
* AF\_APPLETALK - протокол AppleTalk;
* AF\_NETBIOS - протокол NetBIOS;
* type - специфікація для нового сокета. Тут можна указувати одне з наступних значень:
* SOCK\_STREAM - передача з установкою з'єднання. Для інтернет-протоколів використовуватиметься TCP;
* SOCK\_DGRAM - передача даних без установки з'єднання. Для інтернет-протоколів використовуватиметься UDP;
* protocol - протокол для використання. Протоколів дуже багато, і ви можете дізнатися про використовувані константи в довідковій системі по програмуванню, а я найчастіше використовуватиму константу IPPROTO\_TCP, яка відповідає протоколу TCP.
* У функції WSASocket додані ще три параметри:
* lpProtocolInfo - покажчик на структуру WSAPROTOCOL\_INFO, у якій визначаються характеристики створюваного сокета;
* g - ідентифікатор групи сокетів;
* dwFlags - атрибути сокета.

 Детальніше з вказаними параметрами ви познайомитеся в процесі написання прикладів. Це допоможе вам краще зрозуміти їх і відразу ж побачити результат роботи.

 **Серверні функції**

 Ви вже знаєте, що протокол TCP працює за технологією "сервер" клієнта. Щоб два комп'ютери змогли встановити з'єднання, один з них повинен запустити прослуховування на певному порту. І лише після цього клієнт може приєднатися до сервера.

 Давайте розглянемо функції, необхідні для створення сервера. Насамперед слід пов'язати мережеву локальну адресу з вже створеним сокетом. Для цього використовується функція bind:

 int bind (

 SOCKET s

 const struct sockaddr FAR\* name

 int namelen

 );

 Давайте подивимося на параметри цієї функції:

* + заздалегідь створений сокет;
	+ покажчик на структуру типу sockaddr;
	+ розмір структури sockaddr, вказаної як другий параметр.

 Структура sockaddr призначена для зберігання адреси, а в різних протоколах використовується своя адресація. Тому і структура sockaddr може виглядати по-різному. Для інтернет-протоколів структура має ім'я sockaddr\_in і виглядає таким чином:

 struct sockaddr\_in {

 short sin\_family;

 u\_short sin\_port;

 struct in\_addr sin\_addr;

 char sin\_zero[8];

 };

 Розглянемо параметри цієї структури:

* + sin\_family - сімейство протоколів. Цей параметр схожий з першим параметром функції socket. Для інтернет-протоколів указується константа AF\_INET;
	+ sin\_port - порт для ідентифікації програми даними, що поступають;
	+ sin\_addr - структура SOCKADDR\_IN, яка зберігає IP-адресу;
	+ sin\_zero - використовується для вирівнювання адреси з параметра sin\_addr. Це необхідно, щоб розмір структури SOCKADDR\_IN дорівнював розміру SOCKADDR.

 Зараз я хочу докладніше зупинитися на портах. Ви повинні бути дуже уважні при виборі порту, тому що якщо він вже зайнятий якою-небудь програмою, то друга спроба закінчиться помилкою. Ви повинні знати, що деякі порти зарезервовані для певних (найбільш популярних) служб. Номери цих портів розподіляються центром Internet Assigned Numbers Authority. Існує три категорії портів:

* 0-1023 - управляються IANA і зарезервовані для стандартних служб. Не рекомендується використовувати порти з цього діапазону;
* 1024-49151 - зарезервовані IANA, але можуть використовуватися процесами і програмами. Більшість з цих портів можна використовувати;
* 49152-65535 - приватні порти, ніким не зарезервовані.

 Якщо під час виконання функції bind з'ясується, що порт вже використовується якою-небудь службою, то функція поверне помилку WSAEADDRINUSE.

 Давайте розглянемо приклад коду, який створює сокет і прив'язує до нього мережеву локальну адресу:

 SOCKET s=socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

 struct sockaddr\_in addr;

 addr.sin\_family = AF\_INET;

 addr.sin\_port = htons(4888);

 addr.sin\_addr.s\_addr=htonl(INADDR\_ANY);

 bind(s (SOCKADDR\*)&addr), sizeof(addr);

 У даному прикладі створюється сокет з наступними параметрами:

* AF\_INET - означає, що використовуватиметься сімейство інтернет-протоколів;
* SOCK\_STREAM - указує на протокол, що встановлює з'єднання;
* IPPROTO\_TCP - використовується протокол TCP.

 Потім оголошується структура addr типу sockaddr\_in. У параметрі sin\_family структури також указується сімейство інтернет-протоколів (AF\_INET). У параметрі sin\_port указується номер порту. Байти в номері повинні слідувати в певному порядку, який несумісний з порядком байт в числових змінних мови З, тому відбувається перетворення за допомогою функції htons.

 У параметрі sin\_addr.s\_addr указується спеціальна адреса inaddr\_any, який дозволить надалі програмі чекати з'єднання на будь-якому мережевому інтерфейсі. Це означає, що якщо у вас дві мережеві карти, сполучені з різними мережами, то програма чекатиме з'єднання з обох мереж. Є ще одна адреса, яку можна вказати, - INADDR\_ANY. Дозволяє розсилати широкомовні дані для всіх комп'ютерів мережі.

 Після того, як локальна адреса і порт прив'язані до сокета, можна приступити до прослуховування порту в очікуванні з'єднання з боку клієнта. Для цього служить функція listen, яка виглядає таким чином:

 int listen (

 SOCKET s

 int backlog

 );

 Перший параметр - це все той же сокет, який був створений і до якого прив'язана адреса. За цими даними функція визначить, на якому порту потрібно запустити прослуховування.

 Другий параметр - це максимально допустиме число запитів, чекаючих обробки. Допустимо, що ви вказали тут значення 3, а вам прийшло 5 запитів на з'єднання від різних клієнтів. Тільки перші три з них встануть в чергу, а інші одержать помилку WSAECONNREFUSED, тому при написанні клієнта (у частині з'єднання) обов'язково повинна бути перевірка.

 При виклику функції listen ви можете одержати наступні основні помилки:

* WSAEINVAL - функція bind не була викликана для даного сокета;
* WSANOTINITIALISED - не завантажена бібліотека WinSock, тобто не виконана функція WSAStartup;
* WSAENETDOWN - порушена мережева підсистема;
* WSAEISCONN - сокет вже підключений.

 Решта помилок виникає рідше.

 Коли клієнт потрапляє в чергу на підключення до сервера, необхідно вирішити з'єднання за допомогою функції accept. Вона виглядає таким чином:

 SOCKET accept (

 SOCKET s

 struct sockaddr FAR\* addr

 int FAR\* addrlen

 );

 У другій версії є функція WSAAccept, у якої перші три параметри такі ж, як і у функції accept. Функція WSAAccept виглядає таким чином:

 SOCKET WSAAccept (

 SOCKET s

 struct sockaddr FAR \* addr

 LPINT addrlen

 LPCONDITIONPROC lpfnCondition

 DWORD dwCallbackData

 );

 Давайте розглянемо загальні параметри для цих функцій :

* заздалегідь створений і запущений на прослуховування сокет;
* покажчик на структуру типу sockaddr;
* розмір структури sockaddr, вказаної як другий параметр.

 Після виконання функції accept другий параметр (addr) міститиме відомості про IP-адресу клієнта, який провів підключення. Ці дані можна використовувати для контролю доступу до сервера по IP-адресі. Але ви повинні пам'ятати, що зловмиснику не складає труднощів підроблювати IP-адресу, тому такий захист не можна назвати достатньою, але вона може ускладнити злом сервера.

 Функція accept повертає покажчик на новий сокет, який можна використовувати для спілкування з клієнтом. Стара змінна типу SOCKET продовжує слухати порт в очікуванні нових з'єднань, і її використовувати немає сенсу. Таким чином, для кожного підключеного клієнта буде свій SOCKET, дякуючи чому ви зможете працювати з будь-яким з них.

 Якщо ви пригадаєте приклад з передачею даних з використанням MFC-об'єктів *(див. разд. 4.5),* то там застосовувався той же метод. Як тільки клієнт підключався до сервера, ми створювали новий сокет, через який і відбувалася робота з клієнтом, що підключився. Саме цей сокет приймав дані, що прийшли по мережі, і міг їх відправляти назад програмі на стороні клієнта.

 **Клієнтські функції**

 Для з'єднання з сервером потрібні два етапи - створити сокет і підключитися до нього. Але це в ідеальному випадку. Як правило, додається ще один етап. Який? Простим користувачам важко працювати з IP-адресами, тому найчастіше вони використовують символьні імена серверів. В цьому випадку необхідно перед з'єднанням з сервером виконати ще одну дію - перевести символьне ім'я в IP-адресу.

 Як створювати сокет, ви вже знаєте. Тепер давайте розберемося з процесом визначення IP-адреси. Для цього використовується одна з двох функцій: gethostbyname або WSAAsyncGetHostByName (залежно від версії WinSock). Спершу розглянемо функцію gethostbyname:

 struct hostent FAR \* gethostbyname (

 const char FAR \* name

 );

 Як єдиний параметр потрібно передати символьне ім'я сервера. Функція повертає структуру типу hostent, яку розглянемо трохи пізніше.

 Тепер переходимо до розгляду WSAAsyncGetHostByName:

 HANDLE WSAAsyncGetHostByName (

 HWND hWnd

 unsigned int wMsg

 const char FAR \* name

 char FAR \* buf

 int buflen

 );

 Функція виконується асихронно, а це означає, що не блокується виконання програми при її виклику. Програма працюватиме далі, але результат буде отриманий пізніше через повідомлення Windows, вказане як другий параметр. Це дуже зручно, тому що процес визначення адреси може виявитися дуже довгим, і блокування програми на тривалий час буде неефективним. Процесорний час можна використовувати для інших цілей.

 Давайте розберемо параметри докладніше:

* hWnd - дескриптор вікна, якому буде послане повідомлення після закінчення виконання асинхронного запиту;
* wMsg - повідомлення Windows, яке згенерує після визначення IP-адреси;
* name - символьне ім'я комп'ютера, адресу якого треба визначити;
* buf - буфер, в який буде поміщена структура hostent. Буфер повинен мати достатній об'єм пам'яті. Максимальний розмір можна визначити за допомогою макросу MAXGETHOSTSTRUCT;
* buflen - довжина буфера, вказаного в четвертому параметрі.

 Тепер розглянемо структуру hostent, за допомогою якої отриманий результат:

 struct hostent {

 char FAR \* h\_name;

 char FAR \* FAR \* h\_aliases;

 short h\_addrtype;

 short h\_length;

 char FAR \* FAR \* h\_addr\_list;

 };

 Проаналізуємо параметри структури:

* h\_name - повне ім'я комп'ютера. Якщо в мережі використовується доменна система, то цей параметр міститиме повне доменне ім'я;
* h\_aliases - додаткове ім'я вузла;
* h\_addrtype - тип повертаної адреси;
* h\_length - довжина кожної адреси в списку адрес;
* h\_addr\_list - список адрес комп'ютера.

 Комп'ютер може мати декілька адрес, тому структура повертає повний список, який закінчується нулем. В більшості випадків досить вибрати першу адресу із списку. Якщо функція gethostbyname визначила декілька адрес, то найчастіше по будь-якому з них можна буде з'єднатися з шуканим комп'ютером.

 Тепер безпосередньо функція з'єднання з сервером connect. Вона виглядає таким чином :

 int connect (

 SOCKET s

 const struct sockaddr FAR\* name

 int namelen

 );

 Параметри функції:

* s - заздалегідь створений сокет;
* name - структура SOCKADDR, що містить адресу сервера, до якого треба підключитися;
* namelen - розмір структури SOCKADDR, вказаної як другий параметр.

 У другій версії WinSock з'явилася функція WSAConnect:

 int WSAConnect (

 SOCKET s

 const struct sockaddr FAR \* name

 int namelen

 LPWSABUF lpCallerData

 LPWSABUF lpCalleeData

 LPQOS lpSQOS

 LPQOS lpGQOS

 );

 Перші три параметри нічим не відрізняються від параметрів функції connect. Тому розглянемо тільки нові:

* lpCallerData - покажчик на призначені для користувача дані, які будуть відправлені серверу під час установки з'єднання;
* lpCalleeData - покажчик на буфер, в який будуть поміщені передані під час з'єднання дані.

 Обидва параметри мають тип покажчика на структуру WSABUF, яка виглядає таким чином:

 typedef struct \_WSABUF {

 u\_long len;

 char FAR \* buf;

 } WSABUF, FAR \* LPWSABUF;

 Тут перший параметр - розмір буфера, а другий - покажчик на сам буфер. Останні два параметри функції WSAConnect (lpSQOS і lpGQOS) є покажчиками на структури типу QoS. Вони визначають вимоги до пропускної спроможності каналу при прийомі і передачі даних. Якщо вказати нульове значення, то це означатиме, що вимоги до якості обслуговування не пред'являються.

 Під час спроби з'єднання найчастіше можуть зустрітися наступні помилки:

* WSAETIMEDOUT - сервер недоступний. Можлива якась проблема на шляху з'єднання;
* WSAECONNREFUSED - на сервері не запущено прослуховування вказаного порту;
* WSAEADDRINUSE - вказана адреса вже використовується;
* WSAEAFNOSUPPORT - вказана адреса не може використовуватися з даним сокетом. Ця помилка виникає, коли указується адреса у форматі одного протоколу, а проводиться спроба з'єднання по іншому протоколу.

 **Обмін даними**

 Ви дізналися, як створювати сервер, і познайомилися з функціями з'єднання. Тепер необхідно навчитися тому, ради чого все це замислювалося - передавати і приймати дані. Саме ради обміну даними між комп'ютерами ми розглядали таку кількість функцій.

 Відразу відмічу, що функції створювалися тоді, коли ще не було навіть розмов про UNICODE (універсальне кодування, що дозволяє працювати з будь-якою мовою). Тому, щоб відправити дані в цьому кодуванні, потрібно привести їх до типу char\*, а довжину помножити на 2, тому що кожен символ в UNICODE займає 2 байти (на відміну від ASCII, де символ рівний одному байту).

 Щоб прийняти дані, потрібне їх спочатку відправити. Тому почну розгляд функцій обміну даними з цього режиму. Для передачі даних серверу існують функції send і WSASend (для WinSock2). Функція send виглядає таким чином:

 int send (

 SOCKET s

 const char FAR \* buf

 int len

 int flags

 );

 Функція передає наступні параметри:

* s - сокет, через який відбуватиметься відправка даних. У програмі може бути відкрито одночасно декілька з'єднань з різними серверами, і потрібно чітко визначити, який сокет треба використовувати;
* buf - буфер, що містить дані, які необхідно відправити;
* len - довжина буфера в параметрі buf;
* flags - прапори, що визначають метод відправки. Тут можна указувати поєднання з наступних значень:
* 0 - прапори не вказані;
* MSG\_DONTROUTE - пакети, що відправляються, не треба маршрутизувати. Якщо транспортний протокол, який відправляє дані, не підтримує цей прапор, то він ігнорується;
* MSG\_OOB - дані повинні бути відправлені поза чергою (out of band), тобто терміново.

 Функція WSASend виглядає таким чином:

 int WSASend (

 SOCKET s

 LPWSABUF lpBuffers

 DWORD dwBufferCount

 LPDWORD lpNumberOfBytesSent

 DWORD dwFlags

 LPWSAOVERLAPPED lpOverlapped

 LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionROUTINE

 );

 Розглянемо параметри цієї функції:

* s - сокет, через який відбуватиметься відправка даних;
* lpBuffers - структура або масив структур типу WSABUF. З цією структурою ви познайомилися, коли розглядали функцію connect. Ця ж структура використовувалася для відправки даних під час з'єднання;
* dwBufferCount - кількість структур в параметрі lpBuffers;
* lpNumberOfBytesSent - кількість переданих байт для завершеної операції введення/висновку;
* dwFlags - визначає метод відправки і може приймати такі ж значення, як і параметр dwFlags функції send;
* pOverlapped і pCompletionRoutine - задаються при використанні пере-критого введення/висновку (overlapped I/O). Це одна з моделей асинхронної роботи мережі, підтримуваної WinSock.

Якщо функція send (або WSASend) відпрацювала успішно, то вона поверне кількість відправлених байт, інакше - значення -1 (або константу SOCKET\_ERROR, яка рівна -1). Одержавши помилку, ви можете проаналізувати її за допомогою функції WSAGetLastError:

* WSAECONNABORTED - з'єднання було розірвано, або вийшов час очікування або відбулася інша помилка;
* WSAECONNRESET - видалений комп'ютер розірвав з'єднання, і вам необхідно закрити сокет;
* WSAENOTCONN - з'єднання не встановлене;
* WSAETIMEDOUT - час очікування відповіді вийшов.

