Глава 7. Практикум

7.1. Предисловие к главе

Для выполнения практических заданий предлагаемых в этой главе, необходимо иметь несколько объединенных TCP/IP-сетью компьютеров с операционной системой Windows XP. Кроме того, для разработки приложений на языке C++ требуется среда разработки Microsoft Visual Studio не ниже седьмой версии и доступ к соответствующей версии MSDN.

Каждая практическая работа состоит из нескольких заданий. Задания, как правило, связаны между собой и требуют последовательного выполнения. Практическая работа считается выполненной, если успешно выполнены все ее задания.

Все разрабатываемые приложения должны компилироваться в режиме Debug (отладчика) для возможности исследования процесса работки и остановки.

7.2. Практическая работа № 1. Сетевые утилиты

7.2.1. Цель и задачи работы

Целью работы является ознакомление с функциональными возможностями сетевых утилит операционной системы Windows.

В результате работы студент будет уметь определять характеристики TCP/IP-сети, тестировать соединения компьютеров в сети, использовать сетевые утилиты при отладке приложений.

7.2.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены во второй главе этого пособия. В качестве дополнительной литературы рекомендуются источники [5, 6, 7, 10].

7.2.3. Утилита ipconfig

Задание 1. Получите справку о параметрах утилиты **ipconfig.**

Задание 2. Получите короткий отчет утилиты исследуйте его**.**

Задание 3. Получите полный отчет утилиты**.** Выпишите символическое имя хоста , IP-адрес, маску подсети, MAC-адрес адаптера.

Задание 4. Определите, к какому классу адресов относится выписанный IP-адрес; вычислите максимальное количество хостов, которое может быть в подсети и укажите диапазон их адресов; определите код производителя сетевого адаптера.

7.2.4. Утилита hostname

Задание 5. Определите имя NetBIOS-имя компьютера с помощью утилиты **hostname**. Сравните его с именем полученным с помощью утилиты **ipconfig**.

7.2.5. Утилита ping

Задание 6. Получите справку о параметрах утилиты **ping.**

Задание 7. С помощью **ping** проверьте работоспособность интерфейса внутренней петли компьютера.

Задание 8. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров его IP-адрес.

Задание 9. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров символическое имя хоста.

Задание 10. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров символическое имя хоста и увеличив размер буфера отправки до 1000 байт

Задание 11. С помощью утилиты **ping** проверьте доступность интерфейса какого-нибудь компьютера в локальной сети, указав в качестве параметров его IP-адрес и установив количество отправляемых запросов равное 17.

Примечание. Обратите внимание на значение TTL, которое выдается в отчетах утилиты **ping**. Первоначальное значение TTL (Time To Live, время жизни) по умолчанию равно 128. Это значение записывается в заголовок каждой дейтаграммы и уменьшается на единицу после прохождения каждого маршрутизатора. Если в процессе движения дейтаграммы в сети значение TLL уменьшится до нуля, то дейтаграмма уничтожается. Такой подход гарантирует от зацикливания дейтаграмм в сети. С помощью ключа **i** утилиты ping, можно на период проверки значение TTL изменить.

7.2.5. Утилита tracert

Задание 12. Получите справку о параметрах утилиты **tracert**.

Задание 13. С помощью утилиты **tracert** определите маршрут хоста самого к себе (интерфейс внутренней петли).

Задание 14. С помощью утилиты **tracert** определите маршрут к хосту в локальной сети. Определите количество прыжков в полученном маршруте.

7.2.6. Утилита route

Задание 15. Получите справку о параметрах утилиты **route**.

Задание 16. Распечатайте на экран монитора таблицу активных маршрутов компьютера. Исследуйте полученный отчет. Определите строки таблицы, соответствующие интерфейсу внутренней петли и широковещательным адресам. Определите IP- адреса шлюзов.

7.2.7. Утилита arp

Задание 17. Получите справку о параметрах утилиты **arp**.

Задание 18. Распечатайте на экран монитора arp-таблицу. Исследуйте полученный отчет. Определите хосты, которым соответствуют строки arp-таблицы. Определите IP-адрес, которого нет в arp-таблице, но есть в локальной сети. Выполните утилиту **ping** в адрес этого хоста. Распечатайте снова arp-таблицу и объясните произошедшие изменения. Определите MAC-адреса двух хостов с ближайшими IP-адресами.

7.2.8. Утилита nslookup

Задание 19. **З**апустите утилиту **nslookup** в диалоговом режиме и наберите команду **help**. Ознакомьтесь с полученным отчетом, отражающим возможности утилиты **nslookup**.

Задание 20. **З**апустите утилиту **nslookup** в диалоговом режиме. Определите имя и IP-адрес хоста, на котором установлен DNS-сервер по умолчанию. Определите IP-адреса хостов по их именам (имена хостов выдаст преподаватель).

7.2.9. Утилита netstat

Задание 21. Получите справку о параметрах утилиты **netstat**.

Задание 22. Запустите утилиту **netstat -a** для отображения всех подключений и ожидающих портов. Исследуйте отчет. Выясните, какие из известных служб прослушивают порты. С какими из этих портов поддерживается внешнее соединение и по какому протоколу ? Определите имена хостов и номера портов внешних соединений .

Задание 23. Запустите утилиту **netstat -b** для отображения исполняемых файлов участвующих в создании подключений. Определите исполняемые файлы служб, прослушивающих порты, идентификаторы процессов операционной системы.

Задание 24. Запустите утилиту **netstat -ab**. Исследуйте полученный отчет. Для формирования файла отчета утилиты, перенаправьте вывод утилиты в файл с помощью команды: **netstat -ab > c:\report.txt.** Проконтролируйте наличие отчета в файле.

7.2.10. Утилита nbstat

Задание 25. Получите справку о параметрах утилиты **nbtstat**. Выполните все команды отраженные в справке. Исследуйте полученные отчеты.

7.2.11. Утилита net

Задание 26. Получите справку о параметрах утилиты **net**. Получите справку по отдельным командам утилиты с помощью команды **help.** Получите статистику рабочей станции и сервера компьютера с помощью команды **statistics**. Перешлите сообщение на соседний компьютер с помощью команды **send**. Получите список пользователей компьютера с помощью команды **user**.

7.3. Практическая работа № 2. Обмен данными по TCP-соединению

7.3.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков разработки простейшего распределенного приложения архитектуры клиент-сервер, осуществляющего обмен данными в локальной сети через Windows Sockets TCP-соединение.

Результатом практической работы является разработанное распределенное приложение со схемой взаимодействия процессов, описанной в разделе 3.4 и изображенной на рисунке 3.4.2 пособия.

7.3.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в разделах 3.2-3.11, 3.14 пособия.

7.3.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой сервера, изображенной на рисунке 3.4.2 пособия. Создайте с помощью Visual Studio консольное приложение **ServerT** (наименование проекта), которое будет использовано для построения серверной части приложения (сервера). Включите необходимые директивы компилятора (указанные в разделе 3.2 пособия) для подключения динамической библиотеки **WS2_32.LIB**. Откомпилируйте приложение, убедитесь в отсутствии ошибок.

Задание 2. В рамках приложения **ServerT**, созданного в задании 1, разработайте о функцию **SetErrorMsgText,** предназначенную для обработки стандартных ошибок библиотеки **WS2_32.LIB**. Предполагается, что функция **SetErrorMsgText** будет использоваться в операторе **throw** для генерации исключения при возникновении ошибок в функциях интерфейса Winsock2. Для получения кода ошибки функций Winsock2 примените функцию **WSAGetLastError,** описание которой приводится в разделе 3.3 пособия. Там же приводится полный список кодов возврата функции **WSAGetLastError** и пример ее использования.

Задание 3. Доработайте приложение **ServerT** таким образом, чтобы оно только инициализировало библиотеку **WS2_32.LIB** и завешало работу с этой библиотекой. Для этого используйте функции **WSAStartup** и **WSACleanup**, описанные в разделах 3.5 и 3.6 пособия. Обработку ошибок осуществите с помощью конструкции **try-catch** и функции **SetErrorMsgText**, разработанной в задании 2. Используйте пример программы, приведенный в разделе 3.6 пособия. Убедитесь в работоспособности приложения.

Задание 4. Доработайте приложение **ServerT** таким образом**,** чтобы оно создавало и закрывало сокет, предназначенный для ориентированного на поток соединения. Для этого используйте функции **socket** и **closesocket**, описанные в разделе 3.8. Обратите внимание на параметр **type** функции **socket**, указывающий тип соединения. Воспользуйтесь примером из раздела 3.7 пособия. Убедитесь в работоспособности приложения.

Задание 5. Добавьте в приложение **ServerT** вызов функций **bind** и **listen** для установки параметров сокета и перевода его в режим прослушивания. Функция **bind** описана в разделе 3.8, а функция **listen** в разделе 3.9 пособия. Используйте порт **2000**, в качестве параметра сокета. Установка параметров сокета осуществляется с помощью структуры **SOCKADDR_IN**. Описание этой структуры приводится в разделе 3.8 пособия. Сверьте схему полученной программы со схемой сервера, изображенной на рисунке 3.4.2. Выполните приложение, убедитесь в его работоспособности.

Задание 6. Добавьте в приложение **ServerT** вызов функции **accept**, описание которой приводится в разделе 3.10 пособия. Следует обратить

внимание на: 1) успешным результатом работы функции **accept** является новый сокет; 2) первым параметром функции **accept** является уже созданный ранее сокет; 3) второй параметр функции **accept** – указатель на структуру **SOCKADDR_IN** (не надо ее путать с уже применяемой выше), предназначенную для приема параметров, подключившегося сокета со стороны клиента сокета. Запустите приложение в режиме отладки (Debug) и убедитесь, что после выполнения функции **accept**, программа переходит в режим ожидания (зависает). Завершите приложение. Сохраните программу **ServerT** для дальнейшего применения.

7.3.4. Разработка клиентской части распределенного приложения

Задание 7. Ознакомьтесь со схемой клиента, изображенной на рисунке 3.4.2 пособия. Создайте с помощью Visual Studio новое консольное приложение **ClientT** (наименование проекта), которое будет использовано для построения клиентской части приложения (клиента). Повторите все те же действия для этого приложения, которые были сделаны в заданиях 2-4. Убедитесь в работоспособности приложения **ClientT**.

Задание 8. Добавьте в приложение **ClientT,** вызов функции **connect**. Описание функции и примера ее использования приводится в разделе 3.10 пособия. Следует обратить внимание на следующее: 1) параметры сокета сервера устанавливаются в структуре **SOCKADDR_IN**; 2) для номера порта необходимо установить значение **2000** (такой же номер, что установлен при параметризации сокета сервера а задании 5); 3) для установки номера порта используются специальные функции, описание которых приводится в разделе 3.8. Используйте в качестве IP-адреса собственный адрес компьютера **127.0.0.1** (интерфейс внутренней петли) – это даст возможность отладки приложения на одном компьютере. Запустите приложение на выполнение. Убедитесь, что функция **connect** завершилась с ошибкой и обработка ошибок осуществляется корректно. Найдите полученный код ошибки в таблице 3.3.1 пособия и проанализируйте его.

7.3.5. Обмен данными между сервером и клиентом

Задание 9. Запустите на выполнение приложение **ServerT** и убедитесь, что оно приостановилось на вызове функции **accept**. Запустите на выполнение на этом же компьютере (используется интерфейс внутренней петли) приложение **ClientT**. Убедитесь, что сервер **ServerT**, вышел из состояния ожидания, а клиент **ClientT** завершился без ошибок.

Задание 10. Доработайте программу сервера **ServerT** таким образом, чтобы после подсоединения клиента на экран консоли **ServerT** выводился IP-адрес и порт, подсоединившегося клиента. Необходимые значения находятся в структуре **SOCKADDR_IN**, которая заполняется функцией

accept. Используйте функции **htons** и **inet_ntoa**, описанные в разделе 3.8. Убедитесь в работоспособности распределенного приложения **ClientT-ServerT**.

Задание 11. Добавьте в программу сервера **ServerT** вызов функции **recv,** а в программу клиента вызов функции **send.** Описание этих функций приводится в разделе 3.11 пособия. Перешлите текст ***Hello from Client*** от клиента серверу. Выведите полученный сервером текст на экран консоли. Обратите внимание на то, что команда **recv** в программе сервера использует сокет, созданный функцией **accept,** а не созданный ранее с помощью функции **socket**.

Задание 12. Установите программу **ServerT** на другой компьютер локальной сети, а в программу **ClientT** внесите необходимые изменения, позволяющие ей установить связь с сервером. Убедитесь в работоспособности распределенного в локальной сети приложения **ClientT-ServerT**.

Задание 13. Внесите изменения в программы **ClientT** и **ServerT**, позволяющие 1000 раз передать сообщение типа ***Hello from Client xxx*** (***xxx*** – номер сообщения) от клиента серверу. Убедитесь в работоспособности в сети.

Задание 14. Доработайте программы **ClientT** и **ServerT** таким образом, чтобы программа **ServerT** принимала последовательность сообщений вида ***Hello from Client xxx*** от программы **ClientT** и отправляла их без изменения обратно программе **ClientT**. Программа **ClientT,** получив вернувшееся сообщение, должна увеличить в нем счетчик ***xxx*** на единицу и вновь направить в адрес **ServerT**. Количество передаваемых сообщений введите через консоль программы **ClientT**. Условием окончания работы для программы **ServerT** является получение сообщения нулевой длины. Оцените время обмена 1000 сообщениями (с помощью функций **clock**) между **ClientT** и **ServerT** через локальную сеть. Запустите утилиту **netstat** на компьютерах клиента и сервера, проанализируйте отчет и найдите информацию о приложении **ClientT- ServerT**.

Задание 15. Доработайте программу **ServerT** таким образом, чтобы отключения клиента она могла снова установить соединение с другим клиентом и продолжила свою работу.

7.4. Практическая работа № 3. Обмен данными без установки соединения

7.4.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков разработки простейшего распределенного приложения архитектуры клиент-сервер, осуществляющего обмен данными в локальной сети через Windows Sockets без установки соединения (с помощью UDP-сообщений).

Результатом практической работы является разработанное распределенное приложение со схемой взаимодействия процессов, описанной в разделе 3.4 и изображенной на рисунке 3.4.1 пособия.

7.4.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в разделах 3.2-3.12, 3.14 пособия.

7.4.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой взаимодействия процессов без установки соединения в распределенном приложении, приведенной разделе 3.4 пособия (рисунок 3.4.1). Определите основные отличия этой схемы от схемы взаимодействия процессов с установкой соединения. Разработайте программу **ServerU**, реализующую блоки 1, 2 и 5 схемы сервера, изображенной на рисунке 3.4.1. Подключите функции обработки ошибок, разработанные в практической работе № 2 (с применением команд структурной обработки ошибок **try-throw-catch**). Обратите внимание 1) на параметр **type** функции **socket**; 2) на отсутствие функций **listen** и **accept**, которые применялись в приложении с соединением. Убедитесь, что разработанная программа выполняет все функции Winsock2 без ошибок.

Примечание. При разработке программ в заданиях этой практической работы рекомендуется использовать тексты программ, разработанных в предыдущей практической работе.

Задание 2. Реализуйте в программе **ServerU** блок 3 схемы сервера изображенной на рисунке 3.4.1. Используемая в блоке функция **recvfrom** описана в разделе 3.12 пособия. Установите номер серверного сокета равным **2000**. Убедитесь, что при запуске программа **ServerU** приостанавливает свое выполнение (переходит в состояние ожидания) сразу после вызова функции **recvfrom**. Завершите программу.

7.4.4. Разработка клиентской части распределенного приложения

Задание 3. Создайте новое C++ -приложение с именем **ClientU**. Реализуйте блоки 1, 2, 3 и 5 схемы клиента, изображенной на рисунке 3.4.1. Подключите функции обработки ошибок, разработанные в практической

работе № 2. В параметре **to** команды **sendto** (раздел 3.12), установите адрес структуры **SOCKADDR_IN**, содержащей IP-адрес равный **127.0.0.1** и номер порта равный **2000**. Обеспечьте пересылку сообщения ***Hello from ClientU***. Запустите на выполнение программу **ClientU** при отсутствующем сервере. Проанализируйте полученный код возврата.

7.4.5. Обмен данными между сервером и клиентом

Задание 4. Запустите на выполнение программу **ServerU** и убедитесь, что она приостановила свое выполнение. Запустите на этом же компьютере программу **ClientU** и убедитесь, что программы сервера получила сообщение и завершилась нормально.

Задание 5. Реализуйте блоки 4 в обеих программах. Перешлите полученное сервером сообщение обратно в адрес клиента и убедитесь, что сообщение получено.

Задание 6. Внесите необходимые изменения в программу **ClientU** для того, чтобы программы можно было бы расположить на разных компьютерах локальной сети. Убедитесь в работоспособности приложения.

Задание 7. Реализуйте последовательную пересылку данных от клиента к серверу и обратно по тому же принципу как это было сделано в заданиях 13, 14 практической работы № 2. Проведите измерения аналогичные оценки скорости передачи, сравните результаты.

Задание 8. Запустите сервер **ServerU** на одном из компьютеров и одновременно два клиента на двух других компьютерах локальной сети. Оцените количество сообщений, которые успел передать и получить каждый из клиентов.

Задание 9. Запустите сервер **ServerT** (разработанный в практической работе № 2) и программу клиента **ClientU**. Объясните полученный результат.

Задание 10. Запустите сервер **ServerU** и клиент **ClientT**(разработанный в практической работе № 2) . Объясните полученный результат.

7.5. Практическая работа № 4. Применение широковещательных IP-адресов

7.5.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков использования широковещательных адресов распределенными в локальной сети приложениями.

Результатом практической работы являются разработанное распределенное приложение, использующее широковещательные адреса.

7.5.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в разделах 3.2-3.12, 3.15 пособия.

7.5.3. Разработка серверной части распределенного приложения

Задание 1. Разработайте функцию **GetRequestFromClient**, описание которой представлено на рисунке 7.5.1. Функция предназначена для ожидания запроса клиентской программы. Предполагается, что правильный запрос (позывной сервера) состоит из набора символов, который указывается функции в качестве параметра **name**. Ожидание запроса в функции **GetRequestFromClient** осуществляется с помощью функции **recvfrom**. Если поступившее сообщение является позывным сервера, то функция, заполняет возвращаемую структуру **SOCKADDR_IN** (параметры **from** и **flen** функции) и завершается с кодом возврата **true**. Если поступившее сообщение не является позывным сервера, то оно игнорируется, и функция вновь переходит в состояние ожидания. Если функция **recvfrom** завершается аварийно с кодом **WSAETIMEDOUT** (таблица 3.3.1), то функция **GetRequestFromClient** должна завершиться с кодом возврата **false.** Любой другой аварийный код завершения должен приводить к исключительной ситуации (оператор **throw**), соответствующей функциям обработки ошибок разработанных в практическом занятии № 2.

Создайте новое приложение **ServerB**, вызывающее функцию **GetRequestFromClient**. Пусть позывной сервера будет ***Hello***. Запустите приложение **ServerB** и убедитесь, что программа перешла в состояние ожидания. Запустите приложение **ClientU**, разработанное в практической работе № 3. Убедитесь, что **ServerB** не реагирует на ошибочный позывной. Исправьте в приложении **ClientU** посылаемую строку на ***Hello*** и убедитесь, что **ServerB** реагирует на правильный позывной.

```
// -- обработать запрос клиента
// Назначение: функция предназначена для обработки запроса
//             клиентской программы

bool  GetRequestFromClient(
          char*             name, //[in] позывной сервера
          short             port, //[in] номер просушиваемого порта
          struct sockaddr* from, //[out] указатель на SOCKADDR_IN
          int*              flen  //[out] указатель на размер from
                            )

// Код возврата: в случае если пришел запрос клиента, то
//        функция возвращает значение true, иначе возвращается
//        значение false
// Примечание: параметр name – строка, заканчивающаяся 0x00 и
//        содержащая позывной сервера (набор символов,
//        получаемый сервером от клиента и интерпретируемый,
//        как запрос на установку соединения);
//        параметр from – содержит указатель структуры,
//        содержащей параметры сокета клиента приславшего
//        запрос
```

Рисунок 7.5.1. Описание функции GetRequestFromClient

Примечание. При разработке функции **GetRequestFromClient** и других функций в этом практическом задании целесообразно вызов функций **WSAStartup** и **WSACleanup** осуществлять вне разрабатываемых функций.

Задание 2. Разработайте функцию **PutAnswerToClient**, описание которой приводится на рисунке 7.5.2. Функция предназначена для подтверждения сервером запроса клиента на установку соединения. Функция оправляет в адрес клиента (параметры сокета клиента указываются в параметре **to**) свой позывной, что предполагает готовность сервера к дальнейшей работе с клиентом. Предполагается, что функция будет использоваться после завершения функции **GetRequestFromClient**. Внесите изменения в программу **ServerB**, чтобы сервер смог отвечать с помощью функции **PutAnswerToClient** на правильный позывной полученный от клиента. Проверьте правильность работы сервера **ServerB** с помощью программы **ClientU**.

Задание 3. Внесите изменения в программу **ServerB** таким образом, чтобы сервер отвечал на многократные запросы от разных клиентов (необходимо построить цикл с функциями **GetRequestFromClient** и **PutAnswerToClient**). Проверьте работоспособность сервера.

```
// -- ответить на запрос клиента
// Назначение: функция предназначена пересылки позывного
//             сервера программе клиента

bool  PutAnswerToClient(
          char*             name, //[in] позывной сервера
          struct sockaddr* to,    //[in] указатель на SOCKADDR_IN
          int*              lto   //[in] указатель на размер from
                        )

// Код возврата: в случае успешного завершения функция
//        возвращает значение true, иначе возвращается
//        значение false
// Примечание: параметр name – строка, заканчивающаяся 0x00 и
//        содержащая позывной сервера (набор символов,
//        получаемый сервером от клиента и интерпретируемый,
//        как запрос на установку соединения);
//        параметр to – содержит указатель структуры,
//        содержащей параметры сокета клиента
```

Рисунок 7.5.2. Описание функции PutAnswerToClient

7.5.4. Разработка клиентской части распределенного приложения

Задание 4. Разработайте функцию **GetServer**, описание которой приводится на рисунке 7.5.3. Функция предназначена для отправки широковещательного запроса в локальную сеть (всем компьютерам сегмента локальной сети) с позывным сервера. Предполагается, что на одном (или на нескольких) компьютере сети есть сервер **ServerB**, который просушивает порт с номером, указанным в параметре **port** функции **GetServer** . Для отправки широковещательного запроса, функция **GetServer**, должна использовать широковещательный IP-адрес (раздел 3.15 пособия). Использование широковещательного IP-адреса требует специального режима работы сокета, который устанавливается с помощью функции **setsockopt**, входящей в состав Winsock2. Описание этой функции и пример ее использования приводятся в разделе 3.15 пособия. После отправки широковещательного запроса с помощью функции **sendto**, функция **GetServer** должна вызвать функцию **recvfrom** для ожидания отклика сервера. При правильном отклике (отклик должен совпадать с позывным), функция формирует структуру **SOCKADDR_IN** с параметрами сокета сервера, возвращает значение **true** и завешается. Если сообщение в адрес клиента приходит, но отклик не содержит правильный позывной или функция **recvfrom** аварийно завершается с кодом **WSAETIMEDOUT**, функция должна завешаться с кодом возврата **false**. Любой другой аварийный код завершения должен приводить к исключительной ситуации

(оператор **throw**), соответствующей функциям обработки ошибок разработанных в практическом занятии № 2.

```
// -- послать запрос серверу и получить ответ
// Назначение: функция предназначена широковещательной
//             пересылки позывного серверу и приема от
//             сервера ответа.

bool  GetServer(
          char*             call, //[in] позывной сервера
          short             port, //[in] номер порта сервера
          struct sockaddr* from, //[out] указатель на SOCKADDR_IN
          int*              flen  //[out] указатель на размер from
               )
// Код возврата: в случае успешного завершения функция
//        возвращает значение true, иначе возвращается
//        значение false
// Примечание: - параметр call – строка, заканчивающаяся 0x00
//        и содержащая позывной сервера;
//        - параметр from – содержит указатель структуры,
//        которая содержит параметры сокета откликнувшегося
//        сервера
```

Рисунок 7.5.3. Описание функции GetServer

Создайте новое приложение **ClientB**, вызывающее функцию **GetServer**. Запустите сервер **ServerU,** разработанной в практической работе № 3. Убедитесь с помощью отладчика, что происходит обмен данными и функция **GetServer** завершается с возвратом **false** после получения неверного отклика.

7.5.5. Взаимодействие сервера и клиента

Задание 5. Запустите на разных компьютерах программы **ClientB** и **ServerB**. Внесите изменения в программу **ClientB** для того, чтобы она выводила на экран консоли параметры сокета сервера откликнувшегося на позывной. Внесите изменения в программу **ServerB** для того, чтобы она выводила на экран консоли параметры клиента, отправившего правильный позывной в адрес сервера.

Примечание. Разработанные функции **GetRequestFromClient** и **GetServer** имеют существенный недостаток. После вызова функции **recvfrom** они переводят поток в режим ожидания. Выход из этого состояния возможен лишь в том случае, если в адрес сокета поступило сообщение или будет исчерпан допустимый интервал ожидания. Такой алгоритм работы делает эти функции мало применимыми. Сохраните тексты этих функций, они будут дорабатываться в следующих практических работах.

Задание 6. Измените программу **ServerB** таким образом, чтобы при запуске она проверяла наличие в локальной сети еще одного такого же сервера (точнее сервера с тем же позывным) и выдавала на экран консоли предупредительное сообщение о количестве существующих серверов и их IP-адресах.

7.6. Практическая работа № 5. Использование символических имен компьютеров

7.6.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков применения символических имен компьютеров при разработке распределенного в локальной сети приложения.

Результатом практической работы являются разработанное распределенное приложение, использующее символические имена компьютеров.

7.6.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в разделах 2.8.1, 2.8.3, 3.16 пособия.

7.6.3. Определение адреса компьютера по его символическому имени

Задание 1. Разработайте функцию **GetServerByName**, описание которой приводится на рисунке 7.6.1. Функция предназначена для поиска сервера по его символическому имени и позывному. При этом предполагается, что в локальной сети работает одна из систем (DNS, NetBIOS over TCP/IP), разрешающих символические имена компьютеров. Функция **GetServerByName** является, в некотором смысле, альтернативой функции **GetServer (**практическая работа № 4**)** и должна использоваться в том случае, если известно символическое имя компьютера, на котором запущен сервер. Для поиска сервера фунция **GetServerByName** должна применить функцию **gethostbyname**, описание которой приводится в разделе 3.16. Там же имеется описание структуры **hostent**, которая используется этой функцией для хранения результата работы функции **gethostbyname**. После того, как IP-адрес сервера определен, необходимо установить необходимый номер порта и послать позывной в адрес сокета сервера. В остальном функция **GetServerByName** должна работать по тому же принципу, что и функция **GetServer**. Создайте новое приложение **ClentS**, вызывающее функцию **GetServerByName**. Проверьте работоспособность приложения при работе с программой **ServerB**.

Примечание. Функция **GetServerByName** имеет те же недостатки, что и функция **GetServer.** Сохраните текст этой функции, она будет дорабатываться в следующих практических работах

```
// -- послать запрос серверу, заданному символическим именем
// Назначение: функция предназначена пересылки позывного
//             серверу, адрес которого задан в виде
//             символического имени компьютера.

bool  GetServerByName(
       char*             name, //[in] имя компьютера в сети
       char*             call, //[in] позывной
       struct sockaddr* from, //[in,out] указатель на SOCKADDR_IN
       int*              flen  //[in,out] указатель на размер from
                    )
// Код возврата: в случае успешного завершения (сервер
//        откликнулся на позывной) возвращает значение true,
//        иначе возвращается  значение false
// Примечание: - параметр name – строка, заканчивающаяся 0x00
//        и содержащая  символическое имя компьютера;
//        - параметр call – строка, заканчивающаяся 0x00 и
//        содержащая позывной сервера;
//        - параметр from – содержит указатель структуры
//        SOCKADDR_IN, которая содержит параметры сокета
//        откликнувшегося сервера, перед вызовом функции поле
//        sin_port должно  быть заполнено; если после вызова
//        функции, код возврата равен true, то структура
//        SOCKADDR_IN содержит все параметры сокета сервера
```

Рисунок 7.6.1. Описание функции GetServerByName

7.6.4. Определение имени компьютера по его сетевому адресу

Задание 2. Доработайте программу **ServerB** таким образом, чтобы она распечатывала на экран консоли символическое имя собственного компьютера и символические имена компьютеров клиентов, которые подключаются к серверу. Программа **ServerB** должна использовать функции **gethostname** и **gethostbyaddr**, описание которых приводится в разделе 3.16 пособия.

7.7. Практическая работа № 6. Работа с интерфейсом Named Pipe

7.7.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков использования интерфейса Named Pipe для разработки распределенного приложения.

Результатом практической работы являются разработанное распределенное приложение, применяющее интерфейс Named Pipe .

7.7.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в четвертой главе пособия.

7.7.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой сервера, изображенной на рисунке 4.2.1. Создайте с помощью Visual Studio новое консольное приложение **ServerNP** (наименование проекта), которое будет использовано для построения серверной части распределенного приложения (сервера). Реализуйте блоки 1 и 4 сервера. В блоке 1 используются функции **CreateNamedPipe** и **ConnectNamedPipe**, описание которых приводится в разделе 4.3 пособия. Там же приведены примеры использования этих функций. Следует обратить внимание на формат имени канала, используемый функции **CreateNamedPipe –** он должен быть локальным. Пусть имя создаваемого именного канала будет ***Tube***. В блоке 4 используется функция **DisconnectNamedPipe** для разрыва соединения. Описание функции приводится в разделе 4.3 пособия. Разработайте функции обработки ошибок интерфейса Named Pipe, работающие по тому же принципу, что и функции обработки ошибок Winsock2, используемые в предыдущих практических работах. Запустите приложение **ServerNP** на выполнение и убедитесь, что поток приостановился для ожидания соединения после вызова функции **ConnectNamedPipe**.

7.7.4. Разработка клиентской части распределенного приложения

Задание 2. Ознакомьтесь со схемой клиента, изображенной на рисунке 4.2.1. Создайте с помощью Visual Studio новое консольное приложение **ClientNP**, которое будет использовано для построения клиентской части распределенного приложения (клиента). Реализуйте блоки 1 и 4 клиента. В блоке 1 применяется функция **CreateFile**, описание которой приводится в разделе 4.4 пособия. Установите в параметрах вызова функции **CreateFile**, такое же имя именованного канала, что и для сервера (задание 1). Запустите программу **ClientNP** отдельно (без сервера) и проверьте работоспособность функций обработки ошибок.

7.7.5. Обмен данными между клиентом и сервером

Задание 3. Запустите на выполнение программу **ServerNP**. После того, как программа **ServerNP** перейдет в состояние ожидания, запустите на этом же компьютере программу **ClientNP**. Убедитесь, что программа **ServerNP** вышла из состояния ожидания и успешно завершилась. Программа **ClientNP** тоже должна завершиться без ошибок.

Задание 4. Реализуйте блоки 2 и 3 программ **ServerNP** и **ClientNP**. Добейтесь, чтобы программы обменялись сообщениями с помощью функций **WriteFile** и **ReadFile**, описание которых приводится в разделе 4.5 пособия.

Задание 5. Внесите изменения в программы **ServerNP** и **ClientNP** таким образом, чтобы программы взаимодействовали также, как и программы **ServerT** и **ClientT** в заданиях 14 и 15 практической работы № 2.

Задание 6. Внесите изменения в программу **ClientNP** и добейтесь взаимодействие программ клиента и сервера в случае расположения на разных компьютерах локальной сети. Следует иметь в виду, что при вызове функции **CreateFile**, теперь следует использовать сетевой формат имени именованного канала, описанный в разделе 4.3 пособия. В сетевом формате используется символическое имя серверного компьютера. Напомним, что это имя можно получить с помощью утилиты **hostname**, описанной в разделе 2.9.

Задание 7. Разработайте новую программу **ClientNPt**, использующую функцию **TransactNamedPipe** (раздел 4.6 пособия) вместо пары функций **WriteFile** и **ReadFile**. Добейтесь взаимодействия программы **ClientNPt** с сервером **ServerNP**.

Задание 8. Разработайте еще одну новую программу **ClientNPct**, использующую функцию **CallNamedPipe** (раздел 4.6 пособия). Добейтесь взаимодействия программы **ClientNPct** с сервером **ServerNP**.

7.8. Практическая работа № 7. Работа с интерфейсом Mailslots

7.8.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков использования интерфейса Mailslots для разработки распределенного приложения.

Результатом практической работы являются разработанное распределенное приложение, применяющее интерфейс Mailslots .

7.8.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в пятой главе пособия.

7.8.3. Разработка серверной части распределенного приложения

Задание 1. Ознакомьтесь со схемой сервера, изображенной на рисунке 5.2.1. Создайте с помощью Visual Studio новое консольное приложение **ServerMS** (наименование проекта), которое будет использовано для построения серверной части распределенного приложения (сервера). Реализуйте блоки 1 и 3 сервера. В блоке 1 используется функция **CreateMaislot**, описание которой приводится в разделе 5.3 пособия. Следует обратить внимание на формат имени канала, используемый функции **CreateMaislot** – он должен быть локальным. Пусть имя создаваемого именного канала будет ***Box***. Кроме того, установите в параметрах функции **CreateMaislot** бесконечный интервал ожидания для функции **ReadFile** и максимальную длину сообщения равную **300** байт. Разработайте функции обработки ошибок интерфейса Mailslot, работающие по тому же принципу, что и функции обработки ошибок Winsock2, используемые в предыдущих практических работах. Запустите приложение **ServerMS** на выполнение и убедитесь, что поток не приостановился для ожидания и программа **ServerMS** завершилась без ошибок.

Задание 2. Реализуйте в программе **ServerMS** блок 2 схемы сервера (рисунок 5.2.1) и запустите на выполнение. В блоке 2 применяется универсальная функция **ReadFile**, описание которой приводится в разделе 4.5 пособия. Запустите программу на выполнение. Убедитесь, что главный поток программы **ServerMS** приостановился для ожидания сообщения от клиента. Завершите программу с помощью отладчика.

Задание 3. Измените в параметрах вызова функции **CreateMaislot** программы **ServerMS** интервал времени ожидание на значение соответствующее трем минутам. Запустите программу **ServerMS** на выполнение и убедитесь, что главный поток приостановился на три минуты для ожидания сообщения клиента. Обработайте этот случай: выведите соответствующее сообщение на консоль сервера.

