Белгородский государственный технологический университет им. В. Г. Шухова

Кафедра программного обеспечения вычислительной техники  
и автоматизированных систем

## Лабораторная работа №2 по теме: «Программирование протоколов IPX/SPX с использованием библиотеки Winsock»

**Выполнил:**  
студент группы ПВ-31  
Адаменко И. И.

**Проверил:**старший преподаватель  
Федотов Е. А.

Белгород  
2015

**Цель работы:** изучить протоколы IPX/SPX, основные функции библиотеки Winsock и разработать программу для приёма/передачи пакетов.

**Задание:**

1. Разработать программу «Сервер» (на языке программирования C#), которая посылает клиентам некоторое сообщение.
2. Разработать программу «Клиент» (на языке программирования C#), которая ждём от сервера сообщения и затем получает его.
3. Провести анализ функционирования разработанных программ при передаче файла в формате \*.jpg размером 20 Мб (одновременная работа 2, 3 и т. д. приложений на 2, 3 и т. д. компьютерах ЛВС), сделать выводы.
4. Провести сравнительный анализ протоколов IPX и SPX. Сделать выводы.

# Теоретическая часть

Протокол IPX (Internetwork Packet Exchange) является оригинальным протоколом сетевого уровня стека Novell, разработанным в начале 80-х годов на основе протокола Internetwork Datagram Protocol (IDM) компании Xerox.

Протокол IPX соответствует сетевому уровню модели OSI и поддерживает только дейтаграммный (без установления соединений) способ обмена сообщениями. В сети NetWare самая быстрая передача данных при наиболее экономном расходовании памяти реализуется именно протоколом IPX.

Для надёжной передачи пакетов используется протокол транспортного уровня SPX (Sequenced Packet Exchange), который работает с установлением соединения и восстанавливает пакеты при их потере или повреждении. Если по каким-то причинам пакет не дошёл до получателя, выполняется его повторная передача. Следовательно, последовательность отправления совпадает с последовательностью получения пакетов. Обмен пакетами на уровне сеанса связи реализован с помощью протокола SPX, который построен на базе IPX.

Система адресов протокола SPX аналогична системе адресов протокола IPX и также состоит из 3 частей: номера сети, адреса станции и сокета. Протокол SPX использует такой же блок ECB для передачи и приёма пакетов, что и протокол IPX. Однако, пакет, передаваемый при помощи протокола SPX, имеет более длинный заголовок. Дополнительно к 30 байтам стандартного заголовка пакета IPX добавляется ещё 12 байт.

Windows Sockets API (WSA) (сокр. Winsock) — техническая спецификация, которая определяет, как сетевое программное обеспечение Windows будет получать доступ к сетевым сервисам.

Winsock предназначен для использования во всех версиях MS Windows, начиная с 3.0. Для того чтобы программа могла корректно работать с библиотекой Winsock необходимо проверить версию библиотеки Winsock, а также вообще наличие этой библиотеки в системе. Библиотека функции Winsock расположена в файле wsock32.dll (ws2\_32.dll для версии 2.0 этой библиотеки) или winsock.dll для 32-бит и 16-бит приложений соответственно. Также, необходимо подключить заголовочные файлы winsock.h (winsock2.h), а для работы с протоколами IPX и SPX ещё и заголовочный файл wsipx.h.

Функции, необходимые для работы с Winsock в этой лабораторной работе:

* WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData) — инициализирует библиотеку Winsock. В случае успеха возвращает 0.
* WSAGetLastError (void) — возвращает код ошибки, возникшей при выполнении последней операции.
* WSACleanup (void) — осуществляет очистку памяти, занимаемой библиотекой Winsock. Функция деинициализирует библиотеку Winsock и возвращает 0, если операция была выполнена успешно, иначе возвращает SOCKET\_ERROR. Расширенный код ошибки можно получить при помощи функции WSAGetLastError.
* SOCKET (int af, int type, int protocol) — возвращает либо дескриптор созданного сокета, либо ошибку INVALID \_SOCKET.
* bind (SOCKET s, const struct sockaddr FAR\* name, int namelen) — привязывает сокет к какому-нибудь локальному адресу и порту.
* listen (SOCKET s, int backlog) — переводит сокет в состояние «прослушивания» (для протокола SPX).
* connect (SOCKET s, const struct sockaddr FAR\* name, int namelen) — используется процессом-клиентом для установления связи с сервером по протоколу SPX. В случае успешного установления соединения connect возвращает 0, иначе SOCKET\_ERROR и номер ошибки можно получить при помощи функции WSAGetLastError.
* accept (SOCKET s, struct sockaddr FAR\* addr, int FAR\* addrlen) — используется для принятия связи на сокет.
* sendto (SOCKET s, const char FAR \* buf, int len, int flags, const struct sockaddr FAR \* to, int tolen) — осуществляет передачу данных по протоколу IPX.
* send (SOCKET s, const char FAR \* buf, int len, int flags) — осуществляет передачу данных по протоколу SPX.
* recvfrom (SOCKET s, char FAR\* buf, int len, int flags, struct sockaddr FAR\* from, int FAR\* fromlen) — осуществляет приём данных по протоколу IPX.
* recv (SOCKET s, char FAR\* buf, int len, int flags) — осущетвляет приём данных по протоколу SPX.
* Closesocket (SOCKET s) — служит для закрытия сокета.

# Используемые функции

В этой лабораторной работе использовалась встроенная в .NET реализация Winsock, функционал которой заложен в классах Socket и EndPoint.

Используемые методы класса Socket:

* Bind — связывает объект Socket с локальной конечной точкой;
* ReceiveFrom — принимает датаграмму в буфер данных и сохраняет конечную точку;
* SendTo — посылает данные на указанную точку;
* Connect — создаёт подключение к удалённому хосту;
* Receive — возвращает данные из связанного объекта Socket в приёмный буфер;
* Listen — устанавливает объект Socket в состояние прослушивания;
* Accept — создаёт новый объект Socket для заново созданного подключения;
* Send — передаёт данные в подключённый объект Socket;
* Close — закрывает подключение Socket и освобождает все используемые ресурсы.

# Алгоритм работы программы

Сервер IPX:

1. Создание локальной конечной точки, основанной на адресе клиента и выбранном сокете.
2. Создание объекта типа Socket.
3. Последовательное считывание данных с дика и их отправка.
4. Закрытие сокета.

Клиент IPX:

1. Создание локальной конечной точки, основанной на приёме данных со всех узлов сети (бродкаст) и выбранном сокете.
2. Создание объекта типа Socket.
3. Связывание созданного объекта и локальной конечной точки.
4. Последовательное получение данных от сервера и их запись на диск.
5. Закрытие сокета.

Сервер SPX:

1. Создание локальной конечной точки, основанной на адресе клиента и выбранном сокете.
2. Создание объекта типа Socket.
3. Связывание созданного объекта и локальной конечной точки.
4. Устанавливание объекта Socket в режим прослушивания.
5. Создание нового объекта Socket для заново созданного подключения.
6. Последовательное считывание данных с дика и их отправка.
7. Закрытие созданных сокетов.

Клиент SPX:

1. Создание локальной конечной точки, основанной на адресе сервера и выбранном сокете.
2. Создание ещё одной локальной конечной точки, основанной на адресе клиента и выбранном сокете.
3. Создание объекта типа Socket.
4. Связывание созданного объекта и локальной конечной точки.
5. Создание подключения к удалённому серверу.
6. Последовательное получение данных от сервера и их запись на диск.
7. Закрытие сокета.

# Анализ функционирования программы

Было проведено тестирование программы на одном и на двух компьютерах, с использованием:

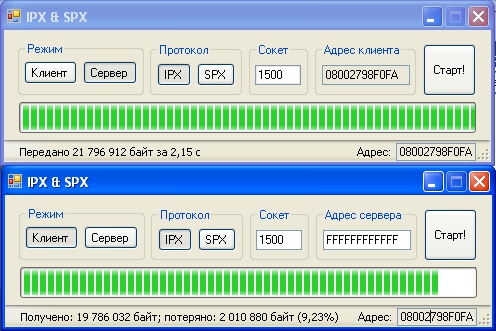
* 1 сервера и 1 клиента;
* 2 серверов и 2 клиентов;
* 3 серверов и 3 клиентов.

Результаты тестирования передачи изображения размером в 21 796 912 байт представлены в таблице ниже:

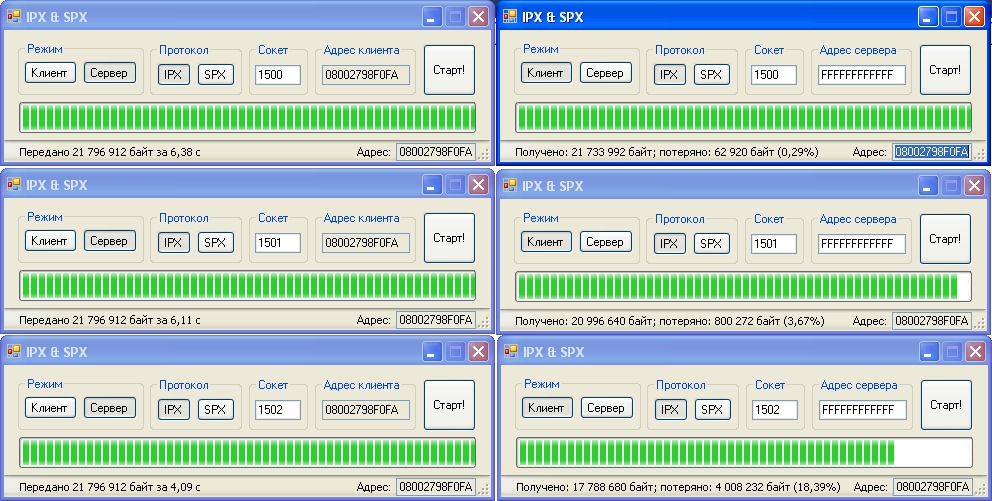
|  |  |  |
| --- | --- | --- |
| **Серверы и клиенты** | **Время** | **Потеряно** |
| **IPX, 1 компьютер** | | |
| 1 сервер, 1 клиент | 2,15 с | 2 010 880 байт, 9,23% |
| 2 сервера, 2 клиента | 3,86 с | 2 409 040 байт, 11,05% |
| 3,51 с | 72 522 байта, 0,33% |
| 3 сервера, 3 клиента | 6,38 с | 62 920 байт, 0,29% |
| 6,11 с | 800 272 байта, 3,67% |
| 4,09 с | 4 008 232 байта, 18,39% |
| **IPX, 2 компьютера** | | |
| 1 сервер, 1 клиент | 14,98 с | 9 496 байт, 0,04% |
| 2 сервера, 2 клиента | 26,01 с | 20 080 байт, 0,09% |
| 24,09 с | 12 520 байт, 0,06% |
| 3 сервера, 3 клиента | 38,30 с | 928 байт, 0,00% |
| 38,05 с | 5 464 байта, 0,03% |
| 35,98 с | 7 984 байт, 0,04% |
| **SPX, 1 компьютер** | | |
| 1 сервер, 1 клиент | 2,37 с | 0 байт, 0% |
| 2 сервера, 2 клиента | 6,58 с | 0 байт, 0% |
| 6,18 с | 0 байт, 0% |
| 3 сервера, 3 клиента | 10,39 с | 0 байт, 0% |
| 10,09 с | 0 байт, 0% |
| 9,24 с | 0 байт, 0% |
| **SPX, 2 компьютера** | | |
| 1 сервер, 1 клиент | 18,07 с | 0 байт, 0% |
| 2 сервера, 2 клиента | 21,08 с | 0 байт, 0% |
| 19,46 с | 0 байт, 0% |
| 3 сервера, 3 клиента | 46,36 с | 0 байт, 0% |
| 46,40 с | 0 байт, 0% |
| 45,13 с | 0 байт, 0% |

# Скриншоты приложения

Тестирование протокола IPX на одной паре клиент-сервер на одном компьютере:



Скриншот программы при тестировании протокола IPX на трёх парах клиент-сервер на одном компьютере:



# Код программы

Класс Utils для работы с клиентом и сервером:

1. **using** System;
2. **using** System.Linq;
3. **using** System.Net.NetworkInformation;
4. **using** System.IO;
5. **using** System.Reflection;
6. **using** System.Threading;
7. **using** System.Diagnostics;
9. **namespace** Lab2
10. {
11. **class** Utils
12. {
13. **public** **static** **string** GetMAC()
14. {
15. **string** macAddresses = **string**.Empty;
17. **foreach** (NetworkInterface nic **in** NetworkInterface.GetAllNetworkInterfaces())
18. {
19. **if** (nic.OperationalStatus == OperationalStatus.Up)
20. {
21. macAddresses += nic.GetPhysicalAddress().ToString();
22. **break**;
23. }
24. }
26. **return** macAddresses;
27. }
29. **public** **static** **long** GetNumMAC()
30. {
31. **return** Convert.ToInt64(GetMAC(), 16);
32. }
34. **public** **static** **void** Starter(**bool** mode, **bool** protocol, **long** node, **ushort** socket, MainForm form)
35. {
36. **short** packetSize = 512;
37. **short** dataSize = Convert.ToInt16(packetSize - 8);
39. **if** (mode == **true**)
40. {
41. Client client;
43. **if** (protocol == **true**)
44. {
45. client = **new** IPXClient(node, socket);
46. }
47. **else**
48. {
49. client = **new** SPXClient(node, socket);
50. }
52. var fs = **new** FileStream(Path.GetDirectoryName(  
     Assembly.GetExecutingAssembly().Location) +
53. @"\img-" + socket + ".jpg", FileMode.Create, FileAccess.Write);
55. client.Start();
57. **byte**[] byteArray = **new** **byte**[packetSize];
58. var maxDataBlock = 21; // mb
59. maxDataBlock = (maxDataBlock \* 1024 \* 1024 / dataSize) \* dataSize;
60. **byte**[] saver = **new** **byte**[maxDataBlock];
62. **long** received = 0;
63. **long** length = 0;
64. var dataBlock = 0;
66. byteArray = client.Receive(packetSize);
68. **if** (byteArray.Any(b => b != 0))
69. {
70. **byte**[] tmp = **new** **byte**[8];
71. Array.Copy(byteArray, dataSize, tmp, 0, 8);
72. length = BitConverter.ToInt64(tmp, 0);
74. **while** (byteArray.Any(b => b != 0))
75. {
76. Array.Copy(byteArray, 0, saver, dataBlock, dataSize);
78. dataBlock += dataSize;
80. **if** (dataBlock == maxDataBlock)
81. {
82. fs.Write(saver, 0, dataBlock);
83. dataBlock = 0;
85. Array.Clear(saver, 0, saver.Length);
86. }
88. byteArray = client.Receive(packetSize);
89. received += dataSize;
91. form.UpdateProgess(Convert.ToInt32((received \* 100) / length));
92. }
94. fs.Write(saver, 0, dataBlock);
95. }
97. client.Stop();
99. **double** lost = length - received;
100. lost = lost < 0 ? 0 : lost;
101. received = received > length ? length : received;
103. form.UpdateStatus("Получено: " + String.Format("{0:N0}",  
      received) + " байт; " + "потеряно: " +
104. String.Format("{0:N0}", Convert.ToInt32(lost)) + " байт " +
105. "(" + String.Format("{0:F2}", lost / length \* 100) + "%)");
107. fs.Close();
108. fs.Dispose();
109. }
110. **else**
111. {
112. Server server;
114. **if** (protocol == **true**)
115. {
116. server = **new** IPXServer(node, socket);
117. }
118. **else**
119. {
120. server = **new** SPXServer(node, socket);
121. }
123. var fs = **new** FileStream(Path.GetDirectoryName(  
      Assembly.GetExecutingAssembly().Location) +
124. @"\img.jpg", FileMode.Open, FileAccess.Read);
126. **byte**[] byteArray = **new** **byte**[packetSize];
128. **long** length = fs.Length;
129. var fSize = BitConverter.GetBytes(length);
130. **long** total = 0;
132. Stopwatch timer = Stopwatch.StartNew();
134. server.Start();
136. **while** (fs.Read(byteArray, 0, dataSize) > 0)
137. {
138. fSize.CopyTo(byteArray, dataSize);
140. server.Send(byteArray);
142. total += dataSize;
144. form.UpdateProgess(Convert.ToInt32((total \* 100) / length));
146. Array.Clear(byteArray, 0, byteArray.Length);
147. }
149. **for** (**int** i = 0; i < 100; i++)
150. {
151. server.Send(byteArray);
152. Thread.Sleep(1);
153. }
155. server.Stop();
157. timer.Stop();
159. form.UpdateStatus("Передано " +   
      String.Format("{0:N0}", length) + " байт " + "за " + String.Format("{0:N2}",
160. Convert.ToDouble(timer.ElapsedMilliseconds) / 1000) + " с");
162. fs.Close();
163. fs.Dispose();
164. }
165. }
166. }
167. }

Класс для работы с IPX клиентом:

1. **using** System.Net.Sockets;
2. **using** System.Net;
4. **namespace** Lab2
5. {
6. **class** IPXClient : Client
7. {
8. **private** Socket \_usingsocket;
9. **private** EndPoint \_endpoint;
11. **public** IPXClient(**long** node, **ushort** socket) : **base**(node, socket)
12. {
13. \_endpoint = **new** NetWare(0, 0xffffffffffff, socket);
14. }
16. **public** **override** **void** Start()
17. {
18. \_usingsocket = **new** Socket(AddressFamily.Ipx, SocketType.Dgram,  
     ProtocolType.Ipx);
19. \_usingsocket.Bind(\_endpoint);
20. }
22. **public** **override** **byte**[] Receive(**short** length)
23. {
24. **byte**[] byteArray = **new** **byte**[length];
26. \_usingsocket.ReceiveFrom(byteArray, **ref** \_endpoint);
28. **return** byteArray;
29. }
31. **public** **override** **void** Stop()
32. {
33. \_usingsocket.Close();
34. }
35. }
36. }

Класс для работы с IPX сервером:

1. **using** System.Net.Sockets;
2. **using** System.Net;
4. **namespace** Lab2
5. {
6. **class** IPXServer : Server
7. {
8. **private** Socket \_usingsocket;
9. **private** EndPoint \_endpoint;
11. **public** IPXServer(**long** node, **ushort** socket) : **base**(node, socket)
12. {
13. \_endpoint = **new** NetWare(0, node, socket);
14. }
16. **public** **override** **void** Start()
17. {
18. \_usingsocket = **new** Socket(AddressFamily.Ipx, SocketType.Dgram,  
     ProtocolType.Ipx);
19. }
21. **public** **override** **void** Send(**byte**[] buf)
22. {
23. \_usingsocket.SendTo(buf, \_endpoint);
24. }
26. **public** **override** **void** Stop()
27. {
28. \_usingsocket.Close();
29. }
30. }
31. }

Класс для работы с SPX клиентом:

1. **using** System.Net.Sockets;
2. **using** System.Net;
4. **namespace** Lab2
5. {
6. **class** SPXClient : Client
7. {
8. **private** Socket \_usingsocket;
9. **private** EndPoint \_endpoint;
10. **private** EndPoint \_localpoint;
12. **public** SPXClient(**long** node, **ushort** socket) : **base**(node, socket)
13. {
14. \_endpoint = **new** NetWare(0, node, socket);
15. \_localpoint = **new** NetWare(0, Utils.GetNumMAC(), socket);
16. }
18. **public** **override** **void** Start()
19. {
20. \_usingsocket = **new** Socket(AddressFamily.Ipx, SocketType.Seqpacket,  
     ProtocolType.Spx);
21. \_usingsocket.Bind(\_localpoint);
23. \_usingsocket.Connect(\_endpoint);
24. }
26. **public** **override** **byte**[] Receive(**short** length)
27. {
28. **byte**[] byteArray = **new** **byte**[length];
30. \_usingsocket.Receive(byteArray);
32. **return** byteArray;
33. }
35. **public** **override** **void** Stop()
36. {
37. \_usingsocket.Close();
38. }
39. }
40. }

Класс для работы с SPX сервером:

1. **using** System.Net.Sockets;
2. **using** System.Net;
4. **namespace** Lab2
5. {
6. **class** SPXServer : Server
7. {
8. **private** Socket \_usingsocket;
9. **private** Socket \_acceptsocket;
10. **private** EndPoint \_endpoint;
12. **public** SPXServer(**long** node, **ushort** socket) : **base**(node, socket)
13. {
14. \_endpoint = **new** NetWare(0, node, socket);
15. }
17. **public** **override** **void** Start()
18. {
19. \_usingsocket = **new** Socket(AddressFamily.Ipx, SocketType.Seqpacket,  
     ProtocolType.Spx);
21. \_usingsocket.Bind(\_endpoint);
22. \_usingsocket.Listen(10);
23. \_acceptsocket = \_usingsocket.Accept();
24. }
26. **public** **override** **void** Send(**byte**[] buf)
27. {
28. **try**
29. {
30. \_acceptsocket.Send(buf);
31. }
32. **catch**
33. {
34. **return**;
35. }
36. }
38. **public** **override** **void** Stop()
39. {
40. \_usingsocket.Close();
41. \_acceptsocket.Close();
42. }
43. }
44. }

Класс для работы с конечными точками и заголовками пакетов:

1. **using** System;
2. **using** System.Net;
3. **using** System.Net.Sockets;
5. **namespace** Lab2
6. {
7. **public** **class** NetWare : EndPoint
8. {
9. **private** **const** **int** SocketSize = 14;
11. **public** **uint** Network;
12. **public** **long** Node;
13. **public** **ushort** Socket;
15. **public** NetWare(**uint** network, **long** node, **ushort** socket)
16. {
17. Network = network;
18. Node = node;
19. Socket = socket;
20. }
22. **public** **override** SocketAddress Serialize()
23. {
24. var address = **new** SocketAddress(AddressFamily.Ipx, SocketSize);
26. **byte**[] network = BitConverter.GetBytes(Network);
28. // GetBytes возвращает биты в обратном порядке!
29. Array.Reverse(network);
31. // первые два байта занимает AddressFamily,
32. // номер сети вставляем в байты 2-5
33. **for** (**int** i = 0; i < 4; i++)
34. {
35. address[i + 2] = network[i];
36. }
38. **byte**[] node = BitConverter.GetBytes(Node);
39. Array.Reverse(node);
41. // адрес рабочей станции (мак) с 6 по 11 байты
42. // node -- это long, поэтому опускаем 0 и 1 байты
43. **for** (**int** i = 0; i < 6; i++)
44. {
45. address[i + 6] = node[i + 2];
46. }
48. **byte**[] socket = BitConverter.GetBytes(Socket);
49. Array.Reverse(socket);
51. // сокет -- это 12 и 13 байты
52. **for** (**int** i = 0; i < 2; i++)
53. {
54. address[i + 12] = socket[i];
55. }
57. **return** address;
58. }
60. **public** **override** AddressFamily AddressFamily
61. {
62. **get**
63. {
64. **return** System.Net.Sockets.AddressFamily.Ipx;
65. }
66. }
68. **public** **override** EndPoint Create(SocketAddress socketAddress)
69. {
70. **byte**[] network = **new** **byte**[4];
72. **for** (**int** i = 0; i < network.Length; i++)
73. {
74. network[i] = socketAddress[i + 2];
75. }
77. **byte**[] node = **new** **byte**[8];
79. **for** (**int** i = 0; i < node.Length; i++)
80. {
81. node[i] = socketAddress[i + 6];
82. }
84. **byte**[] socket = **new** **byte**[2];
86. **for** (**int** i = 0; i < socket.Length; i++)
87. {
88. socket[i] = socketAddress[i + 12];
89. }
91. **return** **new** NetWare(BitConverter.ToUInt32(network, 0),  
     BitConverter.ToInt64(node, 0), BitConverter.ToUInt16(socket, 0));
92. }
93. }
94. }

# Ответы на вопросы

1. **Назовите отличия протокола SPX от IPX**SPX используется для надёжной передачи данных, поскольку он восстанавливает пакеты при их потере или повреждении. Если по каким-то причинам пакет не дошёл до клиента, он отправляется повторно. Т. е., SPX контролирует передачу, а IPX — нет.  
     
   SPX использует такой же блок ECB для передачи и приёма пакетов, что и IPX, но пакет имеет более длинный заголовок (на 12 байт длиннее).
2. **Что представляет собой библиотека Winsock?**Библиотека функции Winsock расположена в файле wsock32.dll (ws2\_32.dll для версии 2.0 этой библиотеки) или winsock.dll для 32-бит и 16-бит приложений соответственно. Также, необходимо подключить заголовочные файлы winsock.h (winsock2.h), а для работы с протоколами IPX и SPX ещё и заголовочный файл wsipx.h.
3. **Какие действия необходимо выполнить для корректного создания сокета, настроенного на приём сообщений?**Необходимо создать сокет функцией Socket, передав ей необходимые параметры в зависимости от используемого протокола. Далее привязать этот сокет с помощью функции Bind к какому-нибудь локальному адресу и порту.   
     
   Если процесс использует протокол SPX, то также необходимо воспользоваться функцией Connect для установления связи с сервером.
4. **Назовите функции библиотеки Winsocket, используемые для отправки и приёма сообщений через протокол IPX.**
   1. WSAStartup (WORD wVersionRequested, LPWSADATA lpWSAData) — инициализирует библиотеку Winsock. В случае успеха возвращает 0.
   2. WSAGetLastError (void) — возвращает код ошибки, возникшей при выполнении последней операции.
   3. WSACleanup (void) — осуществляет очистку памяти, занимаемой библиотекой Winsock. Функция деинициализирует библиотеку Winsock и возвращает 0, если операция была выполнена успешно, иначе возвращает SOCKET\_ERROR. Расширенный код ошибки можно получить при помощи функции WSAGetLastError.
   4. SOCKET (int af, int type, int protocol) — возвращает либо дескриптор созданного сокета, либо ошибку INVALID \_SOCKET.
   5. bind (SOCKET s, const struct sockaddr FAR\* name, int namelen) — привязывает сокет к какому-нибудь локальному адресу и порту.
   6. sendto (SOCKET s, const char FAR \* buf, int len, int flags, const struct sockaddr FAR \* to, int tolen) — осуществляет передачу данных.
   7. recvfrom (SOCKET s, char FAR\* buf, int len, int flags, struct sockaddr FAR\* from, int FAR\* fromlen) — осуществляет приём данных.
   8. Closesocket (SOCKET s) — служит для закрытия сокета.
5. **Что представляет собой структура sockaddr?**

Структура sockaddr имеет вид:

typedef struct sockaddr\_ipx {

short sa\_family;

char sa\_netnum[4];

char sa\_nodenum[6];

unsigned short sa\_socket;

} SOCKADDR\_IPX, \*PSOCKADDR\_IPX,FAR \*LPSOCKADDR\_IPX;

Параметры структуры:  
— sin\_family — семейство протоколов;

— sin\_netnum — номер сети;

— sin\_nodenum — номер узла;

— sa\_socket – номер сокета.

1. **Принцип построения программы «клиент-сервер» с использованием протокола SPX.**
   1. Сервер SPX:
      1. Создание локальной конечной точки, основанной на адресе клиента и выбранном сокете.
      2. Создание объекта типа Socket.
      3. Связывание созданного объекта и локальной конечной точки.
      4. Устанавливание объекта Socket в режим прослушивания.
      5. Создание нового объекта Socket для заново созданного подключения.
      6. Последовательное считывание данных с дика и их отправка.
      7. Закрытие созданных сокетов.
   2. Клиент SPX:
      1. Создание локальной конечной точки, основанной на адресе сервера и выбранном сокете.
      2. Создание ещё одной локальной конечной точки, основанной на адресе клиента и выбранном сокете.
      3. Создание объекта типа Socket.
      4. Связывание созданного объекта и локальной конечной точки.
      5. Создание подключения к удалённому серверу.
      6. Последовательное получение данных от сервера и их запись на диск.
      7. Закрытие сокета.