 Для отримання даних використовуються функції recv і WSARecv (для другої версії WinSock). Функція recv виглядає таким чином:

 int recv (

 SOCKET s

 char FAR \* buf

 int len

 int flags

 );

 Параметри дуже схожі на ті, які описані для функції send:

* s - сокет, дані якого треба одержати;
* buf - буфер, в який будуть поміщені прийняті дані;
* len - довжина буфера в параметрі buf;
* flags - прапори, що визначають метод отримання. Тут можна указувати поєднання з наступних значень:
* 0 - прапори не вказані;
* MSG\_PEEK - рахувати дані з системного буфера без видалення. За умовчанням лічені дані стираються з системного буфера;
* MSG\_OOB - обробити термінові дані out of band.

 Використовувати прапор MSG\_PEEK не рекомендується, тому що ви можете зустрітися з безліччю непередбачуваних проблем. В цьому випадку функцію recv доведеться викликати другий раз (без цього прапора), щоб видалити дані з системного буфера. При повторному прочитуванні в буфері може опинитися більше даних, ніж вперше (за цей час комп'ютер може одержати на порт додаткові пакети), і ви ризикуєте обробити дані двічі або не обробити щось взагалі. Ще одна проблема полягає в тому, що системна пам'ять не очищається, і з кожним разом залишається менше простору для даних, що поступають. Саме тому я рекомендую використовувати прапор MSG\_PEEK тільки при необхідності і дуже акуратно.

 Функція WSARecv виглядає таким чином:

 int WSARecv (

 SOCKET s

 LPWSABUF lpBuffers

 DWORD dwBufferCount

 LPDWORD lpNumberOfBytesRecvd

 LPDWORD lpFlags

 LPWSAOVERLAPPED lpOverlapped

 LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionROUTINE

 );

 Тут також впадає в очі схожість в параметрах з функцією WSASend. Давайте розглянемо їх призначення:

* s - сокет, через який відбуватиметься отримання даних;
* lpBuffers - структура або масив структур типу WSABUF. У ці буфери будуть поміщені одержані дані;
* dwBufferCount - кількість структур в параметрі lpBuffers;
* lpNumberOfBytesSent - кількість одержаних байт, якщо операції введення/висновку вже завершилися;
* dwFlags - визначає метод відправки і може приймати такі ж значення, як і параметр dwFlags функції recv. Але є один новий прапор - MSG\_PARTIAL. Його потрібно указувати для протоколів, орієнтованих на читання повідомлення в декілька прийомів. У разі вказівки цього прапора при кожному прочитуванні можна одержати тільки частину даних;
* pOverlapped і pCompletionRoutine - встановлюються при использова-нді перекритого введення/висновку (overlapped I/O). Це одна з моделей асинхронної роботи мережі, підтримуваної WinSock.

 Варто відмітити, що якщо ви використовуєте протокол, орієнтований на передачу повідомлень (UPD), і вказали недостатній розмір буфера, то будь-яка функція для отримання даних поверне помилку WSAEMSGSIZE. Якщо протокол потоковий (TCP), то така помилка не виникне, тому що одержувані дані кэшируются в системі і надаються додатку повністю. В цьому випадку, якщо вказаний недостатній буфер, то дані, що залишилися, можна одержати при наступному прочитуванні.

 Є ще одна цікава мережева функція, яка з'явилася в WinSock2. Якщо всі розглянуті в цьому розділі функції мережевої бібліотеки (без префікса WSA) існують не тільки в Windows, але і в UNIX-системах, то функція TransmitFile є розширенням Microsoft і працює тільки в Windows.

 Функція TransmitFile посилає по мережі цілий файл. Це відбувається достатньо швидко, тому що відправка йде через ядро бібліотеки. Вам не треба піклуватися про послідовне читання і перевіряти кількість відправлених даних, тому що це гарантується бібліотекою WinSock2.

 Функція виглядає таким чином:

 BOOL TransmitFile(

 SOCKET hSocket

 HANDLE hFile

 DWORD nNumberOfBytesToWrite

 DWORD nNumberOfBytesPerSend

 LPOVERLAPPED lpOverlapped

 LPTRANSMIT\_FILE\_BUFFERS lpTransmitBuffers

 DWORD dwFlags

 );

 Розглянемо її параметри:

* hSocket - сокет, через який потрібно відправити дані;
* hFile - покажчик на відкритий файл, дані якого треба відправити;
* nNumberOfBytesToWrite - кількість тих, що відправляються з файлу байт. Якщо вказати про, то буде відправлений весь файл;
* nNumberOfBytesPerSend - розмір пакету для відправки. Якщо вказати 1024, то дані відправлятимуться пакетами в 1024 байт даних. Якщо вказати 0, то буде використане значення за умовчанням;
* lpOverlapped - використовується при перехресному введенні/висновку;
* lpTransmitBuffers - містить службову інформацію, яку треба послати до і після відправки файлу. За цими даними на приймаючій стороні можна визначити почало або закінчення передачі;
* dwFlags - прапори. Тут можна вказати наступні значення:
* TF\_DISCONNECT - закрити сокет після передачі даних;
* TF\_REUSE\_SOCKET - підготувати сокет для повторного використання;
* TF\_WRITE\_BEHIND - завершити роботу, не чекаючи підтвердження про отримання даних з боку клієнта.

 Параметр lpTransmitBuffers має тип структури наступного вигляду:

 typedef Struct \_TRANSMIT\_FILE\_BUFFERS {

 PVOID Head;

 DWORD HeadLength;

 PVOID Tail;

 DWORD TailLength;

 } TRANSMIT\_FILE\_BUFFERS;

 У цієї структури наступні параметри:

* Head - покажчик на буфер, що містить дані, які треба послати клієнту до почала відправки файлу;
* HeadLength - розмір буфера Head;
* Tail - покажчик на буфер, що містить дані, які треба послати клієнту після завершення відправки файлу;
* TailLength - розмір буфера Tail.

 **Завершення з'єднання**

 Для завершення сеансу спочатку необхідно проінформувати партнера, з яким відбувалося з'єднання, про закінчення передачі даних. Для цього використовується функція shutdown, яка виглядає таким чином:

 int shutdown (

 SOCKET s

 int how

 );

 Перший параметр - це сокет, з'єднання якого необхідно закрити. Другий параметр може приймати одне з наступних значень:

* SD\_RECEIVE - заборонити будь-які функції прийому даних. На протоколи нижнього рівня цей параметр не діє. Якщо використовується потоковий протокол (наприклад, TCP) і в черзі є дані, чекаючі читання функцією recv, або вони прийшли пізніше, то з'єднання скидається. Якщо використовується UDP-протокол, то повідомлення продовжують поступати;
* SD\_SEND - заборонити всі функції відправки даних;
* SD\_BOTH - заборонити прийом і відправку даних.

 Після того, як партнер проінформований про завершення роботи, можна закривати сокет. Для цього використовується функція closesocket, яка виглядає так:

 int closesocket (

 SOCKET s

 );

 Після цього вказаний як єдиний параметр сокет буде закритий. Якщо ви спробуєте використовувати його в якій-небудь функції, то одержите помилку WSAENOTSOCK - дескриптор не є сокетом. Будь-які пакети, чекаючі відправку, уриваються або відміняються.

 **Принцип роботи протоколів без установки з'єднання**

 Все описане вище відноситься до протоколів з установкою з'єднання між клієнтом і сервером (протокол TCP), але існують протоколи без установки з'єднання (наприклад, UDP). Там не потрібна функція connect, а прийом і передача даних відбуваються по-іншому. Я спеціально не зачіпав цю тему, щоб ви не заплуталися у функціях і їх призначенні.

 При роботі з протоколами, що не вимагають з'єднання, на сервері досить викликати функцію socket, щоб пов'язати сокет з портом і адресою (зв'язати сокет і bind). Після цього не можна викликати функції listen або accept, тому що сервер одержує дані від клієнта без установки з'єднання. Натомість потрібно просто чекати приходу даних за допомогою функції recvfrom, яка виглядає таким чином:

 int recvfrom (

 SOCKET s

 char FAR\* buf

 int len

 int flags

 struct sockaddr FAR\* from

 int FAR\* fromlen

 );

 Перші чотири параметри такі ж, як і у функції recv. Параметр from указує на структуру sockaddr, в якій зберігатиметься IP-адреса комп'ютера, з якого пришли дані. У параметрі fromlen зберігається розмір структури.

 У другій версії WinSock з'явилася функція WSARecvFrom, яка схожа на WSARecv, тільки додані параметри recv і fromlen:

 int WSARecvFrom (

 SOCKET s

 LPWSABUF lpBuffers

 DWORD dwBufferCount

 LPDWORD lpNumberOfBytesRecvd

 LPDWORD lpFlags

 struct sockaddr FAR \* lpFrom

 LPINT lpFromlen

 LPWSAOVERLAPPED lpOverlapped

 LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionROUTINE

 );

 З погляду клієнта все теж дуже просто. Достатньо тільки створити сокет, і можна безпосередньо направляти дані. Для передачі даних по мережі використовується функція sendto:

 int sendto (

 SOCKET s

 const char FAR \* buf

 int len

 int flags

 const struct sockaddr FAR \* to

 int tolen

 ;

 Перші чотири параметри відповідають тим, що розглядалися у функції send. Параметр to - це структура типу sockaddr. Вона містить адресу і порт комп'ютера, якому потрібно передати дані. Оскільки у нас немає з'єднання між клієнтом і сервером, то ця інформація повинна указуватися прямо у функції передачі даних. Останній параметр tolen - це розмір структури to.

 Починаючи з другої версії, ми можемо користуватися функцією WSASendTo. У неї параметри такі ж, як і у WSASend, тільки додані два нових - lрTо і iToLen, що зберігають відповідно структуру з адресою одержувача і її розмір.

 int WSASendTo (

 SOCKET S

 LPWSABUF lpBuffers

 DWORD dwBufferCount

 LPDWORD lpNumberOfBytesSent

 DWORD dwFlags

 const struct sockaddr FAR \* lpTo

 int iToLen

 LPWSAOVERLAPPED lpOverlapped

 LPWSAOVERLAPPED\_COMPLETION\_ROUTINE lpCompletionROUTIME

 );

 Як бачите, робота з протоколами, що не вимагають з'єднання, ще простіше. Не треба викликати функції прослуховування порту і з'єднання з сервером. Якщо ви розберетеся з роботою протоколу TCP, то робота UDP вам буде вже зрозуміла.

# Обробка даних, що приймаються

 Всі прийняті по мережі дані слід ретельно верифицировати. Якщо необхідно розділити доступ до певних можливостей по паролю, то я рекомендую насамперед контролювати права на виконання команди. Після цього перевіряйте коректність вказаної команди і передані параметри.

 Допустимо, що клієнт запрошує у сервера які-небудь файли. Якщо ви спочатку перевірите правильність вказівки шляху і імені файлу і у разі невдачі виведете повідомлення про помилку, то хакер знатиме, що файлу в системі немає. Іноді хакеру цієї інформації може опинитися достатньо для пошуку шляху проникнення в систему. Саме тому спочатку потрібно перевіряти право на виконання команди, а потім вже розбирати параметри і оцінювати їх коректність.

 Якщо є можливість, то команди краще перевіряти жорстко. Наприклад, команду get filename необхідно перевіряти так, щоб перші три букви складали слово "get". Не можна робити пошук символів "get" у всьому одержаному тексті, оскільки для хакера відкривається безліч можливостей передати невірні дані. Більшість зломів відбуваються із-за неправильного аналізу одержаних даних і передачі некоректних даних серверу.

 Якщо ви розробляєте протокол обміну командами між клієнтом і сервером, то робіть так, щоб команда передавалася на самому початку, а всі параметри йшли в кінці. Допустимо, що у вас є команда get. Вона може працювати в двох режимах:

* Забрати файл із стороннього сервера - GET ім'я файлу FROM Адреса
* Забрати файл з клієнта, який підключився до сервера - GET Ім'я файлу

 Перша команда з погляду безпеки неефективна. Для визначення наявності ключового слова FROM доведеться робити пошук по рядку. Цього робити не можна. Всі ключові слова бажано шукати в жорстко певній позиції. В даному випадку першу команду бажано перетворити до наступного вигляду:

 GET FROM Ім'я файлу, Адреса

 В цьому випадку ключові слова йдуть на початку команди, і ви жорстко можете визначити їх наявність. Якщо хакер спробує використовувати неправильні параметри, то у нього нічого не вийде.

 Якщо передавані дані різноманітні, але можуть бути підведені під якийсь шаблон, то обов'язково використовуйте його при контролі. Це допоможе вам зробити додаткову перевірку коректності даних, але не може забезпечити повний захист. Саме в шаблонній перевірці програмісти найчастіше припускаються помилки. Чим складніші перевірка або шаблон, тим важче врахувати всі обмеження на передавані дані. Перш ніж використовувати програму в бойових умовах або в комерційних цілях, рекомендується приділити тестуванню цієї ділянки максимально можливий час. Бажано, щоб програму тестувала стороння людина, тому що тільки кінцевий користувач або хакер введе ті дані, про які ви навіть не підозрювали і не думали, що їх взагалі можна використовувати.

 Завдання ускладнюється, якщо за цими даними відбуватиметься доступ до файлової системи. Це може привести до нерегламентованого доступу до вашого диска зі всіма витікаючими наслідками. Коли як параметр указується шлях до файлу, то його легко перевірити за шаблоном, але дуже легко помилитися. Більшість програмістів просто перевіряють початок шляху, але це помилка.

 Допустимо, що у вас відкритий доступ тільки до теки interpub на диску З:. Якщо перевіряти тільки почало шляхи, то хакер зможе без проблем написати ось такий шлях:

 c:\interpub\..\winnt\system32\cmd.eхе\

 Тут, завдяки подвійній крапці, хакер виходить з теки interpub і дістає доступ до всього диска, у тому числі і системних файлів.

 Перш ніж писати перевірку за шаблоном, ви повинні ознайомитися зі всіма його винятковими ситуаціями. І ще раз нагадую, що ви повинні максимально тестувати програму навіть з найнеймовірнішими параметрами. Користувачі непередбачувані, особливо недосвідчені, а хакери достатньо розумні і вивчають систему з усіх боків, навіть з тих, про яких ви не підозрюєте.

# Прийом і передача даних

 Ви вже познайомилися в теорії і на практиці, як приймати і передавати дані між комп'ютерами. Але мистецтво хакера полягає в тому, щоб правильно використовувати різні методи і режими передачі. Існує два режими роботи сокетів, і ви повинні навчитися правильно їх використовувати, тому що це підвищить ефективність і швидкість ваших програм.

 Застосовують наступні режими мережевого введення/висновку (прийом/передача даних):

 1. Блокуючий (синхронний) - при виклику функції передачі програма зупиняє виконання і чекає завершення операції.

 2. Не блокуючий (асинхронний) - після виклику функції програма продовжує виконання незалежно від того, закінчена операція прийому/передачі чи ні.

 При описі функцій ми вже стикалися з цими поняттями *(див. разд. 4.6.5 і 4.6.6),* а зараз зупинюся на них детальніше, тому що дякуючи їм можна значно підвищити швидкість роботи і максимально використовувати ресурси.

 За умовчанням створюються блокуючі сокети, тому у всіх прикладах, які вже розглядалися в цьому розділі, використовувався синхронний режим як найбільш простій. В цьому випадку доводиться створювати потоки, усередині яких працюють мережеві функції, щоб головне вікно програми не блокувалося і реагувало на події від користувача.

 Але це не найголовніша проблема. Простота і надійність - несумісні поняття. Допустимо, що був виклик функції recv, але з якихось причин вона не повернула дані. В цьому випадку вона залишиться заблокованою навіки, і сервер більше не реагуватиме на дії користувача. Щоб уникнути цієї проблеми деякі програмісти перед прочитуванням даних перевіряють їх коректність за допомогою виклику функції recv з прапором MSG\_PEEK. Але ви вже знаєте, що це не безпечно, і довіряти таким даним не стоїть. До того ж цей метод навантажує систему зайвими перевірками буфера прийому на наявність даних.

 Неблокуючі сокети складніше в програмуванні, але позбавлені описаних недоліків. Щоб перевести сокет в асинхронний режим, потрібно скористатися функцією ioctlsocket, яка виглядає так:

 int ioctisocket (

 SOCKET s

 long cmd

 u\_long FAR\* argp

 );

 У цієї функції три параметри:

* сокет, режим якого треба змінити;
* команда, яку необхідно виконати;
* параметр для команди.

 Зміна режиму блокування відбувається при вказівці як команда константи FIONBIO. При цьому, якщо параметр команди має нульове значення, то використовуватиметься блокуючий режим, інакше - неблокуючий.

 Давайте подивимося на приклад створення сокета і перекладу його в неблокуючий режим:

 SOCKET s;

 unsigned long ulMode;

 s = socket(AS\_INET, SOCK\_STREAM, 0);

 ulMode = 1;

 ioctisocket(s, FIONBIO (unsigned long\*)&ulMode);

 Тепер всі функції прийому/передачі завершуватимуться помилкою. Це нормальна реакція, і ви повинні це враховувати при створенні мережевих додатків, що працюють в неблокуючому режимі. Якщо функція введення/висновку повернула помилку WSAEWOULDBLOCK, то це не означає неправильну передачу. Все пройшло успішно, просто використовується неблокуючий режим. Якщо ж дійсно відбувся збій, то ми одержимо помилку, відмінну від WSAEWOULDBLOCK.

 У неблокуючому режимі функція recv не чекатиме прийому даних, а просто поверне помилку WSAEWOULDBLOCK. Тоді як нам дізнатися, що дані поступили на порт? Деякі запускають цикл з постійним викликом функції recv, поки вона не поверне дані. Але це недоцільно, тому що відбувається блокування додатку і надмірно завантажується процесор.

 Звичайно ж, ви можете в циклі між перевірками виконувати якісь дії і тим самим використовувати процесор під час очікування з користю, але я не розглядатиму цей варіант, тому що є спосіб краще.

 **Функція *select***

 Ще в першій версії Winsock була дуже цікава можливість управління сокетами, що не блокуються. Для цього використовується функція select:

 int select (

 int nfds

 fd\_set FAR \* readfds

 fd\_set FAR \* writefds

 fd\_set FAR \* exceptfds

 const struct timeval FAR \* timeout

 );

 Функція повертає кількість готових до використання дескрипторів Socket.

 Тепер розглянемо параметри цієї функції:

* nfds - ігнорується і служить тільки для сумісності з моделлю сокетів Берклі;
* readfds - можливість читання (структура типу fd\_set);
* writefds - можливість запису (структура типу fd\_set);
* exceptfds - важливість повідомлення (структура типу fd\_set);
* timeout - максимальний час очікування або null для блокування подальшої роботи (чекати нескінченно).

 Структура fd\_set - набір сокетів, від яких потрібно чекати дозвіл на виконання певної операції. Наприклад, якщо вам потрібно дочекатися приходу даних на один з двох сокетів, то ви можете зробити наступне:

* додати в набір fd\_set два вже створені сокети;
* запустити функцію select і як другий параметр вказати набір з сокетами.

 Функція select чекатиме дані вказаний час, після чого можна прочитати дані з сокета. Але дані можуть прийти тільки на один з двох сокетів. Як дізнатися, на якій саме? Спершу за допомогою функції FD\_ISSET потрібно обов'язково перевірити, чи входить сокет в набір.

 При роботі із структурою типу fd\_set вам знадобляться наступні функції:

* FD\_ZERO - очищає набір. Перш ніж додавати в набір нові сокети, обов'язково викликайте цю функцію, щоб проїніциалізіровать набір. У цієї функції тільки один параметр - покажчик на змінну типу fd\_set;
* FD\_SET - додає сокет в набір. У функції два параметри - сокет, який потрібно додати, і змінна типу fd\_set, у набір якої потрібно додати сокет;
* FD\_CLR - видаляє сокет з набору. У цієї функції два параметри - сокет, який треба видалити, і набір, з якого походитиме видалення;
* FD\_ISSET - перевіряє, чи входить сокет, визначений в першому параметрі, в набір типу fd\_set, вказаний як другий параметр.

 **Простий приклад використання функції *select***

 Тепер застосовний все сказане на практиці. Відкриємо приклад TCPServer і після створення сокета додайте наступний код:

 ULONG ulBlock;

 ulBlock = 1;

 if (ioctlsocket(sServerListen, FIONBIO &ulBlock)== SOCKET\_ERROR)

 {

 return 0;

 }

 Таким чином сокет переводиться в асинхронний режим. Тепер спробуємо запустити приклад. Спочатку ви повинні побачити два повідомлення "Bind OK" і "Listen OK", після чого програма поверне помилку "Accept filed". У асинхронному режимі функція accept не блокує роботу програми, а значить, не чекає з'єднання. В цьому випадку, якщо в черзі немає чекаючих підключення клієнтів, то функція поверне помилку WSAEWOULDBLOCK.

 Щоб позбавитися цього недоліку, потрібно підкоригувати цикл очікування з'єднання (нескінченний цикл while, який йде після виклику функції listen). Для асинхронного варіанту він повинен виглядати, як в лістингу.

 **Цикл очікування з'єднання**

 FD\_SET ReadSet;

 int ReadySock;

 while (1)

 {

 FD\_ZERO(&ReadSet);

 FD\_SET(sServerListen &ReadSet);

 iSize = sizeof(clientaddr);

 sClient = accept(sServerListen

 (struct sockaddr \*)&clientaddr &iSize);

 if (sClient == INVALID\_SOCKET)

 {

 MessageBox(0, "Accept filed", "Error", 0);

 break;

 hThread = CreateThread(NULL, 0, ClientThread

 (LPVOID)sClient, 0 &dwThreadId);

 if (hThread == NULL)

 {

 MessageBox(0, "Create thread filed", "Error", 0);

 break;

 CloseHandle(hThread);

 }

 Перед циклом додані дві змінні: ReadSet типу FD\_SET для зберігання набору сокетів і ReadySock типу int для зберігання кількості готових до використання сокетів. На даний момент у нас тільки один сокет, тому цю змінну поки використовувати не будемо.

 На самому початку циклу обнуляється набір за допомогою функції FD\_ZERO і додається створений сокет, який чекає підключення. Після цього викликається функція select. Для неї вказаний тільки другий параметр, а вся решта значень - нульові. Якщо вказаний другий параметр, то функція чекає можливості читання для сокетів з набору. Параметр "час очікування" теж встановлений в нуль, що відповідає нескінченному очікуванню.

 Отже, сокет сервера чекає підключення і готовий до читання. Коли від клієнта надійде запит на підключення, сокет прикмет його. Але перш ніж виконувати якісь дії, необхідно перевірити входження сокета в набір за допомогою функції FD\_ISSET.

 Решта коду не змінилася. Ми приймаємо вхідне з'єднання за допомогою функції accept, одержуємо новий сокет для роботи з клієнтом і зберігаємо його в змінній sClient, Після цього створюється новий потік, в якому відбувається обмін даними з клієнтом.

 Запустіть приклад і переконайтеся, що він працює коректно. Тепер немає помилок, і програма терпляче чекає з'єднання з боку клієнта.

 Виникає питання, в якому режимі працює сокет sClient, який створений функцією accept. Я вже говорив, що за умовчанням сокети працюють в блокуючому режимі, і ми не змінювали це значення. Давайте перевіримо. Запустіть додаток і спробуйте під'єднатися до сервера програмою TCPClient, яка приведена*.* Клієнт відправить дані, потім одержить відповідь " Command get OK " і після цього видасть помилку. Чому? Тому що ми в нескінченному циклі намагаємося одержати дані від клієнта, і перша спроба вдала, а друга - ні. Означає, сокет sClient працює в тому ж режимі, що і сокет sServerListen.

 За допомогою функції select можна позбавитися другого потоку, який використовується для обміну даними між клієнтом і сервером. Крім цього, в прикладі в нинішньому вигляді для обробки декількох клієнтів потрібно створювати безліч потоків. Завдяки функції select можна все це зробити без потоків, набагато простіше і ефективніше. Але до цього я повернуся у розділі *6,* де розглядатимуться цікаві алгоритми.

 **Використання сокетів через події Windows**

 Функція select введена в бібліотеку WinSock для сумісності з аналогічними бібліотеками інших платформ. Для програмування в Windows могутнішою є функція WSAAsyncSelect, яка дозволяє відстежувати стан сокетів за допомогою повідомлень Windows. Таким чином, ви зможете одержувати повідомлення у функції WndProc, і немає необхідності заморожувати роботу програми для очікування доступності сокетів.

 Функція виглядає таким чином:

 int WSAAsyncSelect (

 SOCKET s

 HWND hWnd

 unsigned int wMsg

 long lEvent

 );

 Розглянемо кожен параметр:

* s - сокет, події якого необхідно ловити;
* hWnd - вікно, якому посилатимуться події при виникненні мережевих повідомлень. Саме біля цього вікна (або батьківського) повинна бути функція WndProc, яка одержуватиме повідомлення;
* wMsg - повідомлення, яке відсилатиметься вікну. По його типу можна визначити, що ця подія мережі;
* lEvents - бітова маска мережевих подій, які нас цікавлять. Цей параметр може приймати будь-яку комбінацію з наступних значень:
* FD\_READ - готовність до читання;
* FD\_WRITE - готовність до запису;
* FD\_OOB - отримання термінових даних;
* FD\_ACCEPT - підключення клієнтів;
* FD\_CONNECT - з'єднання з сервером;
* FD\_CLOSE - закриття з'єднання;
* FD\_QOS - зміни сервісу QoS (Quality of Service);
* FD\_GROUP\_QOS - зміна групи QoS.

 Якщо функція відпрацювала успішно, то вона поверне значення більше нуля, якщо відбулася помилка - SOCKET\_ERROR.

 Функція автоматично переводить сокет в неблокуючий режим, і немає сенсу викликати функцію ioctlsocket.

 Ось простий приклад використання WSAAsyncSelect:

 WSAAsyncSelect(s, hWnd, wMsg FD\_READ|FD\_WRITE);

 Після виконання цієї строчки коду вікно hWnd одержуватиме подію wMsg кожного разу, коли сокет s буде готовий приймати і відправляти дані. Щоб відмінити роботу події, необхідно викликати цю ж функцію, але як четвертий параметр вказати 0:

 WSAAsyncSelect(s, hWnd, 0, 0);

 В даному випадку необхідно правильно вказати перші два параметри і обнулити останній. Вміст третього параметра не має значення, тому що подія не відправлятиметься, і можна вказати нуль. Якщо вам потрібно просто змінити типи подій, то можете викликати функцію з новими значеннями четвертого параметра. Немає сенсу спочатку обнуляти, а потім встановлювати наново.

 Для кожного сокета можна призначити тільки одне повідомлення на різні події. Це означає, що не можна по події FD\_READ вікну посилати одне повідомлення, а по FD\_WRITE - інше.

 Перш ніж приступати до розгляду прикладу, треба розібратися з параметрами, які передаватимуться у функцію WndProc при виникненні події. Пригадаємо, як виглядає ця функція:

 LRESULT CALLBACK WndProc(

 HWND hWnd

 UINT message

 WPARAM wParam

 LPARAM lParam)

 Параметри wParam і lParam містять допоміжну інформацію (залежно від події). Для подій мережі в параметрі wParam зберігається дескриптор сокета, на якому відбулася подія. Таким чином, вам не треба зберігати масив створених сокетів, їх завжди можна одержати в події.

 Параметр lParam складається з двох слів: молодше визначає подію, а старше - код помилки. Ось тепер можна переходити до розгляду реального прикладу. Створіть новий додаток **Win32 Application**, назвіть проект WSASel. Відкрийте файл WSASel.cpp і підкоригуйте функцію \_tWinMain. Як завжди, весь код потрібно додати до циклу обробки повідомлень. Всю функцію ви можете побачити в лістингу.

 **Функція \_tWinMain**

 int APIENTRY \_tWinMain(HINSTANCE hInstance

 HINSTANCE hPrevInstance

 LPTSTR lpCmdLine

 int nCmdShow)

 {

 // TODO: Place code here.

 MSG msg;

 HACCEL hAccelTable;

 // Initialize global strings

 LoadString(hInstance IDS\_APP\_TITLE, szTitle MAX\_LOADSTRING);

 LoadString(hInstance IDC\_WSASEL, szWindowClass MAX\_LOADSTRING);

 MyRegisterClass(hInstance);

 // Perform application initialization:

 if (!InitInstance (hInstance, nCmdShow))

 return FALSE;

 hAccelTable = LoadAccelerators(hInstance (LPCTSTR)IDC\_WSASEL);

 WSADATA wsd;

 if (WSAStartup(MAKEWORD(2,2) &wsd)!= 0)

 MessageBox(0, "Can't load WinSock", "Error", 0);

 return 0;

 SOCKET sServerListen, sClient;

 struct sockaddr\_in localaddr, clientaddr;

 HANDLE hThread;

 DWORD dwThreadId;

 int iSize;

 ULONG ulBlock;

 ulBlock = 1;

 if (ioctlsocket(sServerListen, FIONBIO &ulBlock)== SOCKET\_ERROR)

 return 0;

 localaddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);

 localaddr.sin\_family = AF\_INET;

 localaddr.sin\_port = htons(5050);

 if (bind(sServerListen (struct sockaddr \*)&localaddr,

 sizeof(localaddr)) == SOCKET\_ERROR)

 MessageBox(0, "Can't bind", "Error", 0);

 return 1;

 WSAAsyncSelect(sServerListen, hWnd WM\_USER+1, FD\_ACCEPT);

 listen(sServerListen, 4);

 // Main message loop:

 while (GetMessage(&msg, NULL, 0, 0))

 if (!TranslateAccelerator(msg.hwnd, hAccelTable &msg))

 TranslateMessage(&msg);

 DispatchMessage(&msg);

 closesocket(sServerListen);

 WSACleanup();

 return (int) msg.wParam;

 }

 Завдяки використанню функції WSAAsyncSelect весь код (без додаткових потоків) можна розташувати прямо у функції \_tWinMain.

 Код практично нічим не відрізняється від того, що був в проекті TCPServer*.* Єдине, перед запуском прослуховування (listen) викликається функція WSAAsyncSelect, щоб вибрати створений сокет і перевести його в асинхронний режим. Тут указуються наступні параметри:

* sServerListen - змінна, яка указує на створений серверний сокет;
* hWnd - покажчик на головне вікно програми, і саме йому передаватимуться повідомлення;
* WM\_USER+1 - всі призначені для користувача повідомлення повинні бути більше константи WM\_USER. Менші значення можуть використовуватися системою і викликати конфлікт. Я використовував таку конструкцію, щоб явно показати необхідність використання такого повідомлення. У реальних додатках я раджу створювати для цього константу із зрозумілим ім'ям і використовувати її. Це можна зробити таким чином: #define WM\_NETMESSAGE WM\_USER+1;
* FD\_ACCEPT - подія, яку потрібно обробляти. Що може робити серверний сокет? Приймати з'єднання з боку клієнта. Саме ця подія нас цікавить.

 Найголовніше відбуватиметься у функції WndProc. Почало функції, де потрібно додати код, показано в лістингу.

 **Обробка мережевих повідомлень у функції WndProc**

 LRESULT CALLBACK WndProc(HWND hWnd, UINT message

 WPARAM wParam, LPARAM lParam)

 {

 int wmId, wmEvent;

 PAINTSTRUCT ps;

 HDC hdc;

 SOCKET ClientSocket;

 int ret;

 char szRecvBuff[1024], szSendBuff[1024];

 switch (message)

 case WM\_USER+1:

 switch (WSAGETSELECTEVENT(lParam))

 case FD\_ACCEPT:

 break;

 case FD\_READ:

 ret = recv(wParam, szRecvBuff, 1024, 0);

 if (ret == 0)

 break;

 else if (ret == SOCKET\_ERROR)

 MessageBox(0, "Recive data filed",

 "Error", 0);

 break;

 szRecvBuff[ret]= '\0';

 strcpy(szSendBuff, "Command get OK");

 ret = send(wParam, szSendBuff

 sizeof(szSendBuff), 0);

 break;

 case FD\_WRITE:

 //Ready to send data

 break;

 case FD\_CLOSE:

 closesocket(wParam);

 break;

 case WM\_COMMAND:

 ...

 Тут на самому початку доданий новий оператор case, який перевіряє, чи рівне спіймане повідомлення шуканому мережевому повідомленню WM\_USER+1. Якщо ця мережева подія, то запускається перебір мережевих подій. Для цього використовується оператор switch, який порівнює вказане в дужках значення з подіями, що поступають:

 switch (WSAGETSELECTEVENT(lParam))

 Як відомо, в параметрі lParam знаходяться код помилки і тип події. Щоб одержати подію, використовується функція WSAGETSELECTEVENT. А потім перевіряються необхідні нам події. Якщо відбулося з'єднання з боку клієнта, то виконується наступний код:

 case FD\_ACCEPT:

 ClientSocket = accept(wParam, 0, 0);

 WSAAsyncSelect(ClientSocket, hWnd WM\_USER+1,

 FD\_READ | FD\_WRITE | FD\_CLOSE);

 break;

 Спочатку приймається з'єднання за допомогою функції accept. Результатом буде сокет, за допомогою якого можна працювати з клієнтом. З цього сокета теж потрібно ловити події, тому викликаємо функцію WSAAsyncSelect. Щоб не плодити повідомлення, використовуємо як третій параметр значення WM\_USER+1. Це не викличе конфліктів, тому що серверний сокет обробляє тільки подію FD\_ACCEPT, а у клієнтського нас цікавлять події читання, запису даних і закриття сокета.

 Коли сервера дійдуть дані, поступить повідомлення WM\_USER+1, а функція WSAGETSELECTEVENT(lParam) поверне значення FD\_READ. В цьому випадку читаються дані, що прийшли, а клієнту посилається текст "Command get OK":

 case FD\_READ:

 ret = recv(wParam, szRecvBuff, 1024, 0);

 if (ret == 0)

 break;

 else if (ret == SOCKET\_ERROR)

 MessageBox(0, "Recive data filed",

 "Error", 0);

 break;

 szRecvBuff[ret]= '\0';

 strcpy(szSendBuff, "Command get OK");

 ret = send(wParam, szSendBuff, sizeof(szSendBuff), 0);

 break;

 Це той же самий код, який використовувався в додатку TCPServer для обміну даними між клієнтом і сервером. Я навмисно не вносив змін, щоб сервер можна було протестувати програмою TCPClient.

 По події FD\_WRITE нічого не відбувається, а тільки стоїть коментар. По події FD\_CLOSE закривається сокет.

 Розглянутий приклад з використанням функції select може працювати тільки з одним клієнтом. Щоб додати можливість одночасної обробки декількох з'єднань, необхідно сформувати масив потоків, використовуваних для прийому/передачі даних. У розділі *6* я приведу приклад з використанням функції select, який і без масивів потоків буде позбавлений від цих недоліків.

 Функція WSAAsyncSelect простіше в програмуванні і спочатку дозволяє обробляти безліч клієнтів. Ну, а найголовніше - немає жодного додаткового потоку.

 Щоб протестувати приклад, спочатку запустите програму-сервер WSASel, а потім - программу-клієнт TCPClient.

 Хочу звернути вашу увагу, що обмін інформацією відбувається асихронно. Відправку і прийом великої кількості даних потрібно буде робити порціями.

 Допустимо, що клієнт повинен передати серверу 1 Мбайт даних. Звичайно ж, за один прийом це зробити нереально. Тому на стороні сервера ви повинні діяти таким чином:

* сервер повинен дізнатися (клієнт повинен повідомити сервер) кількість передаваних даних;
* сервер повинен виділити необхідний об'єм пам'яті або, якщо кількість даних дуже велика, створити тимчасовий файл;
* при отриманні події FD\_READ зберігати прийняті дані в буфері або у файлі. Обробляти подію, поки дані не будуть одержані повністю, і клієнт не пришле певну послідовність байт, що визначає завершення передачі даних.

 Подібним способом повинна відбуватися відправка з клієнта:

* повідомити серверу кількість даних, що відправляються;
* відкрити файл, з якого читатимуться дані;
* послати першу порцію даних, решта даних - по події FD\_WRITE;
* по завершенню відправки послати серверу послідовність байт, що визначає завершення передачі даних.

 Використання повідомлень Windows дуже зручно, але ви втрачаєте сумісність з UNIX-системами, де повідомлення реалізовані по-іншому і немає функції WSAAsyncSelect. Тому при перенесенні такої програми на іншу платформу виникнуть великі проблеми і доведеться переписати дуже багато код. Але якщо перенесення не планується, то я завжди використовую WSAAsyncSelect, що дозволяє добитися максимальної продуктивності і зручності програмування.

 **Асинхронна робота через об'єкт події**

 Якщо в програмі немає процедури обробки повідомлень, то можна скористатися об'єктами подій. В цьому випадку алгоритм роботи буде декілька інший:

 1. Створити об'єкт події за допомогою функції WSACreateEvent.

 2. Вибрати сокет за допомогою функції WSAEventSelect.

 3. Чекати подія за допомогою функції WSAWaitForMultipleEvents.

 Давайте детально розглянемо всі функції, необхідні для роботи з об'єктами подій.

 Насамперед слід створити подію за допомогою функції WSACreateEvent. Функції не треба не передавати ніяких параметрів, вона просто повертає нову подію типу WSAEVENT:

 WSAEVENT WSACreateEvent(void);

 Тепер потрібно пов'язати сокет з цим об'єктом і вказати події, які нам потрібні. Для цього використовується функція WSAEventSelect:

 int WSAEventSelect (

 SOCKET s

 WSAEVENT hEventObject

 long lNetworkEvents

 )

 Перший параметр - це сокет, події якого нас цікавлять. Другий параметр - об'єкт події. Останній параметр - це необхідні події. Як останній параметр можна указувати ті ж константи, що розглядалися для функції WSAAsyncSelect (всі вони починаються з префікса FD\_).

 Раніше ви вже зустрічалися з функціями WaitForSingleObject і WaitForMultipleObjects, які чекають настання події типу HANDLE. Для мережевих подій використовується схожа функція з ім'ям WSAWaitForMultipleEvents:

 DWORD WSAWaitForMultipleEvents (

 DWORD cEvents

 const WSAEVENT FAR \*lphEvents,

 BOOL fWaitAll

 DWORD dwTimeOUT

 BOOL fAlertable

 );

 Давайте розглянемо кожен параметр:

* cEvents - кількість об'єктів подій, зміна стану яких потрібно чекати. Щоб дізнатися максимальне число, скористайтеся константою WSA\_MAXIMUM\_WAIT\_EVENTS;
* lphEvents - масив об'єктів подій, які потрібно чекати;
* fWaitAll - режим очікування подій. Якщо вказано TRUE, то функція чекає, поки всі події не спрацюють, інакше - після першого передає управління програмі;
* dwTimeOUT - часовий інтервал в мілісекундах, протягом якого потрібно чекати події. Якщо в цьому тимчасовому інтервалі не виникло події, то функція повертає значення WSA\_WAIT\_TIMEOUT. Якщо потрібно чекати нескінченно, то можна вказати константу WSA\_INFINITE;
* fAlertable - параметр використовується при перехресному введенні/висновку, яке я не розглядаю в цій книзі, тому вказаний FALSE.

 Щоб дізнатися, яка подія з масиву подій спрацювала, потрібно відняти з поверненого функцією WSAWaitForMultipleEvents значення константу WSA\_WAIT\_EVENT\_0.

 Перш ніж викликати функцію WSAWaitForMultipieEvents, всі події в масиві повинні бути порожніми. Якщо хоч би одне з них буде зайняте, то функція відразу поверне управління програмі, і не буде очікування. Після виконання функції відпрацьовані події стають зайнятими, і після обробки їх треба звільнити. Для цього використовується функція WSAResetEvent:

 BOOL WSAResetEvent (

 WSAEVENT hEvent

 );

 Функція очищає стан події, вказаної як єдиний параметр.

 Коли подія вже не потрібна, його необхідно закрити. Для цього використовується функція WSACloseEvent. Функції слід передати об'єкт події, який необхідно закрити:

 BOOL WSACloseEvent (

 WSAEVENT hEvent

 );

 Якщо закриття пройшло успішно, то функція повертає TRUE, інакше - FALSE.

# Робота з СОМ портом

 Мені за службовим обов'язком часто доводилося працювати з інтерфейсом RS-232. Так в офіційній документації називається СОМ-порт комп'ютера. Сучасне устаткування (контроллери, пристрої збору інформації і т.д.) працюють через цей порт. До будь-якого модему, навіть внутрішнього, звернення відбувається саме через СОМ-порт. А скільки існує зовнішніх пристроїв, що підключаються по цьому інтерфейсу, злічити неможливо.

 Робота з портами схожа на роботу з файлами. Давайте розглянемо простий приклад. Для цього створіть новий додаток **MFC Application** на основі діалогу з ім'ям COMport.

 У верхній частині вікна знаходиться випадний список Combo Box, в якому можна вибирати ім'я порту. Поряд із списком дві кнопки: для відкриття і закриття порту. Трохи нижче розташовані текстове поле для введення команди і кнопка для її відправки.

 У центрі вікна розташувалися елементи управління List Box: для відображення ходу роботи з портом і багаторядкове поле введення для відображення даних, що прийшли.

 Створіть подібний інтерфейс, і можна переходити до програмування. По натисненню кнопки **Open port** повинен виконуватися код з лістингу.

 **Відкриття порту**

 void CCOMportDlg::OnBnClickedOpenportButton()

 {

 if (hCom != INVALID\_HANDLE\_VALUE)

 OnBnClickedButton1();

 Sleep(300);

 char sPortName[10];

 cbPorts.GetWindowText(sPortName, 10);

 hCom = CreateFile(sPortName GENERIC\_READ | GENERIC\_WRITE,

 0, NULL OPEN\_EXISTING, 0, NULL);

 if (hCom == INVALID\_HANDLE\_VALUE)

 lLogList.AddString("Error opening port");

 else

 lLogList.AddString("Port successfully opened.");

 hThread = CreateThread(0, 0, ReadThread

 (LPVOID)this, 0, 0);

 dcb.BaudRate = CBR\_57600;

 dcb.ByteSize = 8;

 dcb.Parity = NOPARITY;

 dcb.StopBits = ONESTOPBIT;

 if (SetCommState(hCom &dcb))

 lLogList.AddString("Configuring OK");

 else

 lLogList.AddString("Configuring Error");

 }

 }

 Якщо спробувати відкрити порт двічі, то буде одержане повідомлення про помилку, тому насамперед потрібно провести цю перевірку. І якщо порт відкритий, то закрити його. Ця перевірка виконується у функції onBnClickedButton1, яку я покажу трохи пізніше, і вона викликатиметься при натисненні на кнопку Close **port.**

 Тепер одержимо ім'я вибраного порту і відкриємо його. Для цього використовується функція роботи з простими файлами CreateFile, тільки замість імені файлу указується ім'я порту.

 Якщо порт відкритий вдало, то виводиться відповідне повідомлення, і можна перейти до конфігурації параметрів з'єднання. Для цього спочатку одержите поточні настройки системи за допомогою функції GetCommState. Їй потрібно передати два параметри: покажчик на відкритий порт і покажчик на структуру типу DCB. Ця структура містить повну інформацію про параметри з'єднання і виглядає таким чином:

 typedef struct \_DCB {

 DWORD DCBlength; // розмір структури DCB

 DWORD BaudRate; // швидкість передачі даних в бодах

 DWORD fBinary: 1; // двійковий режим без перевірки кінця

 // рядки

 DWORD fParity: 1; // включити перевірку парність

 DWORD fOutxCtsFlow:1; // CTS-правління потоком виходу

 DWORD fOutxDsrFlow:1; // DSR-управління потоком виходу

 DWORD fDtrControl:2; // DTR-тип управління потоком швидкості

 // передачі даних

 DWORD fDsrSensitivity:1; // DSR-чутливість

 DWORD fTXContinueOnXoff:1; // стоп-сигнал продовжує виконання

 DWORD fOutX: 1; // старт/стоп-сигнал для управління

 // потоком, що виходить

 DWORD fInX: 1; // старт/стоп - сигнал для управління

 // вхідним потоком

 DWORD fErrorChar: 1; // включити перевірку погрішностей

 DWORD fNull: 1; // відкидати порожній потік даних

 DWORD fRtsControl:2; // RTS - управління потоком даних

 DWORD fAbortOnError:1; // перевіряти операції читання/запису

 DWORD fDummy2:17; // зарезервоване

 WORD wReserved; // зарезервоване

 WORD XonLim; // поріг чутливості сигналу старту

 WORD XoffLim; // поріг чутливості стоп-сигналу

 BYTE ByteSize; // кількість біт (звичайно 7 або 8)

 BYTE Parity; // парність байта

 BYTE StopBits; // стопові біти

 char XonChar; // вид сигналу старту в потоці

 char XoffChar; // вид стоп-сигналу в потоці

 char ErrorChar; // вид сигналу погрішності

 char EofChar; // сигнал закінчення потоку

 char EvtChar; // зарезервоване

 WORD wReserved1; // зарезервоване

 } DCB;

 Якщо неправильно вказані параметри, то дані не передаватимуться і не прийматимуться. Найголовніше - заповнити наступні поля:

* BaudRate - швидкість передачі даних (бит/с). Указується константа у вигляді CBR\_скорость, де швидкість повинна бути рівна швидкості, підтримуваній використовуваним пристроєм, наприклад, 56000;
* ByteSize - розмір передаваного байта (може бути 7 або 8);
* Parity - прапор перевірки парності;
* StopBits - стопові біти, можуть приймати значення ONESTOPBIT (один) ONE5STOPBITS (півтора) або TWOSTOPBITS (два).

 Решту параметрів можна залишити за умовчанням (ті, які повернула система). Але перш ніж указувати які-небудь параметри, обов'язково прочитайте документацію на апаратуру, з якою необхідно з'єднуватися. Я не зустрічав пристроїв, які підтримували б всі режими роботи. Наприклад, модем ZyXel Omni 56 K може підтримувати швидкість від 2400 до 56 000, і можна указувати значення тільки з цього діапазону.

 Крім цього, потрібне, щоб обидва пристрої (передавальне і приймаюче) були настроєні однаково (швидкість, розмір байта і т. д.), інакше дані не передаватимуться.

 Після конфігурації порту запускається потік, в якому ми нескінченно намагатимемося рахувати дані з порту. Це, звичайно ж, не ефективно, тому що зручніше використовувати повідомлення Windows, але для простого прикладу в повчальних цілях достатньо. Функція читання потоку ReadThread виглядає таким чином:

 DWORD \_\_stdcall ReadThread(LPVOID hwnd)

 {

 DWORD iSize;

 char sReceivedChar;

 while(true)

 ReadFile(hCom,&sReceivedChar,1,&iSize,0);

 SendDlgItemMessage((HWND)hwnd,IDC\_EDIT2,WM\_CHAR,

 sReceivedChar,0);

 }

 У цій функції ви можете побачити нескінченний цикл читання даних, яке виконується стандартною функцією читання з файлу - ReadFile.

 Тепер подивитеся на функцію закриття порту, яка викликатиметься по натисненню кнопки **Close port**:

 void CCOMportDlg::OnBnClickedButton1()

 {

 if (hCom == INVALID\_HANDLE\_VALUE)

 return;

 if (MessageBox("Close port?", "Warning" MB\_YESNO)== IDYES)

 TerminateThread(hThread, 0);

 CloseHandle(hCom);

 hCom = INVALID\_HANDLE\_VALUE;

 }

 }

 Перш ніж виконати закриття, треба перевірити змінну hCom. Можливо, що порт вже закритий або взагалі ніколи не відкривався. Якщо змінна містить неправильний покажчик, то слід просто вийти з функції.

 Якщо порт відкритий, то виводиться запит на підтвердження закриття порту. Якщо користувач підтвердить, то уривається потік, закривається покажчик порту і змінної hCom привласнюється значення INVALID\_HANDLE\_VALUE.

 І останнє, що належить додати в програму, - можливість відправки повідомлень. Для цього по натисненню кнопки **Send command** повинен виконуватися код з лістингу.

 **Функція відправки даних в порт**

 void CCOMportDlg::OnBnClickedSendcommandButton()

 {

 if (hCom == INVALID\_HANDLE\_VALUE)

 AfxMessageBox("Open port before send command");

 return;

 char sSend[10224];

 eSendCommand.GetWindowText(sSend, 1024);

 if (strlen(sSend)>0)

 lLogList.AddString(sSend);

 sSend[strlen(sSend)]= '\r';

 sSend[strlen(sSend)]= '\0';

 WriteFile(hCom, sSend, strlen(sSend) &iSize,0);

 hThread = CreateThread(0, 0, ReadThread

 (LPVOID)this, 0, 0);

 }

 }

 Спочатку перевіряється, чи відкритий порт. Якщо він вже закритий або ніколи не відкривався, то немає сенсу писати в нього дані. Після цього треба одержати дані для відправки, і якщо вони більше нуля, то додати в кінець рядка, що відправляється, символ завершення рядка (нульовий символ). Мені доводилося працювати з різним устаткуванням, і більшість типів вимагають в кінці команди відправляти символи кінця рядка і переведення каретки. Іноді буває досить тільки символу кінця рядка.

 Тепер перериваємо потік читання даних і записуємо в порт дані стандартною функцією роботи з файлами WriteFile. Після запису можна знову запускати потік читання.

 Якщо у вас є модем, то можете запустити програму і відкрити порт, на якому настроєний модем. Відправте команду ATDTxxxxx, де хххх - це номер телефону. Модем повинен буде почати набір вказаного номера телефону.