7.8.4. Разработка клиентской части распределенного приложения

Задание 4. Ознакомьтесь со схемой клиента, изображенной на рисунке 5.2.1. Создайте с помощью Visual Studio новое консольное приложение **ClientMS**, которое будет использовано для построения клиентской части распределенного приложения (клиента). Реализуйте схему клиента полностью. Для реализации первого блока необходимо использовать, функцию **CreateFile**, описание которой приводится в разделе 5.4. Установите локальный формат имени почтового сервера в параметрах функции. Для отправки сообщения используется универсальная функция **WriteFile**, описание которой можно найти в разделе 4.5. Разработайте функции обработки ошибок интерфейса Mailslot. Программа **ClientNS** должна пересылать серверу сообщение ***Hello from Maislot-client***.

Запустите программу **ClientMS** без наличия на компьютере сервера. Убедитесь, что обработка ошибок работает корректно.

7.8.5. Обмен данными между клиентом и сервером

Задание 5. Запустите на выполнение программу **ServerMS.** После того, как программа **ServerMS** перейдет в состояние ожидания запустите на этом же компьютере программу **ClientMS**. С помощью отладчика убедитесь, что сообщение от клиента к серверу передается корректно.

Задание 6. Внесите такие изменения в программу **ClientMS** , чтобы она могла пересылать сообщение серверу **ServerMS**, который расположен на другом компьютере. Добейтесь работоспособности распределенного приложения **ClientMS- ServerMS**.

Задание 7. Внесите такие изменения в программу **ClientMS**, чтобы она могла пересылать сообщение нескольким экземплярам сервера **ServerMS**, расположенным на других компьютерах локальной сети. Добейтесь работоспособности одной клиентской программы с тремя экземплярами сервера.

Задание 8. Увеличьте размер максимальный размер пересылаемого сообщения до 500 (параметры функции **CreateMaislot**) и проверьте возможность работы одного клиента с тремя серверами.

Задание 9. Оцените скорость пересылки 1000 сообщений от одного клиента одному серверу, по тому же принципу, как это было сделано в задании 14 практической работы № 2.

7.9. Практическая работа № 8. Разработка параллельного сервера

7.9.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков разработки параллельного сервера, одновременно обслуживающего разнотипные запросы нескольких клиентов.

Результатом практической работы является разработанный параллельный сервер, на основе модели ConcurrentServer, описанной в шестой главе пособия.

7.9.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в третьей, пятой и шестой главах пособия.

7.9.3. Предварительные замечания

Разработка параллельного сервера будет опираться на модель ConcurrentServer, описанную в шестой главе пособия. По мере изложения материала эта модель претерпевала изменения. Последняя версия модели изложена в разделе 6.13 – именно на этот вариант будут ориентированны все дальнейшие построения.

Для того, чтобы избежать двойного толкования, будем считать далее, что модель ConcurrentServer есть ни что иное, как спецификация, описывающая разрабатываемый здесь параллельный сервер с именем ConcurrentServer.

Кроме того, в изложении часто будут использоваться названия функций, которые являются функциями потоков (потоковые функции, раздел 6.4 пособия). Для краткости, если это не приводит к двусмысленности, потоки будут именоваться по имени потоковой функции. Например, функция **AcceptServer** является потоковой функцией, тогда соответствующий ей поток будем именовать – поток **AcceptServer**. Следует отметить, что может быть создано, несколько различных потоков, имеющих общую потоковую функцию. В таких случаях будем говорить о нескольких экземплярах потока.

7.9.4. Исследование структуры параллельного сервера

Задание 1. Исследуйте структуру сервера **ConcurrentServer**, изложенную в разделах 6.1 – 6.3 и 6.11 – 6.13 и изображенную на рисунке 6.13.1. В таблице 7.9.1 приведены все основные программные компоненты сервера. Составьте, краткую спецификацию каждой программной компоненты, отражающую назначение, краткое описание алгоритма, используемые структуры данных и способ взаимодействия с другими компонентами.

Задание 2. В таблице 7.9.2 приведены все основные структуры данных, используемые сервером. Составьте, краткое описание каждой структуры, отразите состав информации, назначение и способ их использования программными компонентами.

Задание 3. В таблице 7.9.3 приведено краткое описание команд управления сервером, которые вводятся с удаленной консоли **RConsole** (клиентская часть консоли управления сервером) и исполняются сервером. Добавьте к описанию каждой команды, принципы ее реализации, название программной компоненты реализующей команду. Опишите структуры данных, которые предполагается применить для хранения списка команд (далее будем называть его **TalkersCmd**), а также для сбора необходимой статистики (будем называть эту структуру **StatsInfo**).

Таблица 7.9.1

Наименование компоненты	Назначение компоненты
main	Главный поток сервера
AcceptServer	Подключение клиентов к серверу
ConsolePipe	Серверная часть консоли управления
GarbageCleaner	Сборщик мусора
DispatchServer	Диспетчеризация запросов клиента
DllMain	Главная функция динамически подключаемой библиотеки
SSS	Запуск обслуживающего потока
EchoServer	Отладочный поток обслуживания

Таблица 7.9.2

Наименование структуры	Состав информации
ListContact	Список подключения клиентов (раздел 6.5)
TableService	Таблица обслуживающих потоков (раздел 6.13)

Таблица 7.9.3

Команда	Краткое описание команды
start	Команда разрешает подключение клиентов к серверу.
stop	Команда запрещает подключение клиентов к серверу.
exit	Команда завершает работу сервера.
statistics	Вывод статистики: общее количество подключений; количество активных подключений, количество отказов в обслуживании.
wait	Команда приостанавливает подключение клиентов до тех пор, пока не обслужится последний клиент, подключенный к серверу.
shutdown	Команда равносильна последовательности команд: wait, exit.
getcomand	Служебная команда, которая не предназначена для ввода с консоли управления, а устанавливается сервером для указания, что сервер готов принять и обработать, очередную команду управления.

Задание 4. В таблице 7.9.4 приведены коды команды (или запросов), которые будет обрабатывать поток **DispatchServer**, после того, как клиент подключится к клиенту. Эти коды хранятся в таблице **TableService**, и каждому коду соответствует свой обслуживающий поток (раздел 6.12-6.13 пособия). Краткое описание обслуживающих потоков приводится в таблице 7.9.4. На первом этапе предполагается, что сервер будет

обслуживать три различных запроса: ***echo***, ***time*** и ***rand***. Опишите, принципы реализации обслуживающих потоков. Для получения системной даты и времени, а также случайных чисел в обслуживающих потоках, следует воспользоваться функциями стандартной библиотеки C++ [13].

Таблица 7.9.4

Код команды (запрос)	**Краткое описание команды**
echo	Вызов обслуживающего потока EchoServer. EchoServer – потоковая функция обеспечивающая прием данных от подключившегося клиента, и пересылку этих же данных обратно без изменения. Условие завершения работы: прием данных нулевой длинны.
time	Вызов обслуживающего потока TimeServer. TimeServer – потоковая функция, принимающая только символьную последовательность ***time*** и отвечающая системным значением даты и времени серверного компьютера в формате: ***dd.mm.yyyy/hh:mm:ss***. Условие завершение работы: прием любой последовательности символов, отличной от ***time***.
rand	Вызов обслуживающего потока RandServer. RandServer – потоковая функция, принимающая только строку ***rand*** и отвечающая случайным целым четырехбайтовым числом в intel-формате. Условие завершение работы: прием любой последовательности символов, отличной от ***rand***.

7.9.5. Создание лавного потока main

Задание 5. Создайте с помощью Visual Studio новое консольное приложение **ConcurrentServer** (наименование проекта). Включите необходимые директивы компилятора для подключения библиотеки **WS2_32.LIB (**раздел 3.2 пособия**).** Подключите функции обработки ошибок интерфейса Winsock2, разработанные в практической работе № 2. Откомпилируйте приложение и убедитесь в отсутствие ошибок.

Задание 6. Создайте в рамках проекта **ConcurrentServer** потоковую функцию **AcceptServer**. Пусть на первом этапе эта функция циклически выдает сообщение ***AcceptServer*** на экран консоли каждые 5 секунд (используйте функцию **Sleep**, описанную в разделе 6.4). Обратите внимание на правильное оформление потоковой функции. Опираясь на описание функции **CreateThread** и пример в разделе 6.4, обеспечьте создание потока **AcceptServer**. Следует помнить о необходимости использовать функции

WaitForSingleObject и **CloseHandle**. Проверьте работоспособность приложения.

Примечание. Единственный параметр функции **AcceptServer** будет в дальнейшем использоваться для передачи адреса области памяти, в котором будет храниться команда управления, записанная туда потоком **ConsolePipe**. Поток **ConsolePipe (**серверная часть консоли управления**)**, получает команды управления (таблица 7.9.3) от удаленной консоли **RConsole** через именованный канал (глава 4).

Задание 7. Обеспечьте передачу номера порта сервера через параметры функции **main**. Причем, если параметр не устанавливается в вызове командной строки сервера **ConcurrentServer**, то по умолчанию должен применяться порта **2000**. Выведите на консоль сервера, номер используемого порта. Кроме того, следует передать через единственный параметр потока **AcceptServer** начальную команду управления сервером: это должна быть команда start (таблица 7.9.3), запускающая процесс подключения клиентов. Убедитесь в корректности передачи параметров, запустив приложение **ConcurrentServer** через командную строку.

7.9.6. Реализация потока AcceptServer

Задание 8. Удалите отладочный цикл вывода в функции **AcceptServer**, созданный в предшествующем задании. Реализуйте в рамках этой функции блоки 1,2,3 и 6 схемы сервера изображенной на рисунке 3.4.2. При этом следуйте указаниям заданий 2-5 практической работы № 2. Убедитесь в работоспособности всего приложения **ConcurrentServer**.

Примечание. Номер используемого сервером порта (полученный через параметры функции **main** или установленный по умолчанию), следует каким-нибудь образом передать в функцию **AcceptServer**. Очевидно, самый простой способ – это организация общедоступной области памяти для потоков **main** и **AcceptServer**.

Задание 9. Реализуйте цикл обработки команд в рамках функции (и потока) **AcceptServer**. Возьмите за основу пример функции **CommandsCycle**, приведенный в разделе 6.9 пособия. На этом этапе разработки замените вызов функции **AcceptCycle** на вызов функции **Sleep** (раздел 6.4). Убедитесь, что команда управления **start** корректно передана в функцию **CommandsCycle**. Следует обратить внимание на применение функции **ioctlsocket (**описание в разделе 6.9 пособия**),** которая переключает сокет в режим без блокировки.

Задание 10. Реализуйте цикл подключения в рамках функции (и потока) **AcceptServer (**в предыдущем примере вместо вызова этой функции

вызывалась функция **Sleep)**. Используйте пример функции **AcceptCycle**, приведенный в разделе 6.9 пособия. Обратите внимание, что вся информация о подключении клиента записывается в список **ListContact**, для последующего его использования в потоке **DispathServer**. Кроме того, следует обратить внимание на обработку ошибок функции **accept**, применяемую для сокета в режиме без блокировки. Используйте программу **ClientT** (разработанную в практической заботе № 2) в качестве клиента для проверки процесса подключения клиентов функцией **AcceptServer**.

7.9.7. Реализация потока DispatchServer

Задание 11. Поток **DispathServer** должен создаваться и запускаться в функции **main –** сразу после запуска потока **AcceptServer**. В качестве параметра передаваемого в потоковую функцию **DispathServer** будем использовать адрес области памяти, в которой хранится команда управления, записанная туда потоком **ConsolePipe** (эта область уже используется потоком **AcceptServer**). На этапе этого задания, по аналогии с заданием 6, зациклите вывод сообщения ***DispathServer***. Не забудьте о необходимости применения функций **WaitForSingleObject** и **CloseHandle**. Убедитесь, что поток создается **DispathServer** и работает.

Примечание. Обратите внимание, что область памяти команды управления используется двумя потоками **AcceptServer** и **DispathServer**. В таблице 7.9.3 приведены команды управления, на которые будет реагировать только поток **AcceptServer (**будем называть их командами управления **AcceptServer)**. В будущем предполагается расширить список команд, добавив туда команды управления **DispathServer**. Примером такой команды может служить команда ***exclude sss*** – позволяющая исключить из таблицы обслуживающих потоков **TableService**, строку с кодом запроса ***sss***. Будем предполагать, для управления каждым потоком, будем использовать свой (кроме общей команды **exit**) набор команд управления. Рекомендуется при объявлении общей области памяти для хранения команды управления использовать квалификатор **volatile**, как это сделано в примерах раздела 6.4.

Задание 12. В соответствии со схемой сервера **ConcurrrenServer**, поток **DispathServer** должен последовательно просматривать список **ListContact (**в котором хранятся все сведения о подключившихся клиентах**)** и считывать запрос, подключившегося клиента. Если этот запрос соответствует одной из строк таблицы **TableService** (запросы приведены в таблице 7.9.4), то **DispathServer** должен запустить соответствующий запросу поток, передав ему в качестве параметра соответствующий элемент списка **ListContact**. Удалите отладочный цикл, построенный в предыдущем задании и реализуйте цикл сканирования списка **ListContact**. Условием выхода из цикла, пусть будет команда управления **exit**. При обнаружении в списке подключившегося, но не обслуженного клиента (это можно

определить по специальным признакам, записанным в элемент списка) следует прочитать первое сообщение клиента (которое должно быть кодом запроса) с помощью функции **recv** (раздел 3.11). Для отладки выведите считанный код запроса подключившегося клиента на экран консоли. Убедитесь, что код запроса считывается корректно. Используйте программу **ClientT**, при отладке потока **DispathServer**. Проверьте работу сервера с несколькими клиентами в сети.

Примечание. Заметим, что сокет, обрабатываемый потоком **DispathServer**, находится в режиме без блокировки (так как получен командой **accept**, использующей не блокирующий сокет) и поэтому требует соответствующего алгоритма обработки. Если бы использовался альтернативный режим (с блокировкой), то был бы возможен случай, когда **DispathServer** остановился в ожидании запроса от клиента, который по неизвестной причине после подключения не отправил серверу запрос. При этом все остальные клиенты, отправившие запрос ожидали бы обслуживания. После того как запрос получен и обработан, следует переключить сокет в блокирующий режим работы, т.к. теперь с сокетом будет работать обрабатывающий поток.

7.9.8. Реализация библиотеки обслуживающих серверов

Задание 13. Создайте с помощью Visual Studio новое DLL-приложение с именем **ServiceLibrary**. Реализуйте в рамках динамически подключаемой библиотеки потоковую функцию **EchoServer**. В качестве параметра этой функции должен передаваться элемент списка **ListContact**, в котором хранятся все параметры подключения (в том числе сокет). Алгоритм работы функции совпадает с алгоритмом работы **ServerT**, разработанной в практической работе № 2, с учетом того, что подключение клиента уже выполнено. Кроме того, перед завершением работы функция **EchoServer** должна сделать соответствующую отметку в переданном ей элементе списка **ListContact**. Реализуйте таблицу **TableService** и экспортируемую функцию **SSS**, используя пример в разделе 6.13 пособия. Заполните таблицу кодами запросов, приведенными в таблице 7.9.4. Для всех кодов, при заполнении таблицы **TableService** для отладки используйте один и тот же адрес обслуживающего потока **EchoServer**.

Задание 14. Обеспечьте передачу имени файла библиотеки через второй параметр функции **main**. Если параметр не задан, то по умолчанию будем предполагать, что файл библиотеки имеет имя **ServiceLibrary**.Загрузите динамически подключаемую библиотеку в рамках функции **main** с помощью функции **LoadLibrary**, описание которой приводится в разделе 6.13 пособия. После загрузки библиотеки, импортируйте функцию **SSS,** применив функцию **GetProcAddress** (раздел 6.13). Обеспечьте возможность вызова

импортированной функции, потоком **DispatchServer**. Перед завершением функции **main** (после завершения всех потоков), отключите библиотеку с помощью функции **FreeLibrary** (раздел 6.13). Используйте примеры программ, приведенные в разделе 6.13. С помощью отладчика убедитесь в корректной передаче параметра с именем функции, успешной загрузке библиотеки и импорте функции.

Примечание. Параметризация имени файла библиотеки дает возможность загружать разные версии библиотеки **ServiceLibrary** при инициализации сервера. Целесообразно было бы разработать несколько функций, которые бы инкапсулировали функции **LoadLibrary**, **FreeLibrary**, **GetProcAddress**, и уже эти функции использовать в **main.** Кроме того, имеет смысл поддерживать номер версии библиотеки (как это сделано, например, для библиотеки WS2_32.DLL, описанной в третьей главе).

7.9.9. Запуск обслуживающего потока из функции DispatchServer

Задание 15. Удалите отладочный вывод из функции **DispatchServer**, реализованный в предшествующих заданиях**.** Используя импортированную из динамической библиотеки функцию **SSS,** обеспечьте запуск обрабатывающего потока соответствующего запросу клиента. Кроме того, обработайте неправильные запросы (не соответствующие таблице **TableService** и таблице 7.9.4). В случае неправильного запроса функция **DispatchServer** должна передать клиенту сообщение ***ErrorInquiry***, разорвать соединение (выполнить функцию **closesocket,** описанную в разделе 3.7 пособия) и сделать соответствующую отметку в элементе списка **ListContact** (адрес элемента найден в результате сканирования списка **ListContact**, раздел 7.9.7). Кроме того возможен случай (об этом упоминалось выше в примечании), когда клиент задерживает отправку запроса после подключения – этот случай тоже требует отдельной обработки и соответствующей диагностики. Для отладки используйте программу **ClientT**, разработанную практической работе № 2. Внесите необходимые изменения в программу **ClientT** для того, чтобы она в первом сообщении серверу передавала запрос (таблица 7.9.4). Убедитесь, что **ConcurrentServer** запускает обслуживающий поток для корректных запросов, а также отсылает сообщение и разрывает соединение в ответ на ошибочные запросы.

7.9.10. Реализация потока GarbageCleaner

Задание 16. Поток **GarbageCleaner** должен создаваться и запускаться в функции **main –** сразу после запуска потока **DispatchServer**. В качестве параметра передаваемого в потоковую функцию **GarbageCleaner** будем использовать адрес области памяти, в которой хранится команда управления, записанная туда потоком **ConsolePipe** (эта область уже используется другими потоками). На этапе этого задания, по аналогии с заданием 6,

зациклите вывод сообщения ***GarbageCleaner***. Кроме того, обеспечьте доступ потока ***GarbageCleaner*** к списку **ListContact**. Следует помнить о необходимости применения функций **WaitForSingleObject** и **CloseHandle**. Убедитесь, что поток создается **GarbageCleaner** и работает.

Задание 17. Удалите отладочный вывод в функции **GarbageCleaner** и создайте цикл сканирования списка **ListContact**. Функция **GarbageCleaner** должна выявлять неиспользуемые (об этом есть соответствующая отметка, сделанная потоком **DispatchServer** или обслуживающим сервером) элементы списка **ListContact**. Запустите сервер **ConcurrentServer** и обеспечьте подсоединение к нему одного клиента. Убедитесь с помощью отладчика, что поток **GarbageCleaner** удаляет неиспользуемые элементы списка **ListContact** в случае успешного обслуживания клиента, в случае выдачи клиентом неправильного запроса, а также, если клиент завершился аварийно во время сеанса связи. В последнем случае может потребоваться доработка функции **EchoServer**.

Примечание. Такой алгоритм работы потока **GarbageCleaner** не является оптимальным, т.к. не рационально используется ценный ресурс процессора. Для исправления этого недостатка, можно использовать функцию **Sleep (**раздел 6.4**)** внутри цикла сканирования для организации задержки (освобождения ресурса), а интервал времени задержки можно параметризировать. Другой способ устранения нерационального использования процессорного времени, основанный на назначении приоритетов, будет предложен в следующих заданиях

7.9.11. Синхронизация потоков AcceptServer и GarbageCleaner

Задание 18. Ознакомьтесь со схемой применения механизма критических секций, приведенной в разделе 6.5 пособия. Критическим ресурсом в сервере **ConcurrentServer** является список **ListContact**: одновременное удаление (поток **GarbageCleaner**) и добавление (поток **AcceptServer**) элементов списка может привести к его разрушению. Используйте механизм критических секций (раздел 6.5 пособия) для синхронизации потоков **AcceptServer** и **GarbageCleaner**. Убедитесь в работоспособности сервера, при интенсивных подключениях и отключениях нескольких клиентов одновременно. Разработайте на базе программы **ClentT**, новую программу клиента **ClientTy**, осуществляющую в цикле подключение к серверу и отключение. Используйте эту программу для тестирования сервера.

7.9.12. Синхронизация потока AcceptServer и обслуживающих потоков

Задание 19. Ознакомьтесь со схемой использования механизма асинхронного вызова процедур, приведенной в разделе 6.6 пособия.

Обеспечьте вывод сообщений на консоль сервера в рамках потока **AcceptServer** о старте и завершении работы обслуживающего потока с помощью механизма асинхронных процедур. Выводимое на консоль сообщение должно содержать наименование обслуживающего сервера, а также дата и время начала и завершения его работы (с функциями для работы с системным временем можно ознакомиться в [13,14]). В параметрах функции **QueueUserAPC** (раздел 6.6) используется дескриптор потока исполняющего асинхронную процедуру. В нашем случае – это поток **AcceptServer**. Рекомендуется при вызове асинхронных процедур, в качестве параметра указывать адрес соответствующего элемента списка **ListContact**. Обслуживающий поток должен иметь доступ к этому дескриптору. Следует обратить внимание на второй параметр функции **SleepEx** (раздел 6.6), которая будет использоваться в функции **AcceptServer** для запуска асинхронных процедур. Значение этого параметра должно быть **TRUE**. Значение первого параметра функции **SleepEx** рекомендуется установить в нуль. Убедитесь в работоспособности сервера при обслуживании нескольких клиентов одновременно.

Примечание. Полезной была бы разработка функции инкапсулирующую функцию **QueueUserAPC**. Процесс разработки обслуживающих серверов должен сопровождаться разработкой технологией и соответствующего API, упрощающие разработку новых обслуживающих серверов.

7.9.13. Предупреждение зацикливания обслуживающих потоков

Задание 20. Ознакомьтесь со схемой использования ожидающего таймера, приведенной в разделе 6.7. Обеспечьте создание в функции **DispatchServer** для каждого запускаемого обслуживающего сервера ожидающий таймер, отслеживающего максимальное время работы обслуживающего сервера. Установите значение ожидающего таймера так, чтобы он переходил в сигнальное состояние через одну минуту после запуска обслуживающего сервера (т.е. одна минута – это максимально допустимое время работы обслуживающего сервера). Дескриптор ожидающего таймера рекомендуется создать в соответствующем элементе списка **ListContact**. Если обслуживающий сервер, завершает свою работу до истечения установленного максимального интервала его работы, то ожидающий таймер должен быть отменен с помощью функции **CancelWaitableTimer** (раздел 6.7). Отмену таймера рекомендуется выполнить в асинхронной процедуре, разработанной для вывода сообщения о завершении работы обслуживающего потока (раздел 7.9.12). Если же таймер срабатывает (обслуживающий поток работает более установленного максимального интервала времени), следует предпринять действия по завершению потока. Используйте процедуру завершения ожидающего таймера, которая помещается в очередь асинхронных процедур потока,

создавшего ожидающий таймер (раздел 6.7). Рекомендуется в качестве параметра этой процедуры использовать адрес соответствующего элемента списка **ListContact**. Запуск этой процедуры можно осуществить (как и в предыдущем задании) с помощью функции **SleepEx**. В самой процедуре завершения ожидающего таймера уже можно завершить зациклившийся поток с помощью функции **TeminateThread (**раздел 6.4**).** Внесите изменения в программу клиента **ClientT**, созданную в предыдущих заданиях для тестирования, чтобы она осуществляла обмен данными с сервером заведомо больше времени, чем одна минута и используйте эту программу для тестирования. Убедитесь в работоспособности созданного механизма предупреждения зацикливания обслуживающих потоков.

Примечание. Применение функции **TeminateThread** может привести к проблемам в работе сервера. Дело в том, что такое завершение потока не гарантирует корректного освобождения всех используемых потоком ресурсов. Правильнее было бы использовать специальное API обслуживающих серверов (это уже обсуждалось в примечаниях выше), которое бы могло корректно завершать работу потока.

7.9.14. Синхронизация потоков AcceptServer и DispatchServer

Задание 21. Поток **DispatchServer** постоянно сканирует список **ListContact** в поиске подключившегося, но не получившего обслуживания клиентского подключения. Очевидно, что сканировать список, нужно только после того, как осуществится очередное подключение в потоке **AcceptServer**. Ознакомьтесь с принципами применения механизмом событий, изложенными в разделе 6.12 пособия. Примените этот механизм для синхронизации потоков **AcceptServer** и **DispatchServer**. Событие должно быть создано в потоке **AcceptServer**, а момент его наступления должен контролироваться в потоке **DispatchServer**. Наступление события должно соответствовать подключению очередного клиента в потоке **DispatchServer**.

7.9.15. Реализация потока ConsolePipe

Задание 22. Поток **ConsolePipe** должен создаваться и запускаться в функции **main –** после запуска потоков **AcceptServer, DispatchServer** и **GarbageCleaner**. Как уже описывалось выше, основным назначением этого потока является ввод команд управления, введенных оператором удаленной консоли **RConsole,** а также вывод на консоль **RConsole** простейших диагностических сообщений. При этом предполагается, что обмен информацией между **ConsolePipe** и **RConsole** будет происходить по именованному каналу. Полученную от удаленной консоли команду функция **ConsolePipe** должна в коды команд понятные потокам-получателям **AcceptServer, DispatchServer, GarbageCleaner** и помещена в общую для этих потоков область памяти (об этом уже говорилось выше). Поток-

получатель команды управления, после считывания команды из общей памяти, должен поместить туда код служебной команды **getcommand**. Поток **ConsolePipe** циклически проверяет общую область памяти потоков, и после получения команды **getcommand** запрашивает следующую команду управления сервером. По аналогии с другими потоками, общую память для хранения команд, следует передать потоку **ConsolePipe** при запуске в параметре потоковой функции. Ознакомьтесь с принципами организации обмена данными по именованному каналу, описанными в четвертой главе пособия. Разработайте потоковую функцию **ConsolePipe,** позволяющую вводить команды управления **start** и **stop (**таблица 7.9.3**)**. При получении этих команд, выведите диагностирующие сообщения на экран консоли сервера. Используйте результаты практической работы № 6. Символическое имя канала должно быть передано, как параметр функции **main** (это будет третий параметр функции **main**, первые два: номер порта и имя файла динамической библиотеки). Используйте для отладки программу **CllientNP**, разработанную в практической работе № 6. Убедитесь в том, что сервер реагирует на команды **start** и **stop (**на экране консоли сервера должны быть соответствующие диагностические сообщения).

Задание 23. Доработайте функцию **ConsolePipe** таким образом, чтобы она могла принимать и обрабатывать все возможные команды, приведенные в таблице 7.9.3. Если код пересланной команды является корректным, обеспечьте отправку в адрес удаленной консоли (пока ее функцию выполняет программа **CllientNP**) это же код, в ином случае оправьте сообщение ***nocmd***. Убедитесь в работоспособности функции **ConsolePipe**.

7.9.16. Обработка команд управления в потоке AcceptServer

Задание 24. Для обработки команд управления в рамках потока **AcceptServer** предусмотрена специальная функция **CommandsCycle (**или ее аналог**),** которая уже рассматривалась в разделе 7.9.6. Доработайте эту функцию так, чтобы она реализовывала выполнение всех команд управления, приведенных в таблице 7.9.3, кроме команды **statistics**. Убедитесь, что сервер выполняет все вводимые команды управления.

7.9.17. Сбор и вывод статистических данных о работе сервера

Задание 25. Сервер **ConcurrentServer** должен поддерживать следующую статистику: общее количество подключений с начала работы сервера, общее количество отказов в обслуживании (по всему спектру причин суммарно) за все время работы сервера, количество подсоединений которые завершились успешно, количество активных подключений на момент запроса. Создайте необходимые переменные-счетчики, в которых будет отражаться данные статистики. Определите программные компоненты

сервера, которые должны участвовать в сборе статистики. Так как эти компоненты работают асинхронно, используете для изменения значений переменных-счетчиков механизм атомарных операций, описанный в разделе 6.8. пособия. Реализуйте выполнение команды управления **statistics**. Убедитесь в корректном подсчете статистики.

7.9.18. Разработка обслуживающих потоков

Задание 26. До сих пор для отладки сервера использовался только один обслуживающий поток **EchoServer**, который подключался для всех возможных запросов (таблица 7.9.4). Проанализируйте структуру функций обслуживающих потоков и разработайте спецификацию потоковой функции обслуживающего потока. Разработайте API, которое бы инкапсулировало все функции Windows API и Winsock2 API. Используя разработанную спецификацию и применив API обслуживающего потока, реализуйте все потоковые функции обслуживающих потоков в соответствии с таблицей 7.9.4 в рамках динамической библиотеки и откорректируйте таблицу **TableService.** Убедитесь в работоспособности сервера.

7.9.19. Навигация сервера в локальной сети

Задание 27. Доработайте функции **GetRequestFromClient** и **PutAnswerToClient** разработанные в практической работе № 4 для того, чтобы исправить отмеченные в примечании к разделу 7.5.5 недостатки. Используйте режим работы сокета без блокировки (раздел 6.9 пособия). Проверьте работоспособность функций в рамках распределенного приложения **ClientB-ServerB**, разработанного в практической работе № 4.

Задание 28. Реализуйте новый поток (потоковую функцию) **ResponseServer** в рамках сервера **ConcurrentServer**. Поток должен быть создан и запущен в функции **main** сервера и предназначен для приема позывного на широковещательные запросы клиента, предназначенные для поиска сервера в локальной сети. Кроме того поток **ResponseServer** должен сформировать ответ: приглашение для подключения. Используйте программу **ServerB** (практическая работа № 4) и доработанные в предыдущем задании функции **GetRequestFromClient** и **PutAnswerToClient**. Позывной сервера и номер UDP-порта для приема широковещательных сообщений следует сделать параметрами сервера. Продумайте и реализуйте механизмы управления потоком **ResponseServer**. Используйте программу **ClientB** для проверки работоспособности потока **ResponseServer**.

7.9.20. Установка приоритетов потоков сервера

Задание 29. Ознакомьтесь с принципами использования системы приоритетов в операционной системе Windows, описанными в разделе 6.10 пособия. Установите приоритеты для потоков **AcceptServer**, **DispatchServer**, **ConsolePipe**, **GarbageCleaner**, **ResponseServer** и для обслуживающих потоков. Убедитесь в работоспособности сервера **ConcurrentServer**.

Задание 30. Предложите методы динамического назначения приоритетов для потоков сервера, позволяющие управлять производительностью сервера **ConncurrentServer**.

7.10. Практическая работа № 9. Разработка клиента параллельного сервера

7.10.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков проектирования API и разработки на его основе программы – клиента параллельного сервера.

Результатом практической работы является разработанный набор функций (API), предназначенный для разработки клиентских программ взаимодействующих с сервером **ConncurrentServer** .

7.10.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в третьей, пятой и шестой главах пособия.

7.10.3. Разработка API клиента

Задание 1. Разработайте спецификацию (протокол), определяющую принципы взаимодействия клиентских программ с сервером **ConncurrentServer**. Разработай API предназначенный для использования на стороне клиента и инкапсулирующий все детали взаимодействия клиента с сервером.

7.10.4. Разработка клиентских приложений

Задание 2. Доработайте функцию **GetServer** разработанную в практической работе № 4 для того, чтобы исправить отмеченные в примечании к разделу 7.5.5 недостатки. Используйте режим работы сокета без блокировки (раздел 6.9 пособия). Проверьте работоспособность функции в рамках распределенного приложения **ClientB-ServerB**, разработанного в практической работе № 4. Включите функцию **GetServer** в состав API клиента.

Задание 3. Разработайте с применением API, разработанным в предыдущем задании, три клиентских программы **ClientEcho**, **ClientTime** и **ClientRand,** предназначенных для тестирования сервера **ConncurrentServer**. Программа **ClientEcho,** предназначена для тестирования запроса **echo**, **ClientTime** для тестирования запроса **time** и **ClientRand** для тестирования запроса **rand**. Все программы должны подключаться к серверу, отыскав его IP-адрес помощью широковещательных сообщений, корректным и некорректным образом осуществлять запросы, проверять на максимальное время работы и т.д. и т.п. Протестируйте работу клиентского API .

7.11. Практическая работа № 10. Разработка удаленной консоли

7.11.1. Цель и задачи работы

Основной целью практической работы является приобретение навыков проектирования API и разработки на его основе программы – удаленной консоли параллельного сервера.

Результатом практической работы является разработанный набор функций (API), предназначенный для разработки программы, реализующей клиентскую сторону консоли сервера **ConncurrentServer**, а также программа **RConsole,** разработанную с применением этого API.

7.11.2. Теоретические сведения

Теоретические сведения необходимые для выполнения практической работы изложены в третьей, пятой и шестой главах пособия.

7.11.3. Разработка API и программы RConsole

Задание 1. Разработайте спецификацию (протокол), определяющую принципы взаимодействия клиентской части консоли управления сервером **ConncurrentServer**. Разработайте API предназначенный для использования на клиентской стороне консоли управления, которая бы инкапсулировала все детали взаимодействия удаленной консоли с сервером.

Задание 2. Создайте с помощью Visual Studio новое консольное приложение **RConsole** (наименование проекта). Используя API, разработанный в предыдущем задании, разработайте клиентскую часть консоли управления сервером **ConncurrentServer**. Программа **RConsole** должна позволять вводить команды управления сервером (таблица 7.9.3) и получать диагностические сообщения сервера. Убедитесь в работоспособности программы **RConsole**.

7.12. Выводы главы

1. Практическая разработка распределенных приложений является сложным и кропотливым процессом. Разработчик таких приложений

должен обладать знаниями в области компьютерных сетей, должен знать и уметь использовать программные интерфейсы, позволяющие организовать обмен данными в сети, обладать навыками параллельного программирования.

2. Построение распределенных приложений требует от разработчика решений, связанных с управлением распределенным приложением.
3. Для обеспечения масшабируемости распределенного приложения необходима разработка стандартов (спецификаций и API) взаимодействия различных его компонент.
4. Распределенные приложения должны обладать высокой степенью параметризации, позволяющей настроить приложение для работы в постоянно изменяющихся условиях распределенной среды.