**7.9. Практическая работа № 8. Разработка параллельного сервера**

**7.9.1. Цель и задачи работы**

Основнойцелью 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 С++ [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 консолиуправления**)**, 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)этом все остальные клиенты, отправившие запрос ожидали бы осблуживание.. 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 версиибиблиотеки **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 **SleepEх** (раздел 6.6), которая будет использоваться в функции **AcceptServer**  для запуска асинхронных процедур. Значение этого параметра должно быть **TRUE**. Значение первого параметра функции **SleepEх** 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** постоянносканирует 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** долженсформировать 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**

Основнойцелью 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**

Основнойцелью 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.