МИНИСТЕРСТВО ПРОСВЕЩЕНИЯ
ПРИДНЕСТРОВСКОЙ МОЛДАВСКОЙ РЕСПУБЛИКИ

ПРИДНЕСТРОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
им. Т.Г. ШЕВЧЕНКО

Инженерно-технический институт
Инженерно-технический факультет

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

ТЕХНИЧЕСКАЯ ИНФОРМАТИКА

(СЕТЕВОЕ ПРОГРАММИРОВАНИЕ)

Методические указания
к лабораторным работам

дневной и заочной формы обучения
по направлению подготовки
09.03.04 «Программная инженерия»

Тирасполь 2018

УДК 004.7(072.8)
ББК 3972.5р30
Т38

Составители:
О.С. Белоконь, старший преподаватель кафедры программного обеспечения вычислительной техники и автоматизированных систем
М.В. Нижегородова, к.п.н., доцент кафедры программного обеспечения вычислительной техники и автоматизированных систем

Рецензенты:
С.Г. Федорченко, к.т.н., доцент, зав. кафедрой программного обеспечения вычислительной техники и автоматизированных систем.
Е.А. Левицкий, инженер–программист ООО «Хелес».

Т38 **Техническая информатика (Сетевое программирование):** методические указания к лабораторным работам /сост.: Белоконь О.С., Нижегородова М.В. – Тирасполь: Изд-во Приднестр. ун-та, 2018. – 123с.

Настоящие методические указания имеют целью изучение приемов, средств и методов сетевого прграммирования. Представлены лабораторные работы, рассматривающие способы применения в сетевых приложениях асинхронных сокетов, протокола UDP, реализация сетевого приложения для пиринговых сетей и применения технологии MapReduce для распределенных сетевых приложений. Составлены в соответствии с программой изучаемого курса.

Предназначены для студентов направления 09.03.04 «Программная инженерия» ИТИ ПГУ. Могут быть использованы, при соответствующем изменении объёма, и для студентов других направлений.

УДК 004.7(072.8)
ББК 3972.5р30

Рекомендовано Научно-методическим советом ПГУ им. Т.Г. Шевченко

.

Лабораторная работа 1

Тема: «Асинхронное программирование сокетов»

Асинхронная организация кода, требует наличие синхронизации – нельзя работать с сокетом, если не закончено установление соединения, может возникнуть блокировка.

Для реализации синхронизации порождаемых потоков используется класс *ManualResetEvent.*

Задание 1: Создать асинхронное приложение-клиент.

```
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace AsynClient
{
 class Program
  {
   public static string theResponce = "”’
   public static byte[] buffer = new byte[1024];
```

При создании объектов *ManualResetEvent* указывается состояние *false*, это означает, что состояние не установленно

```
   public static ManualResetEvent ConnectDone=new ManualResetEvent(false);
   public static ManualResetEvent SendDone=new ManualResetEvent(false);
   public static ManualResetEvent ReceiveDone= new ManualResetEvent(false);

   static void Main (string[] args)
    { try
     {  Thread thr = Thread.CurrentThread;
        byte[] byteData = new byte[1024];
        IPHostEntry ipHost = Dns.Resolve(“127.0.0.1”);
        IPAddress ipAdr = ipHost.AddressList[0];
        IPEndPoint endPoint = new IPEndPoint(ipAdr, 11000);
        Socket sClient = new Socket(AddressFamily.InterNetwork,
                                    SocketType.Stream, ProtocolType.Tcp);
```

Метод *BeginConnect* начинает асинхронно устанавливать соединение с удаленным хостом. Для него требуется метод обратного вызова, в котором реализован представитель *AsyncCallBack*.

Метод обратного вызова должен вызвать метод *EndConnect()*, который завершит запрос на соединение и вернет соединенный сокет.

```
sClient.BeginConnect(endPoint,new AsyncCallback(ConnectCallBack),
                                                        sClient);
```

Этот метод блокирует текущий поток, пока объект не перейдет в сигнальное состояние.

```
ConnectDone.WaitOne();
```

Соединившись с сервером, приступаем к отправке данных. Определим сообщение, которое пошлем серверу.

```
string data = "Это тест";
for (int i=0; i<72; i++)
 {
  data +=i.ToString() + ":" + (new string('=', i));
  byteData = Encoding.ASCII.GetBytes(data+"<TheEnd>");
 }
sClient.BeginSend(byteData, 0, byteData.Lenght, 0,
                    new AsyncCallback(SendCallBack), sClient);
```

Первый параметр – массив байтов, содержащий данные для отправки, второй – позиция в буффере, от которой нужно начать посылать данные, третий – размер буфера, последний используется для сохранения информации о состоянии.

Метод *BeginSend()* вызывает функцию обратного вызова, переданную посредством делегата *AsyncCallback*.

Для иллюстрации асинхронной работы, выполним другую обработку. В цикле переводим текущий поток в ожидание на на сотую долю секунды – имитация вычислений тратящих процессорное время.

```
for (int i=0; i<5; i++)
 {
  Console.Writeln(i);
  Thread.Sleep(10);
 }
SendDone.WaitOne();
sClient.BeginReceive(buffer, 0, buffer.Length, 0,
                 new AsyncCallback(ReceiveCallBack), sClient);
ReceiveDone.WaitOne();
Console.WriteLine("В ответ получено {0}",theResponce);
Console.Read();
sClient.Shutdown(SocketShutdown.Both);
sClient.Close();
}
```

```
 catch(Exception e)
 {
  Console.WriteLine(e.ToString());
 }
}
public static void ConnectCallBack(IAsyncResult ar)
{
```

Инспектируем поток, в котором выполняется асинхронный метод, он покажет, что выполнение метода идет в фоновом потоке. Параметр *IAsyncResult* содержит информацию о состоянии асинхронной операции.

```
 Thread thr = Thread.CurrentThread;
 Console.WriteLine("ConnectCallBack Thread State: "
                                          + thr.ThreadState);
 Socket sClient = (Socket)ar.AsyncState;
```

Для завершения асинхронного запроса.

```
 sClient.EndConnect(ar);
 Console.WriteLine("Сокет соединен с {0}",
 sClient.RemoteEndPoint.ToString());
```

Этим вызовом сообщаем основному потоку, что завершили установление соединения.

```
 ConnectDone.Set();
}
public static void SendCallBack(IAsyncResult ar)
{
 Thread thr = Thread.CurrentThread;
 Console.WriteLine("SendCallBack Thread State: " + thr.ThreadState);
 Socket sClient = (Socket)ar.AsyncState;
 int bytesSend = sClient.EndSend(ar);
 Console.WriteLine("Отправлено {0} байт серверу", bytesSend);
 SendDone.Set();
}
public static void ReceiveCallBack(IAsyncResult ar)
{
 Thread thr = Thread.CurrentThread;
 Console.WriteLine("ReceiveCallBack Thread State: " + thr.ThreadState);
 Socket sClient = (Socket)ar.AsyncState;
 int bytesRead = sClient.EndReceive(ar);
```

Возвращает число отправленных байтов, поэтому проверяем, остались ли еще данные в очереди. Полученные данные сохраняем в *theResponce*.

```
 if (bytesRead > 0)
```

```
      {
       theResponce += Encoding.ASCII.GetString(buffer, 0, bytesRead);
       sClient.BeginReceive(buffer,   0,   buffer.Length,   0,   new
       AsyncCallback(ReceiveCallBack), sClient);
      }
      Else
      {
       ReceiveDone.Set();
      }
     }
   }
}
```

Вывод на консоль асинхронного – клиента, когда он запускается вместе с синхронным сервером, имеет следующий вид (рисунок 1):

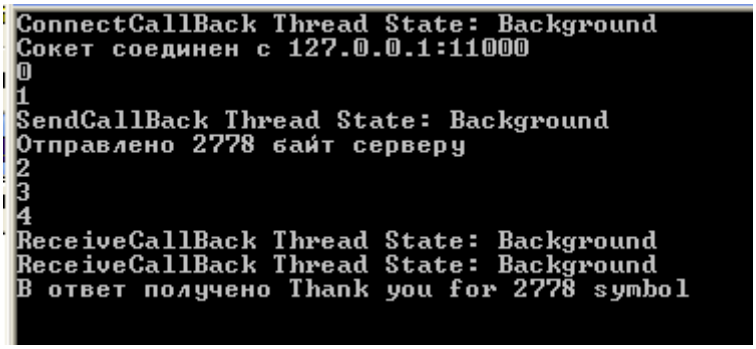

Рисунок 1 – Вывод на консоль асинхронного – клиента

Задание 2. Создать асинхронное приложение – сервер.

Для асинхронного приема соединений асинхронному приложению – серверу необходимо использовать метод *BeginAccept()*, а после установления соединения для получения и отправки данных используются методы *BeginSend()* и *BeginReceive()*.

После вызова метода *BeginAccept()*, устанавливаем событие в режим ожидания, чтобы другой поток приложения мог выполняться пока не будет установлено соединение.

Если этого не сделать, то из-за асинхронной природы сервера, приложение раньше завершится прежде клиента.

```
namespace AsynClient
{
 class Program
  {//буфер для получения и отправки данных
   public static byte[] buffer = new byte[1024];
   //класс события для поддержки синхронизации
   public static ManualResetEvent socketEvent=new ManualResetEvent(false);
```

```
static void Main (string[] args)
 {
  Console.WriteLine("Main Thread ID; " + AppDomain.GetCurrentThreadId());
  byte[] bytes = new byte[1024];
  IPHostEntry ipHost = Dns.Resolve(Dns.GetHostName());
  IPAddress ipAdr = ipHost.AddressList[0];
  IPEndPoint localend = new IPEndPoint(ipAdr, 11000);
  Socket sListener = new Socket(AddressFamily.InterNetwork,
                              SocketType.Stream, ProtocolType.Tcp);
  sListener.Bind(localend);
  sListener.Listen(10);
  Console.WriteLine("Ожидание соединения...");
  AsyncCallback aCallBack = new AsyncCallback(AcceptCallBack);
  //асинхронная функция, дающая согласие на соединение
  sLictener.BeginAccept(aCallBack, sListener);
  //ждем пока другие потоки закончат работу
  socketEvent.WaitOne();
 }
 public static void AcceptCallBack(IAsyncResult ar)
 {
  Console.WriteLine("AcceptCallBack thread ID: "+AppDomain.GetCurrentThreadId());
  //сокет для получения запросов
  Socket listener = (Socket)ar.AsyncState;
  //новый сокет
  Socket handler = listener.EndAccept(ar);
  Handler.BeginReceive(buffer, 0, buffer.Length, 0,
                     new AsyncCallback(ReceiveCallBack), handler);
 }
 public static void ReceiveCallBack(IAsyncResult ar)
 {
  Console.WriteLine("ReceiveCallBack thread ID: "+AppDomain.GetCurrentThreadId());
  string content = String.Empy;
  //новый сокет
  Socket handler = (Socket)ar.AsyncState;
  //если данные есть ...
  if (bytesRead > 0)
  {//присоединяем их к основной строке
   content += Encoding.ASCII.GetString(buffer, 0, bytesRead);
   //если мы находим символ конца сообщения <TheEnd>
   if (content.IndexOf("<TheEnd") > -1)
   {
    Console.WriteLine("Считано {0} байт из сокета\n Данные: {1}",
                                          content.Length, content);
    byte[] bytedata = Encoding.ASCII.GetBytes(content);
    //отправляем данные обратно клиенту
    handler.BeginSend(bytedata, 0, bytedata.Length, 0,
                       new AsyncCallback(SendCallBack), handler);
   }
   else
   {//иначе получаем оставшиеся данные
    handler.BeginReceive(buffer, 0, buffer.Length, 0,
                     new AsyncCallback(ReceiveCallBack), handler);
   }
```

```
      }
     }
     public static void SendCallBack(IAsyncResult ar)
     {
      Console.WriteLine("SendCallBack thread ID: "+AppDomain.GetCurrentThreadId());
      Socket handler = (Socket)ar.AsyncState;
      //отправляем данные обратно клиенту
      int bytesSend = handler.EndSend(ar);
      Console.WriteLine("Клиенту отправлено {0} байт",bytesSend);
      //закрываем сокет
      handler.Shutdown(SocketShutdown.Both);
      handler.Close();
      //устанавливаем событие для основного потока
      Console.ReadKey();
      socketEvent.Set();
     }
    }
}
```

Задание 3. Создать два клиента, передающих сообщения посредством сервера. Сервер ведет подсчет переданных данных и предоставляет их по требованию клиента.

Задание 4. Выполнить индивидуальное задание, полученное у преподавателя.

Варианты индивидуальных заданий к лабораторной работе.

В серверном приложении параллельно работают два потока основной, который проводит действия и выводит результат этих действий на экран и рабочий, который принимает сообщения от клиента, проводит вычислительные действия и выводит данные на экран. В результате в серверном приложении выводятся данные одновременно двух потоков.

ВАРИАНТ 1. Основной поток генерирует числа от 1 до 1000000. Клиентское приложение запрашивает у пользователя число *X* и отправляет серверу. Рабочий поток сервера получает значение *X*, возводит его в квадрат и выводит на экран.

ВАРИАНТ 2. Основной поток генерирует четные числа от 1 до 1000000. Клиентское приложение генерирует число *X* в диапазоне от 1 до 100 и отправляет серверу. Рабочий поток сервера получает значение *X*, возводит его в квадрат и выводит на экран.

ВАРИАНТ 3. Основной поток генерирует числа от 1 до 500000. Клиентское приложение запрашивает у пользователя число *X* и отправляет серверу. Рабочий поток сервера получает значение *X*, возводит его в куб и выводит на экран.

ВАРИАНТ 4. Основной поток генерирует четные числа от 1 до 1000000. Клиентское приложение генерирует число *X* в диапазоне от 1 до 200 и отправляет серверу. Рабочий поток сервера получает значение *X*, возводит его в куб и выводит на экран.

ВАРИАНТ 5. Основной поток генерирует числа от 1 до 100. Клиентское приложение запрашивает у пользователя число *X* и отправляет серверу. Рабочий поток сервера получает значение *X* и генерирует значение *Y* от 1 до 20, вычисляет *X*+*Y* и результат выводит на экран.

ВАРИАНТ 6. Основной поток генерирует четные числа от 1 до 100. Клиентское приложение генерирует число *X* в диапазоне от 1 до 200 и отправляет серверу. Рабочий поток сервера получает значение *X* и генерирует значение *Y*, вычисляет *X***Y* и результат выводит на экран.

ВАРИАНТ 7. Основной поток генерирует числа от 1 до 100000. Клиентское приложение запрашивает у пользователя число *X* и отправляет серверу. Рабочий поток сервера получает значение *X* и генерирует значение *Y* от 1 до 20, вычисляет X^2+*Y* и результат выводит на экран.

ВАРИАНТ 8. Основной поток генерирует четные числа от 1 до 100. Клиентское приложение генерирует число *X* в диапазоне от 1 до 200 и отправляет серверу. Рабочий поток сервера получает значение *X* и генерирует значение *Y*, вычисляет X^2**Y*-2*(*X*+*Y*) и результат выводит на экран.

ВАРИАНТ 9. Основной поток перебирает буквы латинского алфавита. Клиентское приложение запрашивает у пользователя число *X* и отправляет серверу. Рабочий поток сервера получает значение *X* и выводит строку конкатенации с буквами, и результат выводит на экран.

ВАРИАНТ 10. Основной поток перебирает буквы латинского алфавита. Клиентское приложение запрашивает у пользователя слово и отправляет серверу.

Рабочий поток сервера получает слово, определяет количество гласных букв и выводит результат на экран.

ВАРИАНТ 11. Основной поток перебирает буквы русского алфавита. Клиентское приложение запрашивает у пользователя слово и отправляет серверу. Рабочий поток сервера получает слово и если слово начинается с гласной буквы, меняет регистр выводимых в основном потоке букв на экране.

ВАРИАНТ 12. Основной поток перебирает буквы русского алфавита. Клиентское приложение запрашивает у пользователя слово и отправляет серверу. Рабочий поток сервера получает слово, переворачивает слово и выводит на экран столько раз, сколько букв в полученном слове.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Преимущества асинхронного программирования сокетов.
2. Методы, используемые в асинхронном приложении – клиента и приложении – сервера.
3. Как реализовано параллельное взаимодействие потоков?
4. Как возвращается управление текущему потоку?
5. Класс, представители которого, помогают организовать синхронизацию.
6. Отличие передачи данных в синхронном приложении от асинхронного.

Лабораторная работа 2

Тема: «Создание приложений интерактивного форума, использующее *UDP*»

UDP использует следующие известные порты:

Номер порта	Описание
15	*NETSTAT* – состояние сети
53	*DNS* – сервер доменных имен
69	*TFTP* – простейший протокол передачи файлов
137	Служба имен *NetBIOS*
138	Дейтаграммная служба *NetBIOS*
161	*SNMP*

Однонаправленный *IP*-адрес уникально определяет хост в сети, тогда как групповой IP-адрес определяет конкретную группу адресов в сети. Широковещательные адреса получаются и обрабатываются всеми хостами локальной сети или конкретной подсети.

Среда *Microsoft.NET Framework* предоставляет класс *UdpClient* для реализации в сети протокола *UDP*.

Алгоритм:

Создайте экземпляр *UdpClient*. Далее через метод *Connect()* создается соединение с удаленным хостом. Особенность данного протокола, что метод соединения не устанавливает соединение с удаленным хостом до отправки или получения данных.

Далее для отправки и получении данных используется методы *Send()* и *Receive()*. В конце закрывается соединение методом *Close ()*.

Если создать экземпляр класса следующим образом:

```
UdpClient udpClient = new UdpClient ();
```

В этом случае будут использоваться произвольный свободный порт и *IP*-адрес 0.0.0.0.

Можно создать объект *UdpClient*, указав в параметре номер порта. Если номер порта находится вне пределов, указанных полями *MinPort* и *MaxPort* класса *IPEndPoint*, или порт занят, то порождается исключение.

```
//Создаем UdpClient, используя номер порта
try
{UdpClient udpClient = new UdpClient (5001);
}
```

```
catch (ArgumentOutOfRangeExeption e)
{
  Console.WriteLine ("Неправильный номер порта");
}
catch (SocketExeption e)
{
  Console.WriteLine ("Порт уже используется");
}
```

В классе *UdpClient* есть два защищенных свойства, доступных из класса, в котором они объявлены, и из производного класса. Это означает, что к защищенным свойствам нельзя напрямую обращаться из экземпляра *UdpClient*. Свойство *Active* используется для проверки соединения с удаленным хостом. Это свойство возвращает значение *true*, если соединение активно.

Свойство *Client* используется для получения базового объекта *Socket*, оно позволяет обращаться к лежащему в основе сокету и, следовательно, всем членам класса *Socket,* недоступным через класс *UdpClient*. Например, в свойстве *Blocking* класса *Socket* можно указать, находится ли сокет в блокирующем режиме. С помощью доступных членов класса *UdpClien*t этого сделать нельзя. В следующем примере используются оба этих свойства. Реализуйте его:

```
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace UDP_prim
{
 class UdpDerived : UdpClient
 {
  public void insideSocket()
  {
   //метод обращается к защищенному свойству Active, принадлежащему
   //базовому классу UdpClient, чтобы определить наличие соединения
   if (this.Active)
   {
    Console.WriteLine("Соединение активировано!");
    //получаем базовый объект Socket
    Socket richSoc = this.Client;
    Console.WriteLine("Тип сокета: {0}",richSoc.SocketType.ToString());
    Console.ReadKey();
   }
  }
  [STAThread]
  static void Main(string[] args)
  {
   UdpDerived instance = new UdpDerived();
   //соединяемся, чтобы проверить свойство Active
```

```
   IPEndPoint ipEndP = new IPEndPoint(IPAddress.Parse("127.0.0.1"),5001);
   instance.Connect(ipEndP);
   //вызываем защищенный метод
   instance.insideSocket();
  }
 }
```

Приложение интерактивного форума, использующее *UDP*

Пусть диалоговое приложение использует отдельный поток, чтобы слушать сообщения от удаленных хостов. Это приложение разделено на три логические части.

В первой части пользователю предлагается ввести информацию о локальных и удаленных портах и удаленном *IP*-адресе, которые он хочет использовать.

Во второй части приложение слушает входящие данные от удаленного хоста. Метод *Receive()* проверяет наличие входящих дейтаграмм и блокирует поток, пока от удаленного хоста не поступит сообщение. Чтобы отделить этот процесс от основной последовательности действий, создается новый поток.

Третий блок в приложении принимает данные, введенные пользователем, и отправляет их указанному удаленному порту. Он выполняется на основном потоке, пока рабочий поток продолжает слушать входящие сообщения.

Реализуйте следующий код чата:

```
using System.Net;
using System.Net.Sockets;
using System.Threading;

namespace ChatApp
{
 class Chat
 {
  private static IPAddress remoteIPAd;
  private static int remotePort;
  private static int localPort;
  [STAThread]
  static void Main(string[] args)
  {
   try
   {
    //получаем данные необходимые для соединения
    Console.WriteLine("Введите значение локального порта");
    localPort = Convert.ToInt16(Console.ReadLine());
```

```
    Console.WriteLine("Введите значение удаленного порта");
    remotePort = Convert.ToInt16(Console.ReadLine());
    Console.WriteLine("Введите значение удаленного IP-адреса");
    remoteIPAd = IPAddress.Parse(Console.ReadLine());
    //создаем слушающий поток
    Thread tRec = new Thread(new ThreadStart(Receiver));
    tRec.Start();
    while (true)
    {
     Send(Console.ReadLine());
    }
   }
   catch(Exception e)
   {
    Console.WriteLine(e.ToString());
   }
}
private static void Send(string datagram)
{
 //создаем UdpClient
 UdpClient sender = new UdpClient();
 //создаем конечную точку по информации об удаленном хосте
 IPEndPoint endPoint = new IPEndPoint(remoteIPAd,remotePort);
 try
 {
  //преобразуем данные в массив байтов
  byte[] bytes = Encoding.ASCII.GetBytes(datagram);
  //отправляем данные
  sender.Send(bytes,bytes.Length,endPoint);
 }
 catch (Exception e)
 {
  Console.WriteLine(e.ToString());
 }
 finally
 {
  sender.Close();
 }
}
public static void Receiver()
{
 //создаем UdpClient для чтения входящих данных
 UdpClient receivUdpClient = new UdpClient(localPort);
 IPEndPoint remotrIPEndPoint = null;
 try
 {
  Console.WriteLine("Готово для чата!");
  while (true)
  {
   //ожидание дейтаграммы
   byte[] receiveBytes = receivUdpClient.Receive(ref remoteIPEndPoint);
   //преобразуем и отображаем данные
   string returnData = Encoding.ASCII.GetString(receiveBytes);
```

```
    Console.WriteLine("-"+returnData.ToString());
   }
  }
  catch (Exception e)
  {
   Console.WriteLine(e.ToString());
  }
 }
}
}
```

Запустите два экземпляра приложения, для тестирования на локальной машине в качестве номера локального порта выберите 5001, а для удаленного порта 5002, *IP*-адрес 127.0.0.1, во втором приложении локальный порт – 5002, удаленный – 5001, *IP*-адрес 127.0.0.1.

Приложение передачи файлов

Создадим приложение для передачи файлов и сериализованного объекта. Программы отправителя и получателя разделены на две логические части. В первой части отправитель посылает получателю информацию о файле как сериализованный объект, а во второй части отправляет сам файл.

В получателе в первой части принимает сериализованный объект с соответсвующей информацией, а вторая часть создает файл на машине получателя. Сохраненный файл будем открывать соответствующей программой (например, *. *doc* в Word, а *.*htm* – браузером *Internet Explorer*).

Файловый сервер

Файловый сервер – это простое консольное приложение, реализованное в классе *FileSender*. В этом классе есть вложенный класс *FileDetails*, содержащий информацию о файле – тип и размер файла. Начнем с импорта необходимых пространств имен и объявления полей класса. В классе есть память закрытых полей: экземпляр класса *FileDetails*, объект *UdpClient*, а также информация о соединении с удаленным клиентом и объект *FileStream* для считывания файла, который отправляется клиенту.

Чтобы послать через сеть объект *FileDetails*, его требуется сериализовать, поэтому добавляем атрибут [*Serializable*].

Приглашаем пользователя ввести удаленный *IP*-адрес, по которому нужно отправить файл, путь и имя отправляемого файла. Открываем этот файл в объекте *FileStream* и определяем его длину. Если она больше максимально допустимой длины, равной 8192 байтам, закрываем *UdpClient* и *FileStream* и выходим из приложения. Иначе отправляем информацию о файле, выжидаем две секунды, вызвав метод *Thread.Sleep(),* и отправляем сам файл.

Метод *SendFileInfo()* заполняет поля объекта *FileDetails*, а затем сериализует объект в *MemoriStream*, используя объект *XmlSerializer*. Этот объект считывается в массив байтов и передается методу *Send()* класса *UdpClient*, который отправляет информацию о файле клиенту.

Метод *SendFile()* просто считывает содержимое файла из *FileStream* в массив байтов и отправляет его клиенту.

```
using System.IO;
using System.Threading;
using System.Xml.Serialization;
using System.Net;
using System.Net.Sockets;
using System.Diagnostics;

public class FileSender
{
 private static FileDetails fileDet = new FileDetails();
 //поля свзязанные с UdpClient
 private static IPAddress remoteIPAddress;
 private const int remotePort = 5002;
 private static UdpClient sender = new UdpClient();
 private static IPEndPoint endPoint;
 //объект FileStream
 private static FileStream fs;
 //информация о файле (требуется для получателя)
 [Serializable]
 public class FileDetails
 {
  public string FILETYPE = "";
  public long FILESIZE = 0;
 }
 [STAThread]
 static void Main(string[] args)
 {
  try
  {
   //получаем удаленный IP-адрес и создаем IPEndPoint
   Console.WriteLine("Enter Remote IP address");
   remoteIPAddress = IPAddress.Parse(Console.ReadLine().ToString());
```

```
  endPoint = new IPEndPoint(remoteIPAddress,remotePort);
  //получаем путь файла (ВАЖНО: размер файла должен быть меньше 8К)
  Console.WriteLine("Введите путь и имя файла для отправки");
  fs=new FileStream(@Console.ReadLine().ToString(),FileMode.Open,FileAccess.Read);
  if (fs.Length > 8192)
  {
   Console.Write("Эта версия передает файлы размером < 8192 байт");
   Console.ReadKey();
   sender.Close();
   fs.Close();
   return;
  }
  SendFileInfo();  //отправляем информацию о файле получателю
  Thread.Sleep(2000);  //ждем 2 секунды
  SendFile();  //отправляем сам файл
 }
 catch(Exception e)
 {
  Console.WriteLine(e.ToString());
 }
}
public static void SendFileInfo()
{
 //получаем тип и расширение файла
 fileDet.FILETYPE = fs.Name.Substring((int)fs.Name.Length-3,3);
 //получаем длину файла
 fileDet.FILESIZE = fs.Length;
 XmlSerializer fileSerializer = new XmlSerializer(typeof(FileDetails));
 MemoryStream stream = new MemoryStream();
 //сериализуем объект
 fileSerializer.Serialize(stream,fileDet);
 //считываем поток в байты
 Stream.Position = 0;
 byte[] bytes = new byte[stream.Length];
 stream.Read(bytes,0,Convert.ToInt32(stream.Length));
 Console.WriteLine("Отправка деталей файла...");
 //отправляем информацию о файле
 sender.Send(bytes,bytes.Length,endPoint);
 stream.Close();
}
private static void SendFile()
{
 //создаем файловый поток
 byte[] bytes = new byte[fs.Length];
 //поток переводим в байты
 fs.Read(bytes,0,bytes.Length);
 Console.WriteLine("Отправка файла размером = "+fs.Length+" байт");
 try
 {
  sender.Send(bytes,bytes.Length,endPoint);  //посылаем файл
 }
 catch (Exception e)
 {
```

```
   Console.WriteLine(e.ToString());
  }
  finally
  {
   fs.Close();
   sender.Close()'
  }
  Console.Read();
  Console.WriteLine("Файл успешно отправлен.");
 }
 }
}
```

Приемник файла

Приемник файла – тоже консольное приложение, реализованное в классе *FileRecs*. Здесь так же начинаем с экспорта необходимых пространств имен и объявление полей класса. Потребуется десериализовать информацию о файле, отправленную сервером, в объект *FileDetails*, поэтому нужно определить этот класс и в приложение – клиенте.

Метод *Main()* этого приложение только называет два метода, чтобы получить, соответственно, информацию о файле и сам файл.

Метод *ReceiveFile()* получает файл от сервера и сохраняет его на диске под именем *temp*, добавляя расширение, извлеченное из объекта *FileDetails*. Затем вызываем статический метод *Process.Start()* и открываем документ связанной с расширением программой.

```
using System.Xml.Serialization;
using System.Net;
using System.Net.Sockets;
using System.Diagnostics;
using System.IO;

namespace FileRecs
{
 class Program
 {
  private static FileDetails filedet;
  private static int localPort = 5002;
  private static UdpClient receivingUdpClient = new UdpClient(localPort);
  private static IPEndPoint RemoteIpEndPoint = null;
  private static FileStream fs;
  private static byte[] receiveByte = new byte[0];
  [Serializable]
  public class FileDetails
  {
   public string FILETYPE = "";
```

```
 public long FILESIZE = 0;
}
[STAThread]
static void Main(string[] args)
{
 //получаем информацию о файле
 GetFileDetails();
 //получаем файл
 ReceiveFile();
}
private stativ void GetFileDetails()
{
 try
 {
  Console.WriteLine("--**Ожидание получения деталей файла!**--");
  //получаем информацию о файле
  receiveByte = receivingUdpClient.Receive(ref RemoteIpEndPoint);
  Console.WriteLine("--Детали файла получены!--");
  XmlSerializer fileSerializer = new XmlSerializer(typeof(FileDetails));
  MemoryStream stream1 = new MemoryStream();
  //полученные данные – в поток
  stream1.Write(receiveByte,0,receiveByte.Length);
  stream1.Position = 0;
  //обязательно вызываем метод Deserialize и приводим к типу объекта
  fileDet = (FileDetails)fileSerializer.Deserialize(stream1);
  Console,WriteLine("Получен файл с расширением "+fileDet.FILETYPE+
                    " размером "+fileDet.FILESIZE.ToString()+" байт");
 }
 catch (Exception e)
 {
  Console.WriteLine(e.ToString());
 }
}
public static void ReceiveFile()
{
 try
 {
  Console.WriteLine("---***Ожидание получения файла!***---");
  //получаем файл
  receiveByte = receivingUdpClient.Receive(ref RemoteIpEndPoint);
  //преобразуем и отображаем данные
  Console.WriteLine("Сохранение полученного файла...");
  //создаемфайл temp с полученным расширением
  fs=new FileStream("temp."+fileDet.FILETYPE,FileMode.Create,
                    FileAccess.ReadWrite,FileShare.ReadWrite);
  fs.Write(receiveByte,0,receiveByte.Length);
  Console.WriteLine("Файл сохранен");
  Console.WriteLine("открытие файла с помощью соответствующей программы");
  Process.Start(fs.Name);
  //окрываем файл связанной с ним программой
 }
 catch (Exception e)
 {
```

```
     Console.WriteLine(e.ToString());
    }
    finally
    {
     fs.Close();
    }
   }
  }
}
```

Задания для самостоятельного выполнения

1. Создать приложение отправки и получения файла в *Windows*-приложении с подтверждением от сервера о получении отправляемых ему данных.

2. Серверное приложение перекодирует в теги и возвращает ответ в браузере клиента.

3. Выполнить индивидуальное задание, полученное у преподавателя.

Варианты индивидуальных заданий к лабораторной работе.

ВАРИАНТ 1. В приложении клиент выбирает из поля со списком значение Времени года, в зависимости от выбранного значения, рассылается на несколько компьютеров картинка с изображением времени года, если выбрана зима – отправляется генерируемое сообщение с поздравлением или текстом песенки.

ВАРИАНТ 2. В приложении клиента изображен светофор, при нажатии на кнопку Пуск, Светофор циклично переключается с красного через 4 секунды на желтый через 3 секунды на зеленый в зависимости от света светофора, рассылается на несколько компьютеров сообщение «Стой», «Приготовиться», «Иди».

ВАРИАНТ 3. В приложении генерируется три графических объекта, в зависимости от сгенерированного объекта рассылается на несколько компьютеров файл с определенным расширением *txt*, *html*, *jpg*, который открывается соответствующей программой на стороне получателя.

ВАРИАНТ 4. В приложении динамическое передвижение объекта от левого края экрана в правый край, объект достигнет правого края экрана, рассылается на несколько компьютеров один из трех текстовых файлов, порядок отправки определяется случайным образом.

ВАРИАНТ 5. В приложении на экране по дуге перемещается объект – солнце, когда объект находится в левой части экрана компьютерам рассылается картинка – «Рассвет», когда в верхней части экрана – отправляется сообщение «Полдень», когда достигнет правого края экрана, рассылается на несколько компьютеров рассылается картинка – «Закат».

ВАРИАНТ 6. В приложении клиент выбирает пункт из четырех радио кнопок, в зависимости от выбранного значения, если выбран первый или третий пункт рассылается картинка с изображением, если выбраны второй пункт отправляется сообщение, если четвертый отправляется *html*-файл, который открывается при получении в браузере.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Особенности установления соединения при организации соединения по протоколу *UDP*.

2. Методы, используемые в при организации соединения по протоколу *UDP*.

3. Методы, используемые в при организации передачи данных по протоколу *UDP*.

4. Методы, используемые в при организации передачи файлов по протоколу *UDP*.

Лабораторная работа 3

Тема: «Использование сокетов групповой рассылки»

Создадим приложение групповой рассылки и интерактивного форума, к которому могут обратиться несколько пользователей, чтобы отправить сообщение всем остальным клиентам. В этом приложении каждая станция действует и как клиент, и как сервер. Каждый пользователь может ввести сообщение, отправляемое всем участникам форума.

Создаем следующую *Windows*-форму (рисунок 1):

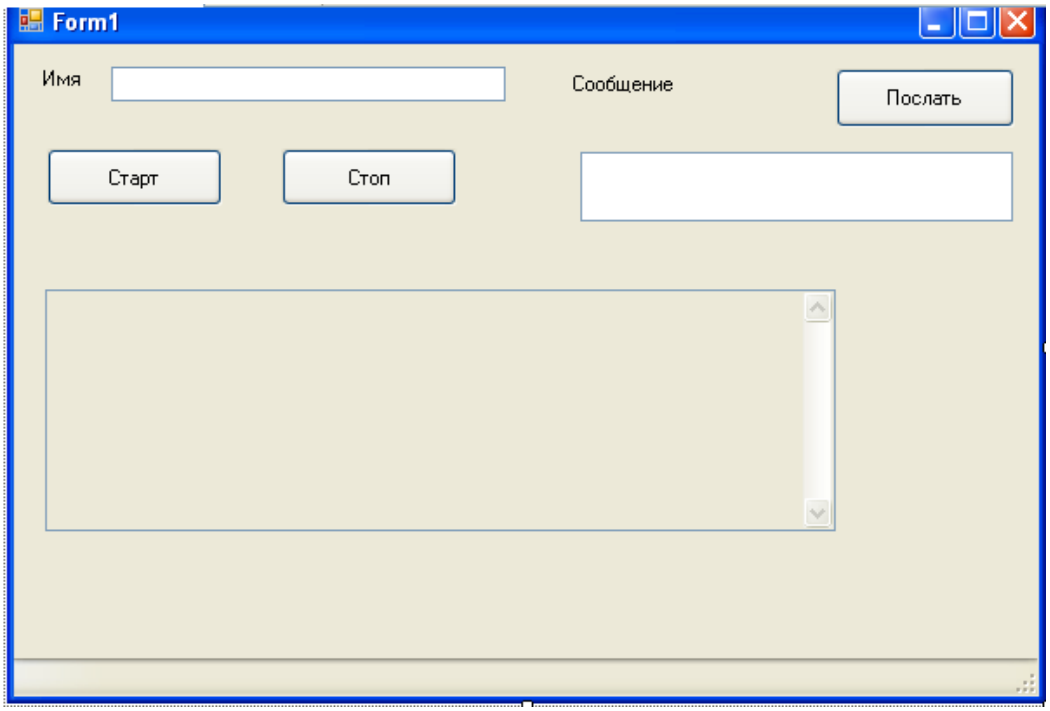

Рисунок 1 – Форма приложения групповой рассылки

В таблице 1 показаны основные элементы управления формы с их именами и значениями свойств по умолчанию:

Таблица 1 – Элементы управления формы

Тип элемента	**Имя**	**Свойства**
Текстовое поле	*textName*	
кнопка	*buttonStart*	
кнопка	*buttonStop*	*Enabled = "false"*
кнопка	*buttonSend*	*Enabled = "false"*
Текстовое поле	*textMessage*	*Multiline = true*
Текстовое поле	*textMessages*	*Multiline = true* *ReadOnly = true* *Scrollbars = vertical*
Полоса состояния	*statusBar*	

```
using System.Collections.Specialized;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Configuration;

namespace MulticastChat
{
 public partial class Form1 : Form
 {
  private bool done=true;//флаг остановки следующего потока
  private UdpClient client;
  private IPAddress groupAddress;//групповой адрес рассылки
  private int localPort;//локальный порт для приема сообщений
  private int remotePort;//удаленный порт для отправки сообщений
  private int ttl;
  private IPEndPoint remoteEP;
  private UnicodeEncoding encod = new UnicodeEncoding();
  private string name;//имя пользователя в разговоре
  private string message;//сообщение для отправки
```

Групповые адреса и номера портов должны быть легко конфигурируемыми, поэтому создадим *XML*-файл конфигурирования приложения с именем *MulticastChat.exe.config* и следующим содержанием.

```
<?xml version="1/0" encoding="utf-8" ?>
<configuration>
 <appSettings>
  <add key="GroupAddress" value="234.5.6.11"/>
  <add key="LocalPort" value="7777"/>
  <add key="RemotePort" value="7777"/>
  <add key="TTL" value="1"/>
 </appSettings>
</configuration>
```

Этот конфигурационный файл нужно поместить в тот же каталог, где находится исполнимый файл (*Debug*).

```
public Form1()
{
 InitializeComponent();
 try
 {
  NameValueCollection configuration = ConfigurationSettings.AppSettings;
  groupAddress=IPAddress.Parse(configuration["GroupAddress"]);
  localPort=int.Parse(configuration["LocalPort");
  remotePort=int.Parse(configuration["RemotePort"]);
  ttl=int.Parse(configuration["TTL"]);
 }
 catch
 {
```

```
  MessageBox.Show(this,"ошибки в конфигурационном файле");
  buttonStart.Enabled=false;
 }
}
```

Для присоединения к группе, получающей рассылку, в обработчике щелчка по кнопке Старт считываем имя.

```
private void buttonStart_Click(object sender, EventArgs e)
{
 Name=textName.Text;
 textName.ReadOnly=true;
 try
 {
  client=new UdpClient(localPort);
  client.JoinMulticastGroup(groupAddress, ttl);
  remoteEP=new IPEndPoint(groupAddress, remotePort);
  Thread receiver=new Thread(new ThreadStart(Listener));
  receiver.IsBackground=true;
  receiver.Start();
  byte[] data=encod.GetBytes(name+" в сети");
  client.Send(data,data.Length,remoteEP);
  buttonStart.Enabled=false;
  buttonStop.Enabled=true;
  buttonSend.Enabled=true;
 }
 catch (SocketException ex)
 {
  MessageBox.Show(this, "Error");
 }
}
```

Получение сообщений, адресованных группе

В методе слушающего потока, который был создан раньше, ждём в методе *client.Receive()*, пока не поступит сообщение. С помощью класса *UnicodeEncoding* полученный массив байтов преобразуется в строку.

Теперь возвращаемое сообщение нужно отобразить в пользовательском интерфейсе. При пользовании потоков элементов управления *Windows* следует обращать внимание на один важный момент.

В традиционной среде программирования *Windows* элементы управления *Windows* можно создавать из разных потоков, но лишь поток, создавший элемент управления, может вызывать в нём методы, поэтому все функции обработки элемента управления *Windows* должны вызываться в создавшем его потоке.

Все методы управления *Windows*-форм должны вызываться на создающем потоке, за исключением метода *Invoke()* и его асинхронных версий *BeginInvoke()* и *EndInvoke()*. Эти методы можно вызывать из любого потока, поскольку они переадресуют вызываемый метод потоку, создавшему элемент управления *Windows*, а уже тот поток вызывает метод.

Поэтому вместо того, чтобы непосредственно отобразить сообщение в текстовом поле, вызываем метод *Invoke()* класса *From*. Поскольку этот же самый поток создал текстовое поле, он удовлетворяет всем требованиям.

Метод *Invoke()* требует параметра типа *Delegate*, и поскольку любой делегат порождён этим классом, то метод может быть передан в любой делегат.

В среде *.NET. Framework* уже определён делегат для вызова метода без параметров: *System.Windows.Forms.MehtodInvoker*. Этот делегат принимает такие методы без параметров, как метод *DisplayRecieviedMassege()*.

```
private void Listener()
{
 done=false;
 try
 {
  while (!done)
  {
   IPEndPoint ep=null;
   byte[] buffer = client.Receive(ref ep);
   message = encod.GetString(buffer);
   this.Invoke(new MethodInvoker(DisplayRecieviedMessage));
  }
 }
 catch (Exception ex)
 {
  MessageBox,Show(this, ex.Message,"Error");
 }
}
private void DisplayDecieviedMessage()
{
 string time = DateTime.Now.ToString("t");
 textMessages.Text = time + " " + message + textMessage.Text;
 statusBar.Text = "Получено последнее сообщение в " + time;
}
private void buttonSend_Click(object sender, EventArgs e)
{
 try
 {
  //отправляем сообщение группу
  byte[] data = encod.GetBytes(name + ":" + textMessage.Text);
```

```
  client.Send(data,data.Length,remoteEP);
  textMessage,Clear();
  textMessage.Focus();
 }
 catch (Exception ex)
 {
  MessageBox.Show(this, ex.Message);
 }
}
private void buttonStop_Click(object sender, EventArgs e)
{
 StopListener();
}
private void StopListener()
{
 //отправляем группе сообщение о выходе
 byte[] data = encod.GetBytes(name + " не в сети");
 client.Send(data, data.Length, remoteEP);
 //покидаем группу
 client.DropMulticastGroup(groupAddress);
 client.Close();
 //останавливаем поток получающий сообщения
 done = true;
 buttonStart.Enabled=true;
 buttonStop.Enabled=false;
 buttonSend.Enabled=false;
}
private void OnClosing(object sender, System.XomponentModel.CancelEventArgs e)
{
 if (!done)
  StopListener();
}
}
```

Задания для самостоятельного выполнения

1. Запустите приложение на нескольких станциях

2. Выполнить индивидуальное задание, полученное у преподавателя.

Варианты индивидуальных заданий к лабораторной работе.

ВАРИАНТ 1. Есть две группы компьютеров. Группа 1 получает файлы с расширением *HTML* и открывает их в браузере. Группа 2 получает файлы изображений и открывает их. На компьютере – источнике одновременно отправляются два файла: один с расширением *HTML*, а другой с изображением, например, с расширением JPEG. Компьютеры, которые принадлежат группе 1, открывают полученный файл в браузере, другие открывают файл изображение.

Необходимо разработать клиент/серверное приложение, в котором сервер каждые 10 секунд распространяет некоторое текстовое сообщение, например, о погоде, всем промежуточным клиентам, зарегистрированным в группе 233.0.0.1, порт 1502 с помощью *UDP*.

ВАРИАНТ 2. Есть две группы компьютеров. Есть файл, разделенный на несколько разделов, например, записка по курсовой работе. Компьютеры, которые относятся к группе 1, получают только Раздел 1 из файла источника. Группа 2 получает только Раздел 2 из файла источника. На компьютере – источнике только один файл.

Необходимо разработать клиент/серверное приложение, в котором на сервер выбирается интервал периодичности, затем через выбранное значение, например 2 секунды, распространяется введенное текстовое сообщение всем промежуточным клиентам, зарегистрированным в группе 234.0.0.1, порт 1503 с помощью *UDP*.

ВАРИАНТ 3. Есть две группы компьютеров. На стороне компьютера – источника, пользователь из списка выбирает два файла. Компьютеры, которые принадлежат группе 1, открывают файл 1, а принадлежащие к группе 2 – открывают файл 2.

Необходимо разработать клиент/серверное приложение, в котором сервер каждую 2-ю, 6-ю и 8-ю секунды отправляет сообщение 1, а каждую 4-ю, 7-ю и 10 секунду распространяет текстовое сообщение 2, всем промежуточным клиентам, зарегистрированным в группе 235.0.0.1, порт 1504 с помощью *UDP*.

ВАРИАНТ 4. Есть две группы компьютеров Группа «Пешеходы» и «Автомобилисты». На стороне компьютера – источника зажигается светофор: красный свет – компьютерам «Автомобилистам», рассылается сообщение «Стоп», а компьютерам «Пешеходам» – сообщение «Идите», когда на сервере загорается желтый свет – всем рассылается сообщение «Внимание!», когда на сервере загорается зеленый свет – компьютерам «Автомобилистам», рассылается сообщение «Проезжайте!», а компьютерам «Пешеходам» – сообщение «Стойте!».

Необходимо разработать клиент/серверное приложение, в котором сервер каждую 5-ю секунду распространяет текстовое сообщение, которое введено на стороне компьютера-источника, всем промежуточным клиентам, зарегистрированным в группе 236.0.0.1, порт 1505 с помощью *UDP*.

ВАРИАНТ 5. Есть две группы компьютеров. Есть файл с числами в столбик от 1 до 20. Компьютеры, которые относятся к группе 1, получают только нечетные цифры из файла. Группа 2 получает только четные числа из файла источника. На компьютере – источнике только один файл.

Необходимо разработать клиент/серверное приложение, в котором на сервер выбирается интервал периодичности, затем через выбранное значение, например 2 секунды, распространяется текстовый файл всем промежуточным клиентам, зарегистрированным в группе 237.0.0.1, порт 1506 с помощью *UDP*.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Особенности установления соединения при организации соединения при реализации групповой рассылки.
2. Роль конфигурационного файла при организации соединения при групповой рассылки.
3. Структура приложения групповой рассылки.
4. Методы, используемые в при подключении к группе и отключении.

Лабораторная работа 4

Тема: «Реализация сетевых приложений средствами *Java*»

Сетевое программирование с Сокетами и Каналами

Одна из сильных сторон *Java* заключается в безпроблемной сетевой работе. Дизайнеры сетевой библиотеки *Java* сделали ее достаточно простой для чтения и записи файлов, за исключением случая, когда «файл» существует на удаленной машине и удаленная машина может решать что ей делать с информацией, которую запрашивают или посылают. Детали сетевого взаимодействия были абстрагированы и о них заботится ядро *JVM* и локальный пакет установки *Java*.

Программная модель, которую используют для такого файла, фактически, это обертка сетевого соединения ("сокет") с объектом потока, так что, в конечном счете, используют те же вызовы методов, которые используют для других потоков. Кроме того, встроенная многопоточность *Java* исключительно удобна, когда имеют дело с такой сетевой возможностью, как обработка множества соединений одновременно.

Идентификация машины

Для того чтобы передать данные с одной машины на другую необходимо убедиться, что выполнено подсоединение к определенной машине в сети. Ранние варианты сетей были удовлетворены предоставлением уникальных имен машинам внутри локальной сети. Однако *Java* работает в пределах *Internet*, что требует способа для уникальной идентификации машины из любой точки всего мира. Это выполняется с помощью *IP* (*Internet Protocol*) адреса, который может существовать в двух формах:

– привычная форма *DNS* (*Domain Name System*).;

– альтернативный вариант: можно использовать форму из четырех чисел, разделенных точками, например 123.255.28.120.

В обоих случаях *IP*-адрес представляется как 32-х битное число (так как каждое из четырех чисел не может превышать 255), и можно получить специальный *Java* объект для представления этого числа из любой из перечисленных выше форм, используя статический метод

InetAddress.getByName(), который определен в *java.net*. Результатом будет объект типа *InetAddress*, который можно использовать для создания "сокета".

В качестве простейшего примера использования *InetAddress.getByName()* рассмотрим, что произойдет при использовании коммутируемого доступа (*dial-up Internet service provider* (*ISP*)). При каждом дозвоне назначается временный *IP*-адрес. Но пока есть соединение, *IP*-адрес имеет такую, же силу, как и другие *IP*-адреса в *Internet*. Если кто-либо соединится с определенной машиной используя *IP*-адрес, то он может соединится с *Web*-сервером или *FTP*-сервером, который запущен на машине. Конечно, ему необходимо знать *IP*-адрес, так как при каждом дозвоне назначается новый адрес.

Приведенная ниже программа использует *InetAddress.getByName()* для воспроизведения *IP*-адреса. Для ее использования необходимо знать имя компьютера. Под управлением *Windows* перейдите в "*Settings*", "*Control Panel*", "*Network*" и выберите закладку "*Identification*". Содержимое в поле "*Computer name*" является той строкой, которую необходимо поместить в командную строку.

```
//: c15:WhoAmI.java
//Нахождение вашего сетевого адреса, когда вы соединены с Internet
//{Запускается руками} Должно быть установлено соединение с Internet
//{Args: www.google.com}

import java.net.*;

public class WhoAmI
{
 public static void main(String[] args) throws Exception
 {
  if (args.length != 1)
   {
    System.err.println("Usage: WhoAmI MachineName");
    System.exit(1);
   }
  InetAddress a = InetAddress.getByName(args[0]);
  System.out.println(a);
 }
}
```

Например, машина называется "*peppy*". Так что, когда соединяемся с провайдером, запускаем программу: `java WhoAmI peppy`.

Получим назад сообщение такого типа (конечно же, адрес отличается при каждом новом соединении): `peppy/199.190.87.75`

Если скажем этот адрес другу, и у нас будет запущен *Web*-сервер на нашем компьютере, он сможет соединиться с сервером, перейдя по ссылке *http://199.190.87.75* (только до тех пор, пока мы остаемся соединенными во время одной сессии).

Серверы и клиенты

Основное назначение сети состоит в том, чтобы позволить двум машинам соединиться и пообщаться друг с другом. Так как две машины могут найти друг друга, они могут провести милую, двустороннюю беседу.

Машина, которая стоит в одном месте, называется сервером, а машина, которая ищет, называется клиентом. Это различие важно лишь до тех пор, пока клиент пробует соединиться с сервером. Как только они соединятся, они становятся двумя сторонами коммуникационного процесса и более не имеет значения, какая машина принимала роль сервера, а какая принимала роль клиента.

Таким образом, работа сервера состоит в прослушивании соединения, она выполняется с помощью специального объекта, который вы создаете. Работа клиента состоит в попытке создать соединение с сервером, и это выполняется с помощью специального клиентского объекта, который вы создаете. Как только соединение установлено, вы увидите, что и клиентская, и серверная сторона соединения магическим образом превращается потоковый объект ввода/вывода, таким образом, вы можете трактовать соединение, как будто вы читаете и пишете файл. Таким образом, после установки соединения, вы просто используете хорошо знакомые команды ввода/вывода. Это одна из прекраснейших особенностей работы по сети в *Java*.

Тестирование программ без сети

По многим причинам, можно не иметь клиентской машины, серверной машины и сети, доступных для тестирования программ. Можно выполнять упражнения в обстановке классной комнаты, или, возможно, вы пишите программы, которые еще не достаточно стабильны и не могут быть выложены в сеть. Создатели *Internet Protocol* учли эту возможность и создали специальный адрес, называемый *localhost, IP*-адрес "локальной заглушки (*local loopback*)" для

тестирования без использования сети. Общий способ для получения такого адреса в *Java* такой:

```
InetAddress addr = InetAddress.getByName(null);
```

Если передадите в *getByName()* значение *null*, метод по умолчанию будет использовать *localhost. InetAddress* является тем, что вы используете для указания определенной машины. Нельзя манипулировать содержимым *InetAddress* (но можно напечатать его). Единственный способ, которым можно создать *InetArddress*, это через один из перегруженных статических методов класса *getByName()*, *getAllByName()*, или *getLocalHost()*.

Также можно получить адрес локальной заглушки, передав строку *localhost*:

```
InetAddress.getByName("localhost");
```

(предполагается, что "*localhost*" сконфигурирован в таблице "*hosts*" на машине), или используя цифровую четырехзначную форму для имени, представляющем заглушку: `InetAddress.getByName("127.0.0.1");`

Все три формы производят одинаковый результат.

Порт: уникальное место внутри машины

IP-адреса не достаточно для уникальной идентификации сервера, так как многие сервера могут существовать на одной машине. Каждая *IP*-машина также содержит порты, и когда устанавливаете клиента или сервер, необходимо выбрать порт, через который и клиент, и сервер согласны соединиться.

Порт – это не физическое расположение в машине, а программная абстракция (в основном для целей учета). Клиентская программа знает, как соединится к машине через ее *IP*-адрес, но как она может присоединиться к определенной службе (потенциально, к одной из многих на этой машине)? Таким образом, номер порта стал вторым уровнем адресации. Идея состоит в том, что при запросе определенного порта вы запрашиваете службу, ассоциированную с этим номером порта.

Служба времени – простейший пример службы. Обычно каждая служба ассоциируется с уникальным номером порта на определенной серверной машине.

Клиент должен предварительно знать, на каком порту запущена нужная ему служба.

Системные службы зарезервировали использование портов с номерами от 1 до 1024, так что невозможно использовать этот или любой другой порт, про который известно, что он задействован. Первым выбором, например, будет порт 8080.

Сокеты

Сокет – это программная абстракция, используемая для представления "терминалов" соединения между двумя машинами. Для данного соединения есть сокет на каждой машине, и можно представить гипотетический "кабель", включенный в сокет. Конечно, физическое оборудование и каблирование между машинами полностью неизвестно. Главное назначение абстракции состоит в том, что не надо знать более того, что необходимо.

В *Java* сокет создается, чтобы создать соединение с другой машиной, затем чтобы получить *InputStream* и *OutputStream* (или, с соответствующими конверторами, *Reader* и *Writer*) из сокета, чтобы можно было трактовать соединение, как объект потока ввода/вывода.

Существует два класса сокетов, основанных на потоках: *ServerSocket*, который использует сервер для "прослушивания" входящих соединения, и *Socket*, который использует клиент для инициализации соединения. Как только клиент создаст сокетное соединение, *ServerSocket* возвратит (посредством метода *accept()*) соответствующий *Socket*, через который может происходить коммуникация на стороне сервера. После этого можно общаться в соединении через *Socket* с *Socket* и оба конца трактуются одинаково, поскольку они и являются одним и тем же. На этой стадии используют методы *getInputStream()* и *getOutputStream()* для получения соответствующих объектов *InputStream* и *OutputStream* для каждого сокета. Они должны быть обернуты внутрь буферных и форматирующих классов точно так же, как и другие объекты потоков.

Использование термина *ServerSocket* может показаться другим примером сбивающей с толку схемы именования в библиотеках *Java*. Можно подумать, что

ServerSocket лучше было бы назвать "*ServerConnector*" или как-то по другому, без слова "*Socket*" внутри. Также можно подумать, что *ServerSocket* и *Socket* должны оба наследоваться от какого-то общего базового класса. На самом деле, два класса имеют некоторые общие методы, но не настолько, чтобы дать им общий базовый класс. Вместо этого, работа *ServerSocket* состоит в том, чтобы ждать, пока некоторая машина не присоединится к нему, а затем он возвращает реальный *Socket*. Вот почему, кажется, что *ServerSocket* назван немножко неправильно, так как его работа состоит не в том, чтобы быть реальным сокетом, а в том, чтобы создавать объект *Socket*, когда кто-то присоединяется к нему.

Однако *ServerSocket* создает физический "сервер" или слушающий сокет на хост-машине. Этот сокет слушает входящие соединения, а затем возвращает "связанный" сокет (с определенными локальной и удаленной конечными точками) посредством метода *accept()*. Сбивающая часть состоит в том, что оба эти сокета (слушающий и связанный) ассоциированы с одним и тем же серверным сокетом. Слушающий сокет может принять только новый запрос на соединение, а не пакет данных. Так что, не смотря на то, что *ServerSocket* имеет мало смыла, с точки зрения программирования, в нем много смысла "физически".

Когда создаете *ServerSocket*, ему даете только номер порта и не даете ему *IP*-адрес, поскольку он уже есть на той машине, на которой он представлен. Однако когда создаете *Socket*, необходимо передать ему и *IP*-адрес, и номер порта, к которому хотите присоединиться. (Однако, *Socket*, который возвращается из метода *ServerSocket.accept()*, уже содержит всю эту информацию.)

Простейший сервер и клиент

Этот пример покажет простейшее использование серверного и клиентского сокета. Все, что делает сервер, это ожидает соединения, затем использует сокет, полученный при соединении, для создания *InputStream* и *OutputStream*. Они конвертируются в *Reader* и *Writer*, которые оборачиваются в *BufferedReader* и *PrintWriter*. После этого все, что будет прочитано из *BufferedReader*, будет переправлено в *PrintWriter*, пока не будет получена строка "*END*", означающая, что пришло время закрыть соединение.

Клиент создает соединение с сервером, затем создает *OutputStream* и создает некоторую обертку, как и в сервере. Строки текста посылаются через полученный *PrintWriter*. Клиент также создает *InputStream* (с соответствующей конвертацией и оберткой), чтобы слушать, что говорит сервер (который, в данном случае, просто отсылает слова назад).

И сервер, и клиент используют одинаковый номер порта, а клиент использует адрес локальной заглушки для соединения с сервером на этой же самой машине, так что нельзя провести тест по сети. (Для некоторых конфигураций может понадобиться сетевое соединение для работы программы даже, если не используете сетевую коммуникацию.)

Вот сервер:

```
//: c15:JabberServer.java
//Очень простой сервер, который просто отсылает
//назад все, что посылает клиент.
//{RunByHand}
import java.io.*;
import java.net.*;
public class JabberServer
{
 //Выбираем порт вне пределов 1-1024:
   public static final int PORT = 8080;
   public static void main(String[] args) throws IOException
   {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Started: " + s);
    try
    {
     //Блокирует до тех пор, пока не возникнет соединение:
       Socket socket = s.accept();
       try
       {
        System.out.println("Connection accepted: " + socket);
        BufferedReader in = new BufferedReader(new InputStreamReader
                                          (socket.getInputStream()));
        //Вывод автоматически выталкивается из буфера PrintWriter
        PrintWriter out = new PrintWriter(new BufferedWriter
          (new OutputStreamWriter(socket.getOutputStream())), true);
        while (true)
        {
         String str = in.readLine();
         if (str.equals("END"))
           break;
         System.out.println("Echoing: " + str);
         out.println(str);
        }
```

```
         //Всегда закрываем два сокета...
        }
        finally
        {
         System.out.println("closing...");
         socket.close();
        }
     }
     finally
     {
      s.close();
     }
    }
}
```

Для *ServerSocket* необходим только номер порта, а не *IP*-адрес (так как он запускается на локальной машине!). Когда вызываете *accept()*, метод блокирует выполнение до тех пор, пока клиент не попробует подсоединиться к серверу. То есть, сервер ожидает соединения, но другой процесс может выполняться. Когда соединение установлено, метод *accept()* возвращает объект *Socket*, представляющий это соединение.

Здесь тщательно обработана ответственность за очистку сокета. Если конструктор *ServerSocket* завершится неудачей, программа просто завершится (обратите внимание, что мы должны предположить, что конструктор *ServerSocket* не оставляет ни каких открытых сокетов, если он завершается неудачей). По этой причине *main()* выбрасывает *IOException*, так что в блоке *try* нет необходимости. Если конструктор *ServerSocket* завершится успешно, то все вызовы методов должны быть помещены в блок *try-finally*, чтобы убедиться, что блок не будет покинут ни при каких условиях и *ServerSocket* будет правильно закрыт.

Аналогичная логика используется для сокета, возвращаемого из метода *accept()*. Если метод *accept()* завершится неудачей, то надо предположить, что сокет не существует и не удерживает никаких ресурсов, так что он не нуждается в очистке. Однако если он закончится успешно, то следующие выражения должны быть помещены в блок *try-finally*, чтобы при каких-либо ошибках все равно произошла очистка. Позаботиться об этом необходимо, потому что сокеты используют важные ресурсы, не относящиеся к памяти, так что надо очищать их (так как в *Java* нет деструкторов, чтобы сделать это за вас).

И *ServerSocket* и *Socket*, производимый методом *accept()*, печатаются в *System.out.* Это означает, что автоматически вызывается их метод *ToString()*. Вот что он выдаст:

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

Следующая часть программы выглядит, как открытие файла для чтения и записи за исключением того, что *InputStream* и *OutputStream* создаются из объекта *Socket*. И объект *InputStream* и *OutputStream* конвертируются в объекты *Reader* и *Writer* с помощью "классов-конвертеров" *InputStreamReader* и *OutputStreamreader*, соответственно. Можно также использовать классы из *Java 1.0 InputStream* и *OutputStream* напрямую, но, с точки зрения вывода, есть явное преимущество в использовании этого подхода. Оно проявляется в *PrintWriter*, который имеет перегруженный конструктор, принимающий в качестве второго аргумента флаг типа *boolean*, указывающий, нужно ли автоматическое выталкивание буфера вывода в конце каждого выражения *println()* (но не *print()*). Каждый раз, когда записываете в вывод, буфер вывода должен выталкиваться, чтобы информация проходила по сети. Выталкивание важно для этого конкретного примера, поскольку клиент и сервер ожидают строку от другой стороны, прежде чем приступят к ее обработке. Если выталкивание буфера не произойдет, информация не будет помещена в сеть до тех пор, пока буфер не заполнится, что может привести к многочисленным проблемам в этом примере.

Когда пишите сетевую программу, необходимо быть осторожным при использовании автоматического выталкивания буфера. При каждом выталкивании буфера пакеты должны создаваться и отправляться. В данном случае это именно то, что надо, так как если пакет, содержащий строку, не будет отослан, то общение между сервером и клиентом остановится. Другими словами, конец строки является концом сообщения. Но во многих случаях, сообщения не ограничиваются строками, так что будет более эффективным использовать автоматическое выталкивание буфера, поэтому позвольте встроенному механизму буферизации построить и отослать пакет. В таком случае могут быть посланы пакеты большего размера, и процесс обработки пойдет быстрее.

Обратите внимание, что фактически все открытые потоки, буферизированы.

В бесконечном цикле *while* происходит чтение строк из входного *BufferedReader* и запись информации в *System.out* и в выходной *PrintWriter*. Обратите внимание, что вход и выход могут быть любыми потоками, так случилось, что они связаны с сетью.

Когда клиент посылает строку, содержащую "*END*", программа прекращает цикл и закрывает сокет.

Вот клиент:

```
//: c15:JabberClient.java
//Очень простой клиент, который просто посылает
//строки на сервер и читает строки, посылаемые сервером
//{RunByHand}

import java.net.*;
import java.io.*;

public class JabberClient
{
 public static void main(String[] args) throws IOException
 {
 //Передаем null в getByName(), получая специальный IP-адрес
 //"локальной заглушки" для тестирования на машине без сети
   InetAddress addr = InetAddress.getByName(null);

 //Альтернативно, вы можете использовать адрес или имя
 //InetAddress addr =
 //InetAddress.getByName("127.0.0.1");
 //InetAddress addr =
 //InetAddress.getByName("localhost");
   System.out.println("addr = " + addr);
   Socket socket = new Socket(addr, JabberServer.PORT);

//Помещаем все в блок try-finally, чтобы
//быть уверенным, что сокет закроется
   try
   {
    System.out.println("socket = " + socket);
    BufferedReader in=new BufferedReader(new InputStreamReader
                                         (socket.getInputStream()));
//Вывод автоматически Output выталкивается PrintWriter
    PrintWriter out=new PrintWriter(new BufferedWriter
        (new OutputStreamWriter(socket.getOutputStream())), true);
    for (int i = 0; i < 10; i++)
    {
     out.println("howdy " + i);
     String str = in.readLine();
     System.out.println(str);
```

```
    }
     out.println("END");
   }
   finally
   {
    System.out.println("closing...");
    socket.close();
   }
 }
}
```

В *main()* можно видеть все три способа получение *InetAddress IP*-адреса локальной заглушки: с помощью *null, localhost* или путем явного указания зарезервированного адреса 127.0.0.1, если хотите соединиться с машиной по сети, замените это *IP*-адресом машины. Когда печатается *InetAddress* (с помощью автоматического вызова метода *toString()*), то получается результат:

При передачи в *getByName()* значения *null*, он по умолчанию ищет *localhost* и затем производит специальный адрес 127.0.0.1.

Обратите внимание, что *Socket* создается при указании и *InetAddress*, и номера порта. Чтобы понять, что это значит, когда будете печатать один из объектов *Socket*, помните, что Интернет соединение уникально определяется четырьмя параметрами: клиентским хостом, клиентским номером порта, серверным хостом и серверным номером порта. Когда запускается сервер, он получает назначаемый порт (8080) на *localhost* (127.0.0.1). Когда запускается клиент, он располагается на следующем доступном порту на своей машине, 1077 – в данном случае, который так же оказался на той же самой машине (127.0.0.1), что и сервер. Теперь, чтобы передать данные между клиентом и сервером, каждая сторона знает, куда посылать их. Поэтому, в процессе соединения с "известным" сервером клиент посылает "обратный адрес", чтобы сервер знал, куда посылать данные. Вот что вы видите среди выводимого стороной сервера:

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

Это означает, что сервер просто принимает соединение с адреса 127.0.0.1 и порта 1077 во время прослушивания локального порта (8080). На клиентской стороне:

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

Это значит, что клиент установил соединение с адресом 127.0.0.1 по порту 8080, используя локальный порт 1077.

Можно заметить, что при каждом повторном запуске клиента номер локального порта увеличивается. Он начинается с 1025 (первый после зарезервированного блока портов) и будет увеличиваться до тех пор, пока не перезапустите машину, в таком случае он снова начнется с 1025 (на машинах под управлением *UNIX*, как только будет достигнут верхний предел диапазона сокетов, номер будет возвращен снова к наименьшему доступному номеру).

Как только объект *Socket* будет создан, процесс перейдет к *BufferedReader* и *PrintWriter*, как это уже видели в сервере (в обоих случаях начинаете с *Socket*). В данном случае, клиент инициирует обмен путем посылки строки "*howdy*", за которой следует число. Обратите внимание, что буфер должен опять выталкиваться (что происходит автоматически из-за второго аргумента в конструкторе *PrintWriter*). Если буфер не будет выталкиваться, процесс обмена повиснет, поскольку начальное "*howdy*" никогда не будет послана (буфер недостаточно заполнен, чтобы отсылка произошла автоматически). Каждая строка, посылаемая назад сервером, записывается в *System.out*, чтобы проверить, что все работает корректно. Для завершения обмена посылается ранее оговоренный "*END*". Если клиент просто разорвет соединение, то сервер выбросит исключение.

Аналогичные меры приняты, чтобы быть уверенным в том, что сетевые ресурсы, представляемые сокетом, будут правильно очищены. Для этого используется блок *try-finally*.

Сокет производит "посвященную" связь, которая остается постоянной до тех пор, пока не будет явного разъединения. (Посвященная связь может быть рассоединена неявно, если одна из сторон, или посредническая связь соединения рушатся.) Это означает, что два партнера замкнуты в коммуникации и соединение постоянно открыто. Это выглядит, как логический подход к сети, но это вносит дополнительную нагрузку на сеть.

Обслуживание множества клиентов

JabberServer работает, но он может обработать только одного клиента одновременно. В обычных серверах захотите, чтобы была возможность иметь дело со многими клиентами одновременно. Ответом является многопоточность, и в языках, которые не поддерживают многопоточность напрямую, это означает, все возможные трудности. Многопоточность в *Java* проста насколько это возможно, учитывая это, можно сказать, что многопоточность весьма сложная тема. Поскольку нити (потоки) в *Java* достаточно прямолинейны, то создание сервера, который обрабатывает несколько клиентов, относительно простое занятие.

Основная схема состоит в создании единственного *ServerSocket* на сервере и вызове метода *accept()* для ожидания новых соединений. Когда *accept()* возвращается, получается результирующий сокет и он используется для создания новой нити (потока), работа которой будет состоять в обслуживании определенного клиента. Затем вызывают метод *accept()* снова, чтобы подождать нового клиента.

В следующем коде сервера можно увидеть, что он очень похож на пример *JabberServer.java*, за исключением того, что все операции по обслуживанию определенного клиента были помещены внутрь отдельного *thread*-класса:

```
//: c15:MultiJabberServer.java
//Сервер, который использует многопоточность
//для обработки любого числа клиентов.

//{RunByHand}
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread
{
 private Socket socket;
 private BufferedReader in;
 private PrintWriter out;
 public ServeOneJabber(Socket s) throws IOException
 {
  socket = s;
  in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
  //Включаем автоматическое выталкивание
  out = new PrintWriter(new BufferedWriter(new OutputStreamWriter
                                (socket.getOutputStream())), true);
  //Если любой из вышеприведенных вызовов приведет к
```

```
  //возникновению исключения, то вызывающий отвечает за
  //закрытие сокета. В противном случае, нить закроет его
  //вызываем run()
  start();
 }
 public void run()
 {
  try
  {
   while (true)
    {
     String str = in.readLine();
     if (str.equals("END"))
       break;
     System.out.println("Echoing: " + str);
     out.println(str);
    }
    System.out.println("closing...");
  }
  catch (IOException e)
  {
   System.err.println("IO Exception");
  }
  finally
  {
   try
   {
    socket.close();
   }
   catch (IOException e)
   {
    System.err.println("Socket not closed");
   }
  }
  }
 }
public class MultiJabberServer
{
 static final int PORT = 8080;
 public static void main(String[] args) throws IOException
 {
  ServerSocket s = new ServerSocket(PORT);
  System.out.println("Server Started");
  try
  {
   while (true)
   {
    //Блокируется до возникновения нового соединения
    Socket socket = s.accept();
    try
    {
      new ServeOneJabber(socket);
    }
```

```
    catch (IOException e)
    {
     //Если завершится неудачей, закрывается сокет,
     //в противном случае, нить закроет его
     socket.close();
    }
   }
  }
  finally
  {
   s.close();
  }
 }
}
```

Нить *ServeOneJabber* принимает объект *Socket*, который производится методом *accept()* в *main()* при каждом новом соединении с клиентом. Затем, как и прежде, с помощью *Socket*, создается *BufferedReader* и *PrintWriter* с возможностью автоматического выталкивания буфера. И наконец, вызывается специальный метод нити *start()*. Здесь выполняются те же действия, что и в предыдущем примере: читается что-то из сокета и затем отсылается обратно до тех пор, пока не будет прочитан специальный сигнал "*END*".

Ответственность за очистку сокета должна быть, внимательно спланирована. В этом случае, сокет создается вне *ServeOneJabber*, так что ответственность может быть совместная. Если конструктор *ServeOneJabber* завершится неудачей, он просто выбросит исключение тому, кто его вызвал, и кто должен очистить нить. Но если конструктор завершился успешно, то объект *ServeOneJabber* принимает ответственность за очистку нити на себя, в своем методе *run()*.

Обратите внимание на упрощенность *MultiJabberServer*. Как и прежде создается *ServerSocket* и вызывается метод *accept()*, чтобы позволить новое соединение. Но в это время возвращаемое значение метода *accept()* (сокет) передается в конструктор для *ServeOneJabber*, который создает новую нить для обработки этого соединения. Когда соединение завершится, нить просто умирает.

Если создание *ServerSocket* проваливается, то из метода *main()*, как и прежде, выбрасывается исключение. Но если создание завершается успешно, внешний блок *try-finally* гарантирует очистку. Внутренний *try-catch* гарантирует

только от сбоев в конструкторе *ServeOneJabber*. Если конструктор завершится успешно, то нить *ServeOneJabber* закроет соответствующий сокет.

Для проверки этого сервера, который реально обрабатывает несколько клиентов, приведенная ниже программа создает несколько клиентов (используя нити), которые соединяются с одним и тем же сервером. Максимальное допустимое число нитей определяется переменной *final int MAX_THREADS*.

```
//: c15:MultiJabberClient.java
//Клиент, который проверяет MultiJabberServer,
//запуская несколько клиентов.
//{RunByHand}

import java.net.*;
import java.io.*;

class JabberClientThread extends Thread
{
 private Socket socket;
 private BufferedReader in;
 private PrintWriter out;
 private static int counter = 0;
 private int id = counter++;
 private static int threadcount = 0;

 public static int threadCount()
 {
  return threadcount;
 }

 public JabberClientThread(InetAddress addr)
 {
  System.out.println("Making client " + id);
  threadcount++;
  try
  {
   socket = new Socket(addr, MultiJabberServer.PORT);
  }
  catch (IOException e)
  {
   System.err.println("Socket failed");
   //Если создание сокета провалилось, ничего ненужно чистить.
  }
  try
  {
   in=new BufferedReader(new InputStreamReader(socket.getInputStream()));
   //Включаем автоматическое выталкивание
   out=new PrintWriter(new BufferedWriter(new OutputStreamWriter
                                  (socket.getOutputStream())), true);
   start();
```

```
   }
   catch (IOException e)
   {
    //Сокет должен быть закрыт при любой ошибке, кроме ошибки конструктора сокета
    try
    {
     socket.close();
    }
    catch (IOException e2)
    {
     System.err.println("Socket not closed");
    }
   }
   //В противном случае сокет будет закрыт в методе run() нити.
   }

public void run()
{
 try
 {
  for (int i = 0; i < 25; i++)
  {
   out.println("Client " + id + ": " + i);
   String str = in.readLine();
   System.out.println(str);
  }
  out.println("END");
 }
 catch (IOException e)
 {
  System.err.println("IO Exception");
 }
 finally
 {
  //Всегда закрывает
  try
  {
   socket.close();
  }
  catch (IOException e)
  {
   System.err.println("Socket not closed");
  }
  threadcount--;
  //Завершаем эту нить
  }
 }
}

public class MultiJabberClient
{
 static final int MAX_THREADS = 40;
 public static void main(String[] args) throws IOException,
```

```
                                                    InterruptedException
 {
  InetAddress addr = InetAddress.getByName(null);
  while (true)
  {
   if (JabberClientThread.threadCount() < MAX_THREADS)
     new JabberClientThread(addr);
   Thread.currentThread().sleep(100);
  }
 }
}
```

Конструктор *JabberClientThread* принимает *InetAddress* и использует его для открытия сокета. Вероятно, вы заметили шаблон: сокет всегда используется для создания определенного рода объектов *Reader*'а и *Writer*'а (или *InputStream* и/или *OutputStream*), которые являются тем единственным путем, которым может быть использован сокет. (Вы можете, конечно, написать класс или два для автоматизации этого процесса вместо набора этого текста, если вас это беспокоит.) Далее, *start()* выполняет инициализацию нити и запуск *run()*. Здесь сообщение посылается на сервер, а информация с сервера отображается на экране. Однако нить имеет ограниченное время жизни и, в конечном счете, завершается. Обратите внимание, что сокет очищается, если конструктор завершился неудачей после создания сокета, но перед тем, как конструктор завершится. В противном случае, ответственность за вызов *close()* для сокета ложиться на метод *run()*.

Threadcount хранит информацию о том, сколько в настоящее время существует объектов *JabberClientThread*. Эта переменная инкрементируется, как часть конструктора, и декрементируется при выходе из метода *run()* (что означает, что нить умерла). В методе *MultiJabberClient.main()* вы можете видеть, что количество нитей проверяется, и если их много, то нить более не создается. Затем метод засыпает. Таким образом, некоторые нити, в конечном счете, умрут, и другие будут созданы. Вы можете поэкспериментировать с *MAX_THREADS*, чтобы увидеть, когда ваша конкретная система почувствует затруднения с множеством соединений.

Дейтаграммы

Пример, который вы недавно видели, использует *Transmission Control Protocol* (*TCP*, также известный, как сокет, основанный на потоках), который

предназначен для наибольшей надежности и гарантии, что данные будут доставлены. Он позволяет передавать повторно потерянные данные, он обеспечивает множественные пути через различные маршрутизаторы в случае, если один из них отвалится, а байты будут доставлены в том порядке, в котором они посланы. Весь этот контроль и надежность добавляют накладные расходы: *TCP* сильно перегружен.

Существует второй протокол, называемый *User Datagram Protocol* (*UDP*), который не гарантирует, что пакет будет доставлен и не гарантирует, что пакеты достигнут точки назначения в том же порядке, в котором они были отправлены. Он называется "ненадежным протоколом" (*TCP* является "надежным протоколом"), что звучит плохо, но так как он намного быстрее, он может быть полезнее. Существуют приложения, такие как аудио-сигнал, в которых не критично, если несколько пакетов потеряются здесь или там, а скорость жизненно необходима. Или, например сервер времени, для которого реально не имеет значения, если одно из сообщений будет потеряно. Также, некоторые приложения могут быть способны отправлять *UDP* сообщения к серверу и затем считать, если нет ответа в разумный период времени, что сообщения были потеряны.

Использование URL из апплета

Для апплета есть возможность стать причиной отображения любого *URL* с помощью *Web*-браузера, в котором запущен апплет. Вы можете сделать это с помощью следующей строки:

```
getAppletContext().showDocument(u);
```

которая и является объектом типа *URL*. Вот простой пример, который перенаправляет на другую страницу. Хотя вы просто перенаправляете на *HTML* страницу, можно также перенаправить на вывод, который дает *CGI* программа.

```
//: c15:ShowHTML.java
// <applet code=ShowHTML width=100 height=50>
// </applet>

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
```

```
import com.bruceeckel.swing.*;

public class ShowHTML extends JApplet
{
 JButton send = new JButton("Go");
 JLabel l = new JLabel();

 public void init()
 {
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  send.addActionListener(new Al());
  cp.add(send);
  cp.add(l);
 }

class Al implements ActionListener
{
 public void actionPerformed(ActionEvent ae)
 {
  try
  {
   //Это может быть CGI программа вместо HTML страницы
   URL u = new URL(getDocumentBase(), "FetcherFrame.html");
   //Отображается вывод URL с помощью Web-браузера, как обычная страница
   getAppletContext().showDocument(u);
  }
  catch (Exception e)
  {
   l.setText(e.toString());
  }
  }
 }

 public static void main(String[] args)
 {
  Console.run(new ShowHTML(), 100, 50);
 }
}
```

Красота класса *URL* состоит в том, что он отлично защищает вас. Можно соединиться с *Web*-серверами без знания многого из того, что происходит за занавесом.

Чтение файла с сервера

Вариация приведенной выше программы, читающей файл, расположенный на сервере. В этом случае файл указывается клиентом:

```
//: c15:Fetcher.java
// <applet code=Fetcher width=500 height=300>
// </applet>
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import com.bruceeckel.swing.*;

public class Fetcher extends JApplet
{
 JButton fetchIt = new JButton("Fetch the Data");
 JTextField f = new JTextField("Fetcher.java", 20);
 JTextArea t = new JTextArea(10, 40);

 public void init()
 {
  Container cp = getContentPane();
  cp.setLayout(new FlowLayout());
  fetchIt.addActionListener(new FetchL());
  cp.add(new JScrollPane(t));
  cp.add(f);
  cp.add(fetchIt);
}

public class FetchL implements ActionListener
{
 public void actionPerformed(ActionEvent e)
 {
  try
  {
   URL url = new URL(getDocumentBase(), f.getText());
   t.setText(url + "n");
   InputStream is = url.openStream();
   BufferedReader in=new BufferedReader(new InputStreamReader(is));
   String line;
   while ((line = in.readLine()) != null)
     t.append(line + "n");
  }
  catch (Exception ex)
  {
   t.append(ex.toString());
  }
 }
}

public static void main(String[] args)
{
 Console.run(new Fetcher(), 500, 300);
}
}
```

Создание объекта *URL* похоже на предыдущий пример – *getDocumentBase()* является начальной точкой, как и прежде, но в, то, же время, имя файла читается

из *JTextField*. Как только объект *URL* создан, его строковая версия помещается в *JTextArea*, так что вы можем видеть, как он выглядит. Затем из *URL* получается *InputStream*, который в данном случае может просто производить поток символов из файла. После конвертации в *Reader* и буферизации, каждая строка читается и добавляется в *JTextArea*. Обратите внимание, что *JTextArea* помещается внутрь *JScrollPane*, так что скроллирование обрабатывается автоматически.

Мультиплексирование, основанное на переключении в JDK 1.4

Когда вы читаете из сокета или пишете в него, вам нужно сделать передачу данных рациональной. Давайте рассмотрим сначала операцию записи. Когда вы пишите данные на уровне приложения (*TCP* или *UDP* сокет), вы пишите данные в рабочий буфер системы. Эти данные, в конечном счете, формируют (*TCP* или *UDP*) пакеты, которые необходимо передать на машину назначения по сети. Когда вы пишите в сокет и, если в буфере нет достаточно доступного места, запись может блокироваться. Если вы читаете из сокета и нет достаточного количества информации для чтения из буфера операционной системы, куда попадают данные после получения из сети, чтение будет блокировано. Если есть нить (поток) для операции чтения или записи, эта нить не может делать ничего и может стать причиной снижения производительности вашей программы. До появления *JDK 1.4* не было способа вывести такую нить из заблокированного состояния. С помощью каналов вы можете выполнить асинхронную операцию закрытия на канале и нить, блокированная на этом канале примет *AsynchronousCloseException*.

Асинхронный ввод-вывод в *Java* достигается тем же способом, который дает вызов метода *select()* в *UNIX* подобных системах. Вы можете дать список дескрипторов (чтения или записи) в функцию *select()* и она отследит этот дескриптор на возникновение некоторых событий. Для дескриптора, представляющего сокет, из которого вы читаете, данные в буфере операционной системы для этого буфера представляются событием. Для дескрипторов, представляющих сокеты, в которые вы пишите, наличие места для записи во внутреннем буфере операционной системы для этого сокета представляется

событием. Поэтому вызов метода *select()* исследует различные дескрипторы для проверки событий.

Что, если вы просто читаете и пишите в дескриптор? *Select* может обрабатывать множество дескрипторов, что позволит вам мониторить множество сокетов. Рассмотрим пример чат-сервера, когда сервер имеет соединения с различными клиентами. Тип данных, достигающих сервера, перемежается. Сервер предназначен для чтения данных из сокета и отображения их в *GUI*, то есть для показа каждому клиенту – чтобы достичь этого, вы читаете данные от каждого клиента и пишите эти данные всем остальным клиентам.

Например, 5 клиентов: 1, 2, 3, 4 и 5. Если сервер запрограммирован на выполнение чтения от 1 и записи в 2, 3, 4 и 5, затем происходит чтения от 2 и запись в 1, 3, 4, 5 и так далее, то может так случиться, что пока нить сервера заблокирована на чтении одного из клиентских сокетов, могут появиться данные на других сокетах. Одно из решений состоит в том, чтобы создавать различные нити для каждого клиента (до *JDK1.4*). Но это не масштабируемое решение. Вместо этого вы можете иметь селектор, основанный на механизме, следящем за всеми клиентскими сокетами. Он знает, какой сокет имеет данные для чтения без блокирования. Но если единственная нить выполняет эту работу (выбор и запись каждому клиенту), он не будет хорошо откликаться. Таким образом, в таких ситуациях одна нить мониторит сокеты на чтение, выбирает сокет, из которого можно осуществить чтение, и делегирует остальную ответственность (запись другим клиентам) другой нити (нитям) или пулу нитей.

Этот шаблон называется шаблоном реактора, когда события отсоединяются от действия, ассоциированного с событиями (*Pattern Oriented Software Architecture – Doug Schmidt*).

В *JDK 1.4* вы создаете канал, регистрируете объект Селектора в канале, который (объект) будет следить за событиями в канале. Многие каналы регистрируют один и тот же объект Селектора. Единственная нить, которая вызывает *Selector.select()*, наблюдает множество каналов. Каждый из классов *ServerSocket*, *Socket* и *DatagramSocket* имеют метод *getChannel()*, но он возвращает

null за исключением того случая, когда канал создается с помощью вызова метода *open()* (*DatagramChannel.open(), SocketChannel.open(), ServerSocketChannel.open()*). Вам необходимо ассоциировать сокет с этим каналом.

Вы мультиплексируете несколько каналов (то есть сокеты), используя Селектор. Статический вызов *Selector.select()* блокирует выполнение до возникновения события в одном из каналов. Существует так же и не блокирующая версия этого метода, которая принимает количество миллисекунд для засыпания или блокирования до того момента, когда вызов метода завершится.

ByteBuffer используется для копирования данных из канала и в канал. *ByteBuffer* является потоком сокетов и вы декодируете этот поток, как символы. Со стороны клиента в *MultiJabberClient.java* это выполняется путем использования классов *Writer* и *OutputStreamWriter*. Эти классы конвертируют символы в поток байтов.

Приведенная ниже программа *NonBlockingIO.java* объясняет, как вы можете использовать Селектор и Канал для выполнения мультиплексирования. Эта программа требует запущенного Сервера. Она может стать причиной исключения на сервере, но ее назначение не в коммуникации с сервером, а в том, чтобы показать, как работает *select()*.

```
//: TIEJ:X1:NonBlockingIO.java
// Сокет и Селектор сконфигурированы для не блокированного
// Соединения с JabberServer.java
// {RunByHand}
import java.net.*;
import java.nio.channels.*;
import java.util.*;
import java.io.*;

/**
```

Цель: Показать как использовать селектор. Нет чтения/записи, просто показывается готовность к совершению операции.

Алгоритм:

– создаем селектор;

– создаем канал;

– связываем сокет, ассоциированный с каналом, с «клиентским портом»;

– конфигурируем канал, как не блокирующий;

– регистрируем канал в селекторе;

– вызываем метод *select()*, чтобы он блокировал выполнение до тех пор, пока канал не будет готов (как это предполагается методом *select(long timeout)*;

– получаем множество ключей, относящихся к готовому каналу для работы, основной интерес состоит в том, когда они зарегистрированы с помощью селектора;

– перебираем ключи;

– для каждого ключа проверяем, что соответствующий канал готов к работе, в которой он заинтересован;

– если он готов, печатаем сообщение о готовности.

Примечание: необходим запущенный *MultiJabberServer* на локальной машине.

Вы запускаете его и соединяетесь с локальным *MultiJabberServer*.

Он может стать причиной исключения в *MultiJabberServer*, но это исключение ожидаемо.

```
**/
public class NonBlockingIO
{
 public static void main(String[] args) throws IOException
 {
  if (args.length < 2)
  {
   System.out.println("Usage: java <client port> <local server port>");
   System.exit(1);
  }
  int cPort = Integer.parseInt(args[0]);
  int sPort = Integer.parseInt(args[1]);
  SocketChannel ch = SocketChannel.open();
  Selector sel = Selector.open();
  try
  {
   ch.socket().bind(new InetSocketAddress(cPort));
   ch.configureBlocking(false);
   //Канал заинтересован в выполнении чтения/записи/соединении
   ch.register(sel,SelectionKey.OP_READ|SelectionKey.OP_WRITE
```

```
                                        |SelectionKey.OP_CONNECT);
    //Разблокируем, когда готовы к чтению/записи/соединению
    sel.select();
    //ключи, относящиеся к готовому каналу, канал заинтересован в работе,
    //которая может быть выполнена in can be без блокирования
    Iterator it = sel.selectedKeys().iterator();
    while (it.hasNext())
    {
     SelectionKey key = (SelectionKey) it.next();
     it.remove();
     //Если связанный с ключом канал готов к соединению
     //if((key.readyOps() & SelectionKey.OP_CONNECT) != 0){
     if (key.isConnectable())
     {
      InetAddress ad = InetAddress.getLocalHost();
      System.out.println("Connect will not block");
      //Вы должны проверить возвращаемое значение,
      //чтобы убедиться, что он соединен. Этот не блокированный
      //вызов может вернуться без соединения, когда
      //нет сервера, к которому вы пробуете подключиться
      //Поэтому вы вызываете finishConnect(), который завершает
      //операцию соединения.
      if (!ch.connect(new InetSocketAddress(ad, sPort)))
        ch.finishConnect();
     }
     //Если канал, связанный с ключом, готов к чтению
     //if((key.readyOps() & SelectionKey.OP_READ) != 0)
     if (key.isReadable())
       System.out.println("Read will not block");
     //Готов ли канал, связанный с ключом, к записи
     //if((key.readyOps() & SelectionKey.OP_WRITE) != 0)
     if (key.isWritable())
        System.out.println("Write will not block");
    }
   }
   finally
   {
    ch.close();
    sel.close();
   }
  }
}
```

Как указано выше, вам необходимо создать канал, используя вызов метода *open(). SocketChannel.open()* создает канал. Так как он наследован от *AbstractSelectableChannel (DatagramChannel и SocketChannel)*, он имеет функциональность для регистрации себя в селекторе. Вызов метода регистрации совершает это. В качестве аргумента он принимает Селектор для регистрации канала, и события, которые интересны для этого канала. Здесь показано, что

SocketChannel заинтересован в соединении, чтении и записи – поэтому в вызове метода регистрации указано *SelectionKey.OP_CONNECT*, *SelectionKey.OP_READ* и *SelectionKeyOP_WRITE* наряду с Селектором.

Статический вызов метода *Selector.select()* наблюдает все каналы, зарегистрированные в нем, относительно тех событий, которые указаны (второй аргумент при регистрации). Вы можете иметь каналы, заинтересованные в нескольких событиях.

Следующий пример работает так же, как и *JabberClient1.java*, но использует Селектор.

```
//: TIEJ:X1:JabberClient1.java
//Очень простой клиент, который просто посылает строки на сервер
//и читает строки, посылаемые сервером
//{RunByHand}
import java.net.*;
import java.util.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;

public class JabberClient1
{
 public static void main(String[] args) throws IOException
 {
  if (args.length < 1)
  {
   System.out.println("Usage: java JabberClient1 <client-port>");
   System.exit(1);
  }
  int clPrt = Integer.parseInt(args[0]);
  SocketChannel sc = SocketChannel.open();
  Selector sel = Selector.open();
  try
  {
   sc.configureBlocking(false);
   sc.socket().bind(new InetSocketAddress(clPrt));
   sc.register(sel,SelectionKey.OP_READ|SelectionKey.OP_WRITE
                                        |SelectionKey.OP_CONNECT);
   int i=0;
   //По причине асинхронной природы, вы не знаете
   //когда чтение и запись закончены, поэтому вам необходимо
   //следить за этим, переменная boolean written используется
   // для переключения между чтением и записью. Во время записи
   //отосланные назад символы должны быть прочитаны.
   //Переменная boolean done используется для проверки, когда нужно
   //прервать цикл.
```

```
   boolean written = false, done = false;
   //JabberServer.java, которому этот клиент подсоединяется, пишет с
   //помощью
   //BufferedWriter.println(). Этот метод выполняет
   //перекодировку в соответствии с кодовой страницей по умолчанию
   String encoding = System.getProperty("file.encoding");
   Charset cs = Charset.forName(encoding);
   ByteBuffer buf = ByteBuffer.allocate(16);
   while (!done)
   {
    sel.select();
    Iterator it = sel.selectedKeys().iterator();
    while (it.hasNext()) {
    SelectionKey key = (SelectionKey) it.next();
    it.remove();
    sc = (SocketChannel) key.channel();
    if (key.isConnectable() && !sc.isConnected())
    {
     InetAddress addr = InetAddress.getByName(null);
     boolean success = sc.connect(new InetSocketAddress(addr,
                                                  JabberServer.PORT));
     if (!success)
        sc.finishConnect();
    }
    if (key.isReadable() && written)
    {
     if (sc.read((ByteBuffer) buf.clear()) > 0)
     {
      written = false;
      String response=cs.decode((ByteBuffer) buf.flip()).toString();
      System.out.print(response);
      if (response.indexOf("END") != -1)
        done = true;
     }
    }
    if (key.isWritable() && !written)
    {
     if (i < 10)
      sc.write(ByteBuffer.wrap(new String("howdy "+i+'n').getBytes()));
     else if (i == 10)
            sc.write(ByteBuffer.wrap(new String("ENDn").getBytes()));
     written = true;
     i++;
    }
   }
  }
 }
 finally {
  sc.close();
  sel.close();
  }
 }
}
```

Следующий пример показывает простой механизм, основанный на селекторе, для *MultiJabberServer*, обсуждаемый ранее. Этот сервер работает таким же образом, как и старый сервер, но он более эффективен в том, что он не требует отдельной нити для обработки каждого клиента.

```
//: TIEJ:X1:MultiJabberServer1.java
//Имеет туже семантику, что и многопоточный
//MultiJabberServer
//{RunByHand}
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;

/**
     Сервер принимает соединения не блокирующим способом. Когда
соединение установлено, создается сокет, который регистрируется с
селектором для чтения/записи. Чтение/запись выполняется над этим
сокетом, когда селектор разблокируется. Эта программа работает точно
так же, как и MultiJabberServer.
**/
public class MultiJabberServer1
{
 public static final int PORT = 8080;

 public static void main(String[] args) throws IOException
 {
  //Канал будет читать данные в ByteBuffer, посылаемые
  //методом PrintWriter.println(). Декодирование этого потока
  //байт требует кодовой страницы для кодировки по умолчанию.

  String encoding = System.getProperty("file.encoding");

  //Инициализируем здесь, так как мы не хотим создавать новый
  //экземпляр кодировки каждый раз, когда это необходимо
  //Charset cs = Charset.forName(
  //System.getProperty("file.encoding"));

  Charset cs = Charset.forName(encoding);
  ByteBuffer buffer = ByteBuffer.allocate(16);
  SocketChannel ch = null;
  ServerSocketChannel ssc = ServerSocketChannel.open();
  Selector sel = Selector.open();
  try
  {
   ssc.configureBlocking(false);
   //Локальный адрес, на котором он будет слушать соединения
   //Примечание: Socket.getChannel() возвращает null, если с ним не
   //ассоциирован канал, как показано ниже
```

```
   //т.е. выражение (ssc.socket().getChannel() != null) справедливо

   ssc.socket().bind(new InetSocketAddress(PORT));
   //Канал заинтересован в событиях OP_ACCEPT

   SelectionKey key=ssc.register(sel, SelectionKey.OP_ACCEPT);
   System.out.println("Server on port: " + PORT);
   while (true)
   {
    sel.select();
    Iterator it = sel.selectedKeys().iterator();
    while (it.hasNext())
    {
     SelectionKey skey = (SelectionKey) it.next();
     it.remove();
     if (skey.isAcceptable())
     {
      ch = ssc.accept();
      System.out.println("Accepted connection from:"+ ch.socket());
      ch.configureBlocking(false);
      ch.register(sel, SelectionKey.OP_READ);
     }
     else
     {
      //Обратите внимание, что не выполняется проверка, если
      //в канал можно писать или читать - для упрощения
      ch = (SocketChannel) skey.channel();
      ch.read(buffer);
      CharBuffer cb = cs.decode((ByteBuffer) buffer.flip());
      String response = cb.toString();
      System.out.print("Echoing : " + response);
      ch.write((ByteBuffer) buffer.rewind());
      if (response.indexOf("END") != -1)
        ch.close();
      buffer.clear();
     }
    }
   }
  finally
  {
   if (ch != null)
      ch.close();
   ssc.close();
   sel.close();
  }
 }
}
```

Здесь приведена простейшая реализация Пула Нитей. В этой реализации нет полинга (занят-ожидает) нитей. Она полностью основана на методах *wait()* и *notify()*.

```
//: TIEJ:X1:Worker.java
//Instances of Worker are pooled in threadpool
//{Clean: WorkerErr.log, WorkerErr.log.lck}
//{RunByHand}

import java.io.*;
import java.util.logging.*;

public class Worker extends Thread
{
  public static final Logger logger = Logger.getLogger("Worker");
  private String workerId;
  private Runnable task;
  //Необходима ссылка на пул нитей в котором существует нить, чтобы
  //нить могла добавить себя в пул нитей по завершению работы.
  private ThreadPool threadpool;
  static
  {
   try
   {
     logger.setUseParentHandlers(false);
     FileHandler ferr = new FileHandler("WorkerErr.log");
     ferr.setFormatter(new SimpleFormatter());
     logger.addHandler(ferr);
   }
   catch (IOException e)
   {
    System.out.println("Logger not initialized..");
   }
  }

 public Worker(String id, ThreadPool pool)
 {
  workerId = id;
  threadpool = pool;
  start();
 }

 //ThreadPool, когда ставит в расписание задачу, использует этот метод
 //для делегирования задачи Worker-нити. Кроме того для установки
 //задачи (типа Runnable) он также переключает ожидающий метод
 //run() на начало выполнения задачи.
 public void setTask(Runnable t)
 {
  task = t;
  synchronized (this)
  {
   notify();
  }
 }

 public void run()
 {
```

```
  try
  {
   while (!threadpool.isStopped())
   {
    synchronized (this)
    {
     if (task != null)
     {
      try
      {
       task.run(); // Запускаем задачу
      }
      catch (Exception e)
      {
       logger.log(Level.SEVERE,"Exception in source Runnable task", e);
      }
      //Возвращает себя в пул нитей
      threadpool.putWorker(this);
     }
     wait();
    }
   }
   System.out.println(this + " Stopped");
  }
  catch (InterruptedException e)
  {
   throw new RuntimeException(e);
  }
 }

 public String toString()
 {
  return "Worker : " + workerId;
 }
}
```

Основной алгоритм:

while true:

1. Проверить очередь задач.

2. Если она пуста, подождать, пока в очередь будет добавлена задача (вызов метода *addTask()* добавляет задачу и уведомляет очередь для разблокирования).

3. Пробуем получить рабочую (Worker) нить из пула нитей.

4. Если нет ни одной доступной нити, ожидаем в пуле нитей (когда нить освободится, она уведомит пул нитей для разблокировки).

5. На этой стадии есть задачи в очереди, и есть свободная рабочая нить.

6. Делегируем задачу из очереди рабочей нити.

end while

```
//: TIEJ:X1:ThreadPool.java
//Пул нитей, которые выполняют задачи
//{RunByHand}

import java.util.*;

public class ThreadPool extends Thread
{
 private static final int DEFAULT_NUM_WORKERS = 5;
 private LinkedList workerPool = new LinkedList(),taskList=new LinkedList();
 private boolean stopped = false;

 public ThreadPool()
 {
  this(DEFAULT_NUM_WORKERS);
 }

 public ThreadPool(int numOfWorkers)
 {
  for (int i = 0; i < numOfWorkers; i++)
    workerPool.add(new Worker("" + i, this));
  start();
 }

 public void run()
 {
  try
  {
   while (!stopped)
   {
    if (taskList.isEmpty())
    {
     synchronized (taskQueue)
     {
      //Если очередь пустая, подождать, пока будет добавлена задача
      taskList.wait();
     }
    }
    else if (workerPool.isEmpty())
    {
     synchronized(workerPool)
     {
      //Если нет рабочих нитей, подождать, пока не появится
      workerPool.wait();
     }
    }
    //Запускаем следующую задачу из расписания задач
    getWorker().setTask((Runnable) taskList.removeLast());
   }
  }
  catch (InterruptedException e)
```

```
 {
  throw new RuntimeException(e);
 }
}

public void addTask(Runnable task)
{
 taskList.addFirst(task);
 synchronized (taskList)
 {
  taskList.notify(); // Если добавлена новая задача, уведомляем
 }
}

public void putWorker(Worker worker)
{
 workerPool.addFirst(worker);
 //Здесь может быть случай, когда вы будете иметь пул из 5 нитей, а
 //будет требоваться больше. Это происходит тогда, когда требуется
 //рабочая нить, но ее нет (свободной), тогда просто блокируем пул нитей
 //Это событие, при котором появляется свободная рабочая нить
 //пуле нитей
 //Поэтому эта нить посылает уведомление и разблокирует
 //нить ThreadPool, ожидающую пул нитей
 synchronized (workerPool)
 {
  workerPool.notify();
 }
}

private Worker getWorker()
{
 return (Worer) workerPool.removeLast();
}

public boolean isStopped()
{
 return stopped;
}

public void stopThreads()
{
 stopped = true;
 Iterator it = workerPool.iterator();
 while (it.hasNext())
 {
  Worker w = (Worker) it.next();
  synchronized (w)
  {
   w.notify();
  }
 }
}
```

```
 // Junit test

 public void testThreadPool()
 {
  ThreadPool tp = new ThreadPool();
  for (int i = 0; i < 10; i++)
  {
   tp.addTask(new Runnable()
   {
    public void run()
    {
     System.out.println("A");
    }
   });
  }
  tp.stopThreads();
 }
}
```

Следующий пример *MultiJabberServer2.java* использует пул нитей. Это шаблон Реактора. Как установлено выше, события отделяются от ассоциированных с ними действий. Пул нитей асинхронно разделяет действия, ассоциированные с событиями. В системах масштаба предприятия такое разделение обычно достигается путем использования Системы сообщений *Java – Java Messaging System* (*JMS*).

```
//: TIEJ:X1:MultiJabberServer2.java
//Семантика аналогична MultiJabberServer1, с использованием пула нитей.
//{RunByHand}

import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.util.*;

class ServeOneJabber implements Runnable
{
 private SocketChannel channel;
 private Selector sel;

 public ServeOneJabber(SocketChannel ch) throws IOException
 {
  channel = ch;
  sel = Selector.open();
 }

 public void run()
 {
```

```
ByteBuffer buffer = ByteBuffer.allocate(16);
boolean read = false, done = false;
String response = null;
try
{
 channel.register(sel,SelectionKey.OP_READ|SelectionKey.OP_WRITE);
 while (!done)
 {
  sel.select();
  Iterator it = sel.selectedKeys().iterator();
  while (it.hasNext())
  {
   SelectionKey key = (SelectionKey) it.next();
   it.remove();
   if (key.isReadable() && !read)
   {
    if (channel.read(buffer) > 0)
       read = true;
    CharBuffer cb=MultiJabberServer2.CS.decode((ByteBuffer)buffer.flip());
    response = cb.toString();
   }
   if (key.isWritable() && read)
   {
    System.out.print("Echoing : " + response);
    channel.write((ByteBuffer) buffer.rewind());
    if (response.indexOf("END") != -1)
       done = true;
    buffer.clear();
    read = false;
   }
  }
 }
}
catch (IOException e)
{
 //будет поймано Worker.java и залогировано.
 //Необходимо выбросить исключение времени выполнения,
 //так как мы не можем оставить IOException
 throw new RuntimeException(e);
}
finally
{
 try
 {
  channel.close();
 }
 catch (IOException e)
 {
  System.out.println("Channel not closed.");
  //Выбрасываем это, чтобы рабочая нить могла залогировать.
  throw new RuntimeException(e);
 }
}
```

```
 }
}

public class MultiJabberServer2
{
 public static final int PORT = 8080;
 private static String encoding = System.getProperty("file.encoding");
 public static final Charset CS = Charset.forName(encoding);
 //Создаем пул нитей с 20 рабочими нитями
 private static ThreadPool pool = new ThreadPool(20);

 public static void main(String[] args) throws IOException
 {
  ServerSocketChannel ssc = ServerSocketChannel.open();
  Selector sel = Selector.open();
  try
  {
   ssc.configureBlocking(false);
   ssc.socket().bind(new InetSocketAddress(PORT));
   SelectionKey key = ssc.register(sel, SelectionKey.OP_ACCEPT);
   System.out.println("Server on port: " + PORT);
   while (true)
   {
    sel.select();
    Iterator it = sel.selectedKeys().iterator();
    while (it.hasNext())
    {
     SelectionKey skey = (SelectionKey) it.next();
     it.remove();
     if (skey.isAcceptable())
     {
      SocketChannel channel = ssc.accept();
      System.out.println("Accepted connection from:"+ channel.socket());
      channel.configureBlocking(false);
      //Отделяем события и ассоциированное действие
      pool.addTask(new ServeOneJabber(channel));
     }
    }
   }
  }
  finally
  {
   ssc.close();
   sel.close();
  }
 }
}
```

Это минимальное обновления для *JabberServer.java*. Изначально, когда клиент посылает "*END*", *JabberServer* не отправляет его назад. Эта версия *JabberServer* отсылает строку "*END*" назад. Эти изменения были сделаны, чтобы упростить *JabberClient1.java*.

```
//: TIEJ:X1:JabberServer.java
// Очень простой сервер, который просто
// отсылает назад то, что получил от клиента.
// {RunByHand}

import java.io.*;
import java.net.*;

public class JabberServer
{
 //Выбираем порт за пределами диапазона 1-1024
 public static final int PORT = 8080;

 public static void main(String[] args) throws IOException
 {
  ServerSocket s = new ServerSocket(PORT);
  System.out.println("Started: " + s);
  try
  {
   //Блокируем до возникновения соединения
   Socket socket = s.accept();
   try
   {
    System.out.println("Connection accepted: " + socket);
    BufferedReader in = new BufferedReader
                        (new InputStreamReader(socket.getInputStream()));
    //Вывод автоматически выталкивается PrintWriter
    BufferedWriter out = new BufferedWriter(new
                        OutputStreamWriter(socket.getOutputStream()));
    while (true)
    {
     String str = in.readLine();
     System.out.println("Echoing: " + str);
     out.write(str, 0, str.length());
     out.newLine();
     out.flush();
     if (str.equals("END"))
     break;
    }
    //Всегда закрываем два сокета...
   }
   finally
   {
    System.out.println("closing...");
    socket.close();
   }
  }
  finally
  {
   s.close();
  }
 }
}
```

Упражнения

1. Скомпилируйте и запустите программы *JabberServer* и *JabberClient*. Отредактируйте файл и удалите всю буферизацию для ввода и вывода, затем скомпилируйте и запустите программу опять, чтобы посмотреть результат.

2. Создайте сервер, который спрашивает пароль, затем открывает файл и посылает файл по сетевому соединению. Создайте клиента, который соединяется с этим сервером, передает пароль, затем получает и сохраняет файл. Проверьте эту пару программ на вашей машине, использую *localhost* (*IP* адрес локальной заглушки 127.0.0.1, производимый при вызове *InetAddress.getByName(null)*).

3. Измените, сервер в упражнении 2 так, чтобы он использовал множественные нити для обработки множества клиентов.

4. Измените *JabberClient.java* таким образом, чтобы для вывода не происходило выталкивания буфера, и пронаблюдайте эффект.

5. Измените *MultiJabberServer* таким образом, чтобы он использовал пул нитей. Вместо выбрасывания нити при каждом отсоединении клиента, нить должна помещать себя в "пул доступных" нитей. Когда новый клиент хочет соединиться, сервер будет искать нить в пуле доступных нитей, чтобы обработать запрос, а если нет доступной нити, создается новая. Таким образом, количество необходимых нитей будет расти до требуемого количества. Назначение пулинга нитей в том, что нет затрат на накладные расходы при создании и разрушении новой нити для каждого нового клиента.

6. Начиная с *ShowHTML.java*, создайте апплет, который будет защищенным паролем шлюзом к определенной части вашего *web*-сайта.

Пример клиентского приложения: *SampleClient.java*

```
import java.net.*;
class SampleClient extends Thread
{
 public static void main(String args[])
 {
  try
  {
   Socket s = new Socket("localhost", 3128);
   //берем поток вывода и выводим первый аргумент
```

```
   //заданный при вызове, адрес открытого сокета и его порт
   args[0]=args[0]+"\n"+s.getInetAddress().getHostAddress()+":"+s.getLocalPort();
   s.getOutputStream().write(args[0].getBytes());
   //читаем ответ
   byte buf[] = new byte[64*1024];
   int r = s.getInputStream().read(buf);
   String data = new String(buf, 0, r);
   //выводим ответ в консоль
   System.out.println(data);
  }
  //вывод исключений
  catch (Exception e)
  {
   System.out.println("init error: "+e);
  }
 }
}
```

Пример серверного приложения: *SampleServer.java*

```
import java.io.*;
import java.net.*;

class SampleServer extends Thread
{
 Socket s;
 int num;

 public static void main(String args[])
 {
  try
  {
   //счетчик подключений
   int i = 0;
   ServerSocket server=new ServerSocket(3128,0,InetAddress.getByName("localhost"));
   System.out.println("server is started");
   //слушаем порт
   while (true)
   {
    //ждем нового подключения, после чего запускаем обработку клиента
    new SampleServer(i,server.accept());
    i++;
   }
  }
  //вывод исключений
  catch (exception e)
  {
   System.out.println("init error: "+e);
  }
 }
 public SampleServer(int num,Socket s)
 {
  //копируем данные
```

```
  this.num = num;
  this.s = s;
  //и запускаем новый вычислительный поток (смотри функцию run())
  setDaemon(true);
  setPriority(NORM_PRIORITY);
  start();
 }
 public void run()
 {
  try
  {
     //из сокета клиента берем поток входящих данных
   InputStream is = s.getInputStream();
     //и оттуда же – поток данных от сервера к клиенту
   OutputStream os = s.getOutputStream();
     //буффер данных в 64 килобайта
   bytebuf[] = newbyte[64*1024];

   //читаем 64 кб от клиента, результат – кол-во реально принятых данных
   int r = is.read(buf);

   //создаем строку, содержащую полученную от клиента информацию
   String data = new String(buf,0,r);

   //добавляем данные об адресе сокета
   data = "" + num + ":" + "\n" + data;

   //выводим данные
   os.write(data.getBytes());

   //завершаем соединение
   s.Close();
  }
  //вывод мсключений
  catch (Exception e)
  {
   System.out.println("init error: "+e);
  }
 }
```

После компиляции, получаем файлы *SampleServer.class* и *SampleClient.class* (все программы здесь и далее откомпилированы с помощью *JDK v1.4*) и запускаем вначале сервер:

```
javaSampleServer
```

а потом, дождавшись надписи "*serverisstarted*", и любое количество клиентов:

```
javaSampleClient test1
javaSampleClient test2

javaSampleClienttestN
```

Если во время запуска программы-сервера, вместо строки "*serverisstarted*" выдало строку типа:

init error: java.net.BindException: Address already in use: JVM_Bind

то это будет обозначать, что порт 3128 на вашем компьютере уже занят какой-либо программой или запрещён к применению политикой безопасности.

Следующий пример кода это серверное приложение, которое получает текстовые строки от клиентского приложения с помощью потоковых сокетов, отображает их на консоли и отправляет обратно клиенту.

```
import java.net.*;
importjava.util.*;
import java.io.*;

public class SocketServer
{
 public static void main(String args[])
 {
  System.out.println("* Socket Server *");
  ServerSocket ss = null;
  try
  {
   ss = new ServerSocket(9999);
  }
  catch (Exception ex)
  {
   System.out.println(ex.ToString());
   System.exit(0);
  }
  int nPort = ss.getLocalPort();
  System.out.println("Local Port: "+nPort);
  ServerThread sThread = null;
  sThread = new ServerThread(ss);
  sThread.start();
  System.out.println("Waiting connection...");
  try
  {
   sThread.join();
  }
  catch(InterruptedException ex)
  {
   System.out.println(ex.toString());
  }
  try
  {
   ss.close();
  }
  catch (Exception ex)
  {
```

```
    System.out.println(ex.toString());
   }
   System.exit(0);
  }
 }

//Class ServerThread
  classServerThread extends Thread
   {
    ServerSocketss = null;
    Socket s = null;

//ServerThread
  publicServerThread(ServerSocketsSocket)
   {
    ss = sSocket;
   }

//run
  public void run()
   {
    InputStream is;
    OutputStreamos;
    try
    {
     s = ss.accept();
    }
    catch (Exception ex)
    {
     stop();
    }
    System.out.println("Connected.");
    try
    {
     is = s.getInputStream();
     os = s.getOutputStream();
     //Прием команд, их обработка и передача результата клиенту
     while(true)
     {
      String szStr = recvString(is);
      sendString(os, "* " + szStr + " *");
      os.flush();
      System.out.println(szStr);
      if(szStr.equals("quit"))
         break;
     }
     is.close();
     os.close();
    }
    catch(Exception ex)
    {
      stop();
    }
```

```
    try
    {
      s.close();
      ss.close();
    }
    catch(Exceptionex)
    {
      stop();
    }
   }

//sendString
```

Метод в цикле извлекает из строки очередной символ и записывает его в выходной поток. После записи всех символов добавляется символ новой строки "*n*", который в данном случае используется как разделитель, и сбрасываем буфер выходного потока.

```
 static void sendString(OutputStreamos,String s)throwsIOException
 {
  for(int i = 0; i <s.length(); i++)
  {
   os.write((byte)s.charAt(i));
  }
  os.write('\n');
  os.flush();
 }

//recvString
```

Метод выполняет посимвольное чтение из входного потока *is* класса *InputStream* и возвращает строку, сформированную из прочитанных символов:

```
 static String recvString(InputStream is)throwsIOException
  {
   String szBuf = "";
   intch = is.read();
   while (ch>= 0 &&ch != '\n')
   {
    szBuf += (char)ch;
    ch = is.read();
   }
   returnszBuf;
  }
}
```

Программа клиентского приложения, которое посылает серверу текстовые строки, введенные пользователем с консоли, получает ответ от сервера и отображает его на консоли.

```
 import java.net.*;
```

```
importjava.util.*;
import java.io.*;

public class SocketClient
{
 public static void main(String args[])
 {
  System.out.println("* Socket Ckient *");
  Socket s = null;
  try
  {
   s = new Socket("localhost",9999);
  }
  catch (Exception ex)
  {
   System.out.println(ex.ToString());
   System.exit(0);
  }
  int nPort = s.getLocalPort();
  System.out.println("Local Port: "+nPort);
  InputStream is;
  OutputStream os;
  try
  {
   is = s.getInputStream();
   os = s.getOutputStream();
   String szStr;
   while(true)
   {
    szStr = getKbdString();
    sendString(os, szStr);
    os.flush();
    if(szStr.equals("quit"))
      break;
    szStr = recvString(is);
    System.out.println(szStr);
   }
   is.close();
   os.close();
   s.close();
  }
  catch (Exception ex)
  {
   System.out.println(ex.toString());
  }
 }

//sendString
  static void sendString(OutputStreamos,String s)throwsIOException
   {
    for(int i=0; i<s.length(); i++)
     {
      os.write((byte)s.charAt(i));
```

```
     }
    os.write('\n');
    os.flush();
   }

//recvString
  static String recvString(InputStream is)throwsIOException
  {
   String szBuf = "";
   intch = is.read();
   while (ch>=0 && ch != '\n')
   {
    szBuf += (char)ch;
    ch = is.read();
   }
   returnszBuf;
  }

//getKbdString
  static public String getKbdString()
  {
   bytebKbd[] = new byte[256];
   intiCnt = 0;
   String szStr = "";
   try
   {
    iCnt = System.in.read(bKbd);
   }
   catch (Exception ex)
   {
    System.out.println(ex.toString());
   }
   szStr = new String(bKbd, 0, iCnt);
   szStr = szStr.trim();
   returnszStr;
  }
}
```

Реализация сетевого приложения на языке *Java* с использованием протокола *UDP*

Пример 1. Листинг программы сервера:

```
importjava.io.*;
importjava.net.*;
importjava.util.*;

public class DatagramServer
{
 public static void main(String args[])
 {
  //массив для ввода строки с клавиатуры
  byte bKbdInput[] = new byte[256];
  //буфер для чтения команд
```

```
byte buf[] = new byte[512];
//сокет сервера
DatagramSocket s;
//принимаемый пакет
DatagramPacket pinp;
//адрес узла, откуда пришел принятый пакет
InetAddress SrcAddress;
//порт, откуда пришел принятый пакет
int SrcPort;
try
{
 System.out.println("Datagramm Socket Server Application");
}
catch(Exception ioe)
{
 System.out.println(ioe.toString());
}
try
{
 //создаем сокет сервера
 s = new DatagramSocket(9998);
 //создаем пакет для приема команд
 pinp = new DatagramPacket(buf, 512);
 while (true)
 {
  //принимаем пакет от клиента
  s.receive(pinp);
  //получаем адрес узла, приславшего пакет
  SrcAddress = pinp.getAddress();
  //получаем порт, на котором был передан пакет
  SrcPort = pinp.getPort();
  //формируем строку из принятого блока
  Strinr str = new String(buf,0);
  //обрезаем строку, удаляя символы конца строки
  StringTokenizerst;
  st = new StringTokenizer(str,"\r\n");
  str = new String((String)st.nextElement());
  //выводим строку команды на консоль
  System.out.println("> " + str + "  <" + "port: "+ SrcPort);
  //если пришла команда quit, прерываем цикл
  if (str.equals("quit"))
    break;
 }
 s.close();
}
catch(Exception ioe)
{
 System.out.println(ioe.toString());
}
try
{
 System.out.println("Press <Enter> to terminate application...");
 System.in.read(bKbdInput);
```

```
  }
  catch(Exception ioe)
  {
   System.out.println(ioe.toString());
  }
 }
}
```

Пример 2. Листинг программы клиента *DatagramClient.java*:

```
importjava.io.*;
importjava.net.*;
importjava.util.*;

public class DatagramClient
{
 public static void main(String args[])
 {
  //массив для ввода строки с клавиатуры
  byte bKbdInput[] = new byte[256];
  //размер введенной строки
  int length;
  //рабочая строка
  String str;
  //сокет клиента
  DatagramSocket s;
  //передаваемый пакет
  DatagramPacket pout;
  try
  {
   System.out.println("Datagramm Socket Client Application" +
                      "\nEnter any string or 'quit' to exit...");
  }
  catch(Exception ioe)
  {
   System.out.println(ioe.toString());
  }
  try
  {
   //получаем адрес локального узла
   InetAddress OutAddress = InetAddress.getLocalHost();
   //создаем сокет с использованием любого свободного порта
   s = new DatagramSocket();
   //создаем передаваемый пакет
   pout=new DataframPacket(bKbdInput,bKbdInput.length,OutAddress, 9998);
   while(true)
   {
    //читаем строку команды с клавиатуры
    length = System.in.read(bKbdInput);
    //если строка не пустая, обрабатываем ее
    if(length !=1)
    {
     //преобразуем строку в формат String
     str = new String(bKbdInput, 0);
```

```
      //обрезаем строку, удаляя символы конца строки
      StringTokenizer st;
      st = new StringTokenizer(str,"\n");
      str = new String((String)st.nextElement());
      //выводим передаваемую строку команды на консоль для контроля
      System.out.println("> " + str);
      //посылаем пакет серверу
      s.send(pout);
      //если введена команда quit, прерываем цикл
      if (str.equals("quit"))
        break;
     }
    }
    s.close();
   }
   catch(Exception ioe)
   {
    System.out.println(ioe.toString());
   }
   try
   {
    System.out.println("Press <Enter> to terminate application...");
    System.in.read(bKbdInput);
   }
   catch(Exception ioe)
   {
    System.out.println(ioe.toString());
   }
  }
}
```

Задания для самостоятельного выполнения

1. Реализовать рассмотренные выше примеры.

2. Создать приложение, которое широковещательно осуществляет отправку сообщений, детали задания уточнить у преподавателя.

3. Реализовать клиент-серверное приложение, обменивающееся сообщениями.

4. Выполнить упражнения (2, 3, 5).

5. Создать два клиента, передающих сообщения посредством сервера. Сервер ведет подсчет переданных данных и предоставляет их по требованию клиента.

6. Выполнить задание варианта:

6.1.Сервер генерирует значение от 1 до 10, передает клиенту, в зависимости от полученного значения рисуется количество кругов.

6.2.Сервер генерирует значения от 1 до 4, передает клиенту, в зависимости от полученного значения рисуется соответствующий примитив: 1 – круг, 2 – прямоугольник, 3- многоугольник, 4 – линия.

6.3.Сервер генерирует значения от 1 до 4, передает клиенту, в зависимости от полученного значения примитив закрашивается в определенный цвет: 1 – синий, 2 – красный, 3 – зеленый, 4 – желтый.

6.4.В клиентском приложении генерируется значение цвета светофора и прорисовывается, который передается на сервер, который в свою очередь возвращает команду к действию: «Ждать», «Приготовиться», «Двигаться».

7. Выполнить индивидуальное задание.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Опишите алгоритм установления соединения в сетевом приложении на языке *Java*.

2. Схема сетевого приложения на языке *Java*.

3. Методы языка *Java*, используемые в при организации соединения по протоколу *UDP*.

4. Методы языка *Java*, используемые в при организации передачи данных по протоколу *UDP*.

5. Методы языка *Java*, используемые в при организации передачи файлов по протоколу *UDP*.

Лабораторная работа 5

Тема: «Моделирование клиент-серверной системы удаленных вычислений методом *Map-Reduce*»

Целью лабораторной работы является моделирование системы удаленных вычислений, т.е. системы, предоставляющей свое машинное время для решения трудоемких задач по клиентским запросам. Имеются сервера, считающиеся системами с высокой вычислительной мощностью, и клиент, решающий некоторую np-полную задачу путем разделения ее на подзадачи, раздачи частей по серверам (*map*) и дальнейшего сокращения результатов (*reduce*).

В качестве решаемой задачи рекомендуется брать np-полные задачи, легко разделяющиеся на более мелкие задачи, например, вычисление простых чисел, вычисление совершенных чисел и пр.

Используемые технологии: ввод-вывод, сетевое взаимодействие, многопоточность, сериализация, загрузчик классов, *reflection*.

Требуется написать приложение типа *client-server*.

Сервер является приложением, принимающим соединения по порту, указанному в его конфигурационных параметрах. При получении запроса сервер устанавливает соединение с клиентом в отдельном потоке. Сервер может поддерживать соединение одновременно с произвольным количеством клиентов, но одновременно исполняет задачи только одного клиента.

Требования к серверу. Сервер умеет выполнять два действия = регистрация задачи клиента и выполнение задачи клиента. Регистрация задачи клиента заключается в получении по сети *class*-файла и загрузки его в *java*-машину. По условиям задачи считается, что сервер знает интерфейс *class*-файла.

Требования к клиенту. Клиент должен уметь регистрировать произвольное количество серверов. Клиент умеет передавать серверу установленное соединение класс-файл, представляющий задачу, после чего сервер регистрирует этот класс-файл и считается способным исполнять код клиента. По условию интерфейс класс-файла известен. Далее клиент опрашивает сервера на предмет – свободны ли они, если свободны то отправляет им части задачи и собирает ответы.

Задание. Реализовать возможность загрузки кода в виде текста на *javascript*.

Задание. Использовать *ForkJoinPool*.

При демонстрации лабораторной работы необходимо продемонстрировать работоспособность системы при одновременно подключении клиента к десяти серверам. Клиент должен информировать пользователя о прогрессе работы. Вычисления должны производиться с неограниченной точностью.

Далее будет приведена информация о некоторых необходимых для реализации задачи технологиях.

Map-Reduce

Алгоритм *Map-Reduce* не следует путать с одноименным фреймворком распределенных вычислений, хотя в определенном смысле целью лабораторной работы является построение упрощенной модели этого фреймворка.

Идея алгоритма состоит в том, что клиентский процесс делит задачу на куски и рассылает серверам, которые генерируют ему ответы. Эти ответы клиент склеивает в конечный результат.

Например, перед клиентом стоит задача найти все простые числа от одного до ста. Допустим, у клиента имеется три подключенных сервера вычислений. Клиент делит задачу, например, на десять диапазонов по 10 чисел и рассылает серверам, собирая их ответы (шаг *Map*). Получившиеся ответы склеиваются в конечный результат (шаг *Reduce*).

Ввод и вывод

В *java* существует две основных технологии ввода и вывода, а именно, «старый» ввод вывод через потоки (для бинарных файлов) либо через объекты *Reader* и *Writer* (для текстовых файлов) и «новый» ввод-вывод *NIO*. *NIO* является новым весьма условно, так как впервые появился в *Java 1.4.2*; тем не менее, в *Java 7 NIO* был существенно переработан. В процессе выполнения данной лабораторной работы у пользователя возникает задача чтения файла, которая может быть решена с помощью любого из этих подходов.

Тем не менее, рекомендуется использовать *NIO* по причине краткости кода. Следующий пример иллюстрирует применение *NIO* для чтения коротких файлов

на примере файла *c:/myfile.dat* :

```
byte[] data=Files.readAllBytes(Paths.get("c:/myfile.dat"));
System.out.println("Read "+data.length+" bytes");
```

Установление сетевого подключения

Сетевое подключение между двумя процессами устанавливается через так называемый механизм сетевых сокетов.

Сокет – это абстракция, состоящая из сетевого адреса и сетевого порта. Как внутри устроен сокет для решения задачи установления сетевого соединения знать абсолютно не нужно, этот вопрос полностью отдается на совесть операционной системы. Для нас важно знать, что сокет умеет устанавливать соединение с другим сокетом, адрес и порт которого мы знаем, после чего возвращает нам два потока – поток ввода и поток вывода, которые могут использоваться для передачи информации от клиента серверу и назад.

В соединении один из сокетов всегда должен быть клиентским, а другой – серверным. Клиентский сокет имеет тип *java.net.Socket*. Типовой код для установления соединения выглядит так:

```
SocketAddress addr=new InetSocketAddress(адрес_сервера, номер_порта);
Socket client = new Socket();
try
{
  client.connect(addr, таймаут_на_подключение);
  if (!client.isConnected())
  {
    //сюда мы попадаем, если соединение не установилось
    System.out.println("Сервер не отвечает");
    return;
  }
  InputStream inStream = client.getInputStream();
  OutputStream outStream = client.getOutputStream();
  //Здесь идет основной код программы
}
finally
{
  try
  {
    client.close();
  }
  catch (Exception e)
  {
    e.printStackTrace();
  }
```

```
}
```

Серверный сокет имеет тип *java.net.ServerSocket*. Логика работы этого сокета заключается в том, что он вызывает метод *bind*, связывающий его с сокетом, а затем в бесконечном цикле вызывает метод *accept()*. Каждый раз, когда серверный сокет получает запрос на соединение, он создает новый клиентский сокет на одном из свободных портов операционной системы, и переадресует входящий запрос на него, после чего возвращает его в качестве результата вызова *accept()*. Серверный код должен взять этот новый сокет и передать отдельному потоку – обработчику команд, после чего продолжить бесконечный цикл. Типичный пример кода сервера:

```
try
{
  java.net.ServerSocket serverSocket=new java.net.ServerSocket();
  SocketAddress address= new InetSocketAddress
              (InetAddress.getByName(адрес_Сервера),порт_Сервера);
  serverSocket.bind(address);
  for(;;)
  {
    Socket s=serverSocket.accept();
    //Здесь идет код по перемещению обработки в другой поток
  }
}
catch (Exception e)
{
  e.printStackTrace();
}
finally
{
  serverSocket.close();
}
```

Classloader и *Reflection*

Каким образом мы можем взять произвольный поток байт, преобразовать его в класс, создать экземпляр этого класса и выполнить его методы? Эта задача решается с помощью технологий *classloader* и *reflection*.

Classloader – загрузчик классов – отвечает за преобразование потока байт в определение класса. В случае, если класс хранится каким-либо нестандартным методом – например, лежит на сети, в виде *xml*, зашифрован, а также в любых других нестандартных вариантах, может потребоваться написание собственного загрузчика классов, способного к получению данных из нестандартного места. В

нашем случае нам надо преобразовать только поток байт, пришедший по сети, и для этого можно использовать стандартный загрузчик классов *ClassLoader.getSystemClassLoader()*.

У полученного загрузчика классов есть два интересующих нас метода. Сначала необходимо вызвать метод *defineClass()*, который преобразует поток байт в класс. Затем необходимо вызвать метод *resolve()*, который собственно, подготовит полученный класс к использованию.

С этого момента мы имеем класс, с которым можем работать через технологию *Reflection*.

Вначале необходимо получить объект класса *Class*, представляющий данный класс. Это делается вызовом вида:

```
Class c = Class.forName("имя класса");
```

Этот объект может ответить на вопрос, какие в классе есть методы и переменные, соответственно, с помощью методов *getConstructors()*, *getMethods()* и *getFields()*. Наша задача состоит в:

– конструировании экземпляра данного класса;

– вызове метода *execute()*.

Создать новый объект заданного класса можно двумя способами:

1) Вызвать *c.newInstance()*, что приведет к вызову конструктора по умолчанию.

2) Второй – получить ссылку на конструктор с помощью вызова *getConstructors()* или аналогичного, и вызвать метод *newInstance* уже у этого конструктора.

Полученный объект будет объектом данного класса, но, так как на этапе компиляции программы мы про этот класс ничего не знаем, нам по прежнему придется работать через *reflection*. Для того, чтобы вызвать определенный метод, необходимо найти его с помощью метода *getMethods()*, после чего вызвать на полученном методе *invoke()*.

Таким образом, две упомянутые технологии – *ClassLoader* и *Reflection* – позволяют преобразовать поток байт в исполняемый код и выполнить этот код.

Сериализация

Сериализация – запись и чтение объектов из потока. Для сериализации используются специальные классы, представляющие поток – *ObjectInputStream* и *ObjectOutputStream*.

Объект любого класса, который реализует интерфейс *java.lang.Serializable*, может быть сохранен в поток и восстановлен из потока с помощью методов *writeObject* и *readObject*. Некоторые классы, для которых напрямую не указываются предки, также реализуют этот интерфейс, например, массивы. Важно знать, что все классы коллекций (списки, словари, множества) реализуют этот интерфейс, таким образом, сохранение коллекции или массива возможно. Если все элементы этой коллекции или массива поддерживают интерфейс *Serializable*.

При сериализации в поток сохраняются и потом восстанавливаются из потока все поля, которые не объявлены статическими и не имеют модификатора *transient*. В случае если сериализуемый класс наследуется от класса, не реализующего интерфейс *Serializable*, то последний должен иметь конструктор по умолчанию (т.е. без аргументов), который будет вызван для инициализации всех членов класса, унаследованных сериализуемым классом. Десериализованного класса конструктор и секции инициализации не вызываются. Члены класса, объявленные как *transient*, будут инициализированы значениями по умолчанию.

В некоторых ситуациях программист может посчитать, что его не устраивают стандартные средства сериализации и десериализации. В таком случае он может переопределить то, как они выполняются для объекта, с помощью переопределения следующих методов:

```
private void writeObject(java.io.ObjectOutputStream out)
     throws IOException
private void readObject(java.io.ObjectInputStream in)
     throws IOException, ClassNotFoundException;
private void readObjectNoData()
     throws ObjectStreamException;
```

В некоторых случаях может потребоваться сериализовать в поток один класс, а восстановить другой. Для этого программист может переопределить методы:

```
Object writeReplace() throws ObjectStreamException;
Object readResolve() throws ObjectStreamException;
```

Первый должен быть переопределен у сериализуемого класса, а второй – у десериализуемого.

В некоторых ситуациях может оказаться, что класс, который был сериализован в поток, на другой стороне имеет другую версию (например, вследствие изменений при разработке клиента, в то время как сервер не менялся). При десериализации с целью проверки совместимости проверяется версия класса, которая генерируется автоматически. Соответственно, если код изменился, возможны ошибки, связанные с проверкой версии. Для того чтобы указать, что это тот же самый класс, необходимо указать его версию сериализации напрямую, например:

```
static final long serialVersionUID = 42L;
```

Задания для самостоятельного выполнения

ВАРИАНТ 1. Посчитать количество каждой из букв в текстовом файле.

ВАРИАНТ 2. Посчитать количество типов файлов на диске (*.doc, .docx, .jpg, .txt, .xls*).

ВАРИАНТ 3. Найти все простые числа в диапазоне от 100 до 10^7.

ВАРИАНТ 4. Найти все совершенные числа в диапазоне от 100 до 10^7.

ВАРИАНТ 5. Найти все числа полиндромы в диапазоне от 100 до 10^7.

ВАРИАНТ 6. Найти все числа Армстронга, т.е. сумма его цифр, возведенных в 3 степень, равна самому числу (например: 153=13 +53+33) в диапазоне от 100 до 10^7

ВАРИАНТ 7. Найти все числа в диапазоне от 100 до 10^7, в которых цифры числа различны.

ВАРИАНТ 8. Найти все числа в диапазоне от 100 до 10^7, в которых сумма цифр числа кратна 3.

ВАРИАНТ 9. В числах диапазона от 100 до 10^7 выбросить из записи числа цифры 0 и 5, оставив прежним порядок остальных цифр, например: из числа 1509 должно получиться 19.

ВАРИАНТ 10. Найти все числа в диапазоне от 100 до 10^7, в которых сумма цифр числа и само число кратны 7.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. В чем заключается технология *MapReduce*.
2. Приведите пример для решения каких профессиональных задач целесообразно использовать технологию *MapReduce*.
3. Обоснуйте алгоритм, используемый в задании.
4. Роль сериализации в сетевых приложениях.

Лабораторная работа 6

Тема: «Реализация сетевого приложения с использованием технологии *P2P*»

Технология организации одноранговых сетей (*peer-to-peer networking*), часто называемая технологией *P2P*, является одной из самых полезных и при этом часто неправильно понимаемых среди средств, появившихся в последние несколько лет. Когда люди думают о *P2P*, им на ум, как правило, приходит лишь одна вещь: возможность обмена музыкальными, или видео файлами, зачастую незаконным образом. Это связано с тем, что приложения для обмена файлами наподобие *BitTorrent* стали очень популярными, а в них для работы используется именно технология *P2P*.

Однако, хотя технология *P2P* применяется в приложениях для обмена файлами, это вовсе не означает, что она не может использоваться в других приложениях. На самом деле эта технология может применяться в целом ряде других приложений, и она становится все более и более важной в современном мире повсеместных коммуникаций.

В *Microsoft* тоже не обошли стороной появление технологии *P2P* и стали разрабатывать собственные инструменты и средства для ее применения. Так появилась платформа *Microsoft Windows Peer-to-Peer Networking*, исполняющая роль своего рода каркаса для коммуникаций в приложениях *P2P*. В состав этой платформы входят такие важные компоненты, как *PNRP* (*Peer Name Resolution Protocol* – протокол преобразования имен членов) и *PNM* (*People Near Me* – соседние пользователи).

Кроме того, в версию *.NET Framework 3.5* было включено новое пространство имен *System.Net.PeerToPeer* и несколько новых типов и средств, позволяющих создавать приложения *P2P* с минимальными усилиями.

Обзор технологии *P2P*

Технология *P2P* представляет собой альтернативный подход к организации сетевых коммуникаций. Для того чтобы понять, чем *P2P* отличается от стандартного подхода к обеспечению коммуникаций, не помешает сделать шаг назад и вспомнить, что собой представляет связь типа «клиент-сервер».

Коммуникации такого типа очень часто применяется в современных сетевых приложениях.

Архитектура типа «клиент-сервер»

Традиционно взаимодействие с приложениями по сети (в том числе Интернет) организуется с использованием архитектуры типа «клиент-сервер». Прекрасным примером могут служить веб-сайты. При просмотре веб-сайта происходит отправка по Интернет соответствующего запроса веб-серверу, который затем возвращает требуемую информацию. Если необходимо загрузить какой-то файл, это делается напрямую с веб-сервера.

Аналогично, настольные приложения, имеющие возможность подключения к локальной или глобальной сети, обычно устанавливают соединение с каким-то одним сервером, например, сервером баз данных или сервером, предоставляющим набор служб.

На рисунке 1 ниже показан простой вариант архитектуры типа «клиент-сервер»:

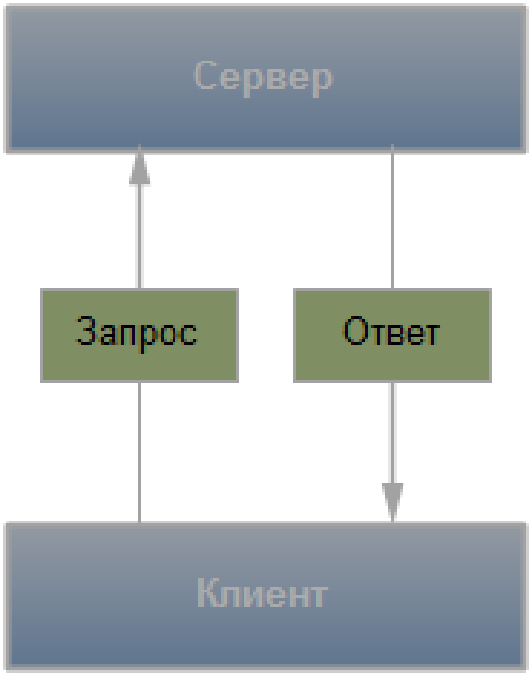

Рисунок 1 – Архитектура «клиент-сервер»

Ничего по сути неправильного в такой архитектуре нет, и на самом деле во многих случаях она будет оказываться именно тем, что нужно. Однако ей

присуща проблема с масштабируемостью. На рисунке 2 показано, как она будет масштабироваться при добавлении дополнительных клиентов:

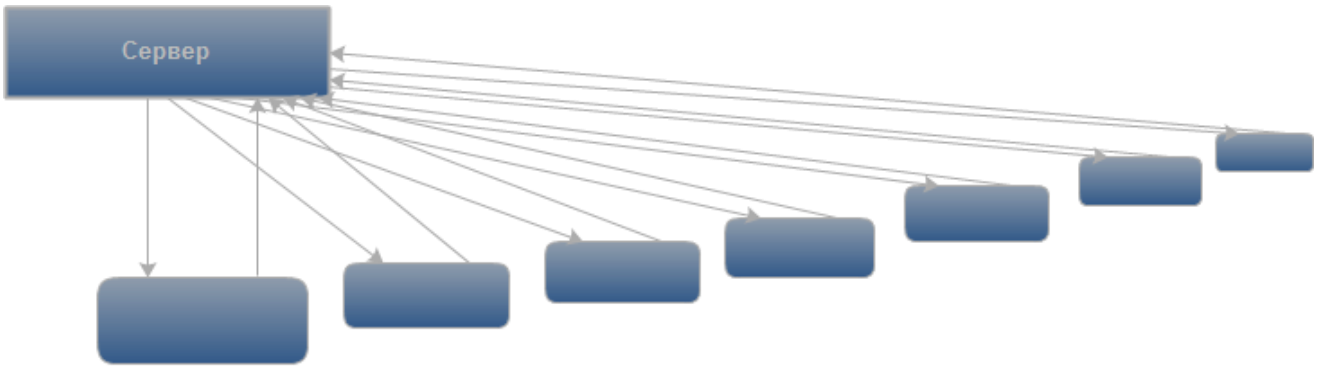

Рисунок 2 – Масштабируемость архитектуры «клиент-сервер»

С добавлением каждого клиента нагрузка на сервер, который должен взаимодействовать с каждым клиентом, будет увеличиваться. Если снова взять пример с веб-сайтом, то такое увеличение нагрузки может стать причиной выхода веб-сайта из строя. При слишком большом трафике сервер просто перестанет реагировать на запросы.

Конечно, существуют варианты масштабирования, с помощью которых можно смягчить подобную ситуацию. Один из них предусматривает масштабирование «вверх» за счет увеличения мощи и ресурсов сервера, а другой – масштабирование «вширь» путем добавления дополнительных серверов.

Первый способ, естественно, ограничивается доступными технологиями и стоимостью более мощного оборудования.

Второй способ потенциально более гибкий, но требует добавления дополнительного уровня в инфраструктуру для обеспечения клиентов возможностью либо взаимодействовать с отдельными серверами, либо поддерживать состояние сеанса независимо от сервера, с которым осуществляется взаимодействие. Для этого доступна масса решений, таких как продукты, позволяющие создавать веб-фермы или фермы серверов.

Архитектура типа *P2P*

Одноранговый (*peer-to-peer*) подход полностью отличается от подхода с масштабированием «вверх» или «вширь». В случае применения *P2P* вместо того, чтобы сосредоточить усилия на попытках улучшить коммуникации между

сервером и его клиентами, все внимание уделяется поиску способов, которыми клиенты могут взаимодействовать между собой.

Давайте для примера представим, что веб-сайтом, с которым взаимодействуют клиенты, является www.williamspublishing.com, а издательство *Williams* объявило о выходе новой книги на этом сайте и предоставлении его для бесплатной загрузки всем желающим, но лишь на протяжении одного дня.

Не трудно догадаться, что при таком положении дел накануне появления книги веб-сайт начнет просматривать масса людей, которые будут постоянно обновлять его содержимое в своих браузерах и ожидать появления файла.

Как только файл станет доступным, все они одновременно начнут пытаться загрузить его и, скорее всего, веб-сервер, который обслуживает веб-сайт, не выдержит такого натиска и выйдет из строя.

Чтобы предотвратить выход веб-сервера из строя, можно воспользоваться технологией *P2P*. Вместо отправки файла прямо с сервера сразу всем клиентам он может быть отправлен только определенному числу клиентов. Несколько остальных клиентов могут далее загрузить его у тех клиентов, у которых он уже есть.

После этого еще несколько клиентов могут загрузить его у клиентов, получивших его вторыми, и т.д.

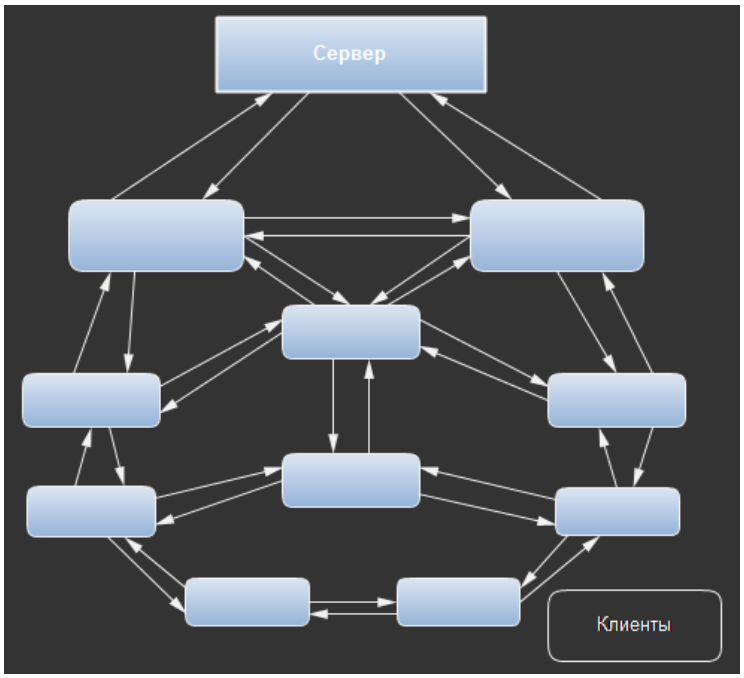

Рисунок 3 – Файлообменная система *BitTorrent*

По сути, этот процесс может происходить даже быстрее благодаря разбиению файла на куски и распределению этих кусков среди клиентов, одни из которых будут загружать их прямо с сервера, а другие – из других клиентов.

Именно так и работают технологии файлообменных систем вроде *BitTorrent*, как показано на рисунке 3.

Особенности архитектуры *P2P*

Тем не менее, в описанной здесь архитектуре обмена файлами все равно остались кое-какие проблемы, которые должны быть решены. Для начала, каким образом клиенты узнают о том, что существуют другие клиенты, и как они будут обнаруживать фрагменты файла, которые, возможно, имеются у других клиентов? Кроме того, каким образом гарантировать оптимальное взаимодействие между клиентами, если их могут отделять друг от друга континенты?

Каждый клиент, участвующий в работе сетевого приложения P2P, для преодоления этих проблем должен быть способен выполнять следующие операции:

- обнаруживать других клиентов;
- подключаться к другим клиентам;
- взаимодействовать с другими клиентами.

В том, что касается способности обнаруживать других клиентов, возможны два очевидных решения: поддержка списка клиентов на сервере, чтобы клиенты могли получать его и связываться с другими клиентами (называемыми *peers* – равноправными участниками), либо использование инфраструктуры (например, *PNRP*), которая позволяет клиентам обнаруживать друг друга напрямую. В большинстве файлообменных систем применяется решение с поддержкой списка на сервере и используются серверы, называемые «трекерами» (*trackers*).

В файлообменных системах в роли сервера может также выступать и любой клиент, как показано на рисунке выше, объявляя, что у него имеется доступный файл, и регистрируя его на сервере-трекере. На самом деле в чистой сети *P2P* вообще не нужны никакие серверы, а лишь равноправные участники.

Проблема подключения к другим клиентам является более тонкой и распространяется на всю структуру используемой приложением *P2P* сети. При наличии одной группы клиентов, в которой все должны иметь возможность взаимодействовать друг с другом, топология соединений между этими клиентами может приобретать чрезвычайно сложный вид. Зачастую производительность удается улучшать за счет создания нескольких групп клиентов с возможностью установки подключения между клиентами в каждой из них, но не с клиентами в других группах.

В случае создания этих групп по принципу локальности можно добиться дополнительного повышения производительности, поскольку в таком случае клиенты получают возможность взаимодействовать друг с другом по более коротким (с меньшим числом прыжков) сетевым путям между машинами.

Способность взаимодействовать с другими клиентами, пожалуй, не так важна, поскольку существуют хорошо зарекомендовавшие себя протоколы вроде *TCP/IP*, которые вполне могут применяться и здесь. Конечно, допускается привносить свои улучшения, как в высокоуровневые технологии (например, использовать службы *WCF*, получая в распоряжение все предлагаемые ими функциональные возможности), так и в низкоуровневые протоколы (например, применять протоколы многоадресной рассылки и тем самым обеспечивать отправку данных во множество конечных точек одновременно).

Обеспечение клиентов возможностью обнаруживать, подключаться и взаимодействовать друг с другом играет центральную роль в любой реализации *P2P*.

Терминология *P2*

Ранее уже было представлено понятие равноправного участника (*peer*) – именно так называют клиентов в сети *P2P*.

Слово «клиент» в сети *P2P* не имеет никакого смысла, потому что здесь нет обязательного сервера, клиентом которого нужно быть.

Группы равноправных участников, которые соединяются друг с другом, называются ячейками (*meshes*), облаками (*clouds*) или графами (*graphs*).

Каждая отдельная группа считается хорошо соединенной, если соблюдено, хотя бы какое-то одно из следующих условий:

✓ Между каждой парой равноправных участников существует путь соединения, позволяющий каждому участнику подключаться к другому равноправному участнику требуемым образом.

✓ Между каждой парой равноправных участников существует относительно небольшое количество соединений, по которым они могут связываться.

✓ Удаление одного равноправного участника из группы не лишает остальных равноправных участников возможности взаимодействия друг с другом.

Обратите внимание, что это вовсе не означает, что каждый равноправный участник должен обязательно иметь возможность подключаться к каждому другому равноправному участнику напрямую. На самом деле, если проанализировать сеть с математической точки зрения, то можно обнаружить, что для соблюдения упомянутых выше условий равноправным участникам необходимо иметь возможность подключаться к относительно небольшому количеству других равноправных участников.

Еще одним понятием в технологии *P2P*, о котором следует знать, является волновое распространение(*flooding*). Под волновым распространением подразумевается способ, которым один фрагмент данных может передаваться по сети всем равноправным участникам и которым может производиться опрос других узлов в сети для обнаружения конкретного фрагмента данных. В неструктурированных сетях *P2P* этот процесс протекает довольно произвольно; при этом сначала устанавливается связь с ближайшими соседними равноправными участниками, которые затем, в свою очередь, связываются со своими ближайшими соседями, и т.д. до тех пор, пока не будет охвачен каждый равноправный участник в сети.

Также допускается создавать и структурированные сети *P2P* с четко определенными путями, по которым должно происходить распространение запросов и данных среди равноправных участников.

Решения *P2P*

При наличии подходящей инфраструктуры для *P2P* можно начинать разрабатывать не просто улучшенные версии клиент-серверных приложений, но и совершенно новые приложения. Технология *P2P* особенно подходит для приложений следующих классов:

- ✓ приложения, предназначенные для распространения содержимого, в том числе упоминавшиеся ранее приложения обмена файлами;
- ✓ приложения, предназначенные для совместной работы, такие как приложения, позволяющие открывать общий доступ к рабочему столу и "белой доске" (whiteboard);
- ✓ приложения, предназначенные для обеспечения многопользовательской связи и позволяющие пользователям общаться и обмениваться данными напрямую, а не через сервер;
- ✓ приложения, предназначенные для распределения обработки, как альтернатива приложениям для суперкомпьютеров, которые обрабатывают огромные объемы данных;
- ✓ приложения Web 2.0, объединяющие в себе некоторые или все перечисленные выше приложения и превращающие их в динамические веб-приложения следующего поколения.

Платформа *Microsoft Peer-to-Peer Networking*

Платформа *Microsoft Windows Peer-to-Peer Networking* (Платформа для создания одноранговых сетей *Windows* производства *Microsoft*) представляет собой предлагаемую *Microsoft* реализацию технологии *P2P*. Она поставляется в составе операционных систем *Windows XP SP2, Windows Vista, Windows 7* и *Windows 8*, а также доступна в виде дополнения для *Windows XP SP1*. Она включает в себя две технологии, которые можно использовать для создания приложений *P2P* в *.NET*:

Протокол *Peer Name Resolution Protocol* (*PNRP*)	Его можно применять для публикации и преобразования адресов равноправных участников сети.
Сервер *People Near Me*	Его можно применять для обнаружения локальных равноправных участников (и который в настоящее время доступен только в Windows Vista и Windows 7)

Протокол *PNRP*

Естественно, для реализации приложения *P2P* можно использовать любой имеющийся в распоряжении протокол, но при работе в среде *Microsoft Windows* имеет смысл хотя бы рассмотреть вариант применения протокола *PNRP*. К настоящему времени успело выйти две версии этого протокола. Первая версия предлагалась в составе *Windows XP SP2, Windows XP Professional x64 Edition* и *Windows XP SP1* с пакетом *Advanced Networking Pack for Windows XP*. Вторая версия была выпущена вместе *Windows Vista* и сделана доступной для пользователей *Windows XP SP2* в виде отдельного загружаемого компонента.

Первая и вторая версии *PNRP* не совместимы между собой, и потому здесь рассматривается только вторая версия.

Сам по себе протокол *PNRP* не предоставляет всего, что необходимо для создания приложения *P2P*. Скорее, он является лишь одной из основополагающих технологий, которые применяются для преобразования адресов равноправных участников сети *P2P*.

Протокол *PNRP* позволяет клиенту регистрировать конечную точку (называемую именем равноправного участника(*peer name*)), которая автоматически распространяется между остальными равноправными участниками в группе (облаке). Это имя инкапсулируется в идентификатор *PNRP* (*PNRP ID*). Любой равноправный участник, который обнаруживает идентификатор *PNRP*, может использовать *PNRP* для его преобразования в имя самого равноправного участника и затем начинать взаимодействовать с соответствующим клиентом напрямую.

Например, можно определить имя равноправного участника, представляющее конечную точку службы *WCF*, и воспользоваться *PNRP* для его регистрации в группе (облаке) в виде идентификатора *PNRP ID*. Равноправный участник, выполняющий подходящее клиентское приложение, которое применяет механизм обнаружения, способный распознавать имена равноправных участников для предоставляемой службы, тогда сможет обнаружить этот идентификатор *PNRP ID*. После обнаружения участник с помощью протокола *PNRP* установит

местонахождение конечной точки службы *WCF* и начнет пользоваться этой службой.

Важным моментом является то, что *PNRP* не делает никаких предположений относительно того, что на самом деле скрывается за именем равноправного участника. Право решать, как использовать это имя после обнаружения оставляется за самими равноправными участниками.

Информация, которую равноправный участник получает от *PNRP* при преобразовании идентификатора *PNRP*, включает в себя адрес *IPv6*, а также обычно и адрес *IPv4* участника, который опубликовал этот идентификатор, вместе с номером порта и, необязательно, небольшим количеством дополнительных данных. Если равноправный участник не знает, что означает имя другого равноправного участника, он вряд ли сможет сделать с этой информацией что-нибудь полезное.

Идентификаторы *PNRP*

Идентификаторы *PNRP ID* являются 256-битными значениями. Младшие 128 бит используются для обозначения уникальным образом индивидуального равноправного участника сети, а старшие 128 бит – для представления его имени. Старшие 128 бит представляют собой хеш-комбинацию, состоящую из хешированного значения открытого ключа, которое получается от осуществляющего публикацию равноправного участника, и строки длиной до 149 символов, которая представляет имя этого участника.

Хешированный открытый ключ (называемый авторитетным источником (*authority*)) в сочетании с этой строкой (называемой классификатором (*classifier*)) называются идентификатором *P2P*. Вместо хешированного открытого ключа также может использоваться значение *0*, в случае чего имя равноправного участника считается незащищенным (если применяется открытый ключ, то имя считается защищенным).

На рисунке 4 схематично показана структура идентификатора *PNRP*:

Рисунок 4 – Структура идентификатора *PNRP*

Служба *PNRP* на стороне равноправного участника отвечает за поддержание списка идентификаторов *PNRP*, как тех, которые публикует сама, так и тех, которые она получает в виде кэшированного списка от экземпляров служб *PNRP*, находящихся в других местах облака. Когда равноправный участник пытается преобразовать идентификатор *PNRP*, служба *PNRP* либо использует кэшированную копию конечной точки для выяснения адреса того равноправного узла, который опубликовал данный идентификатор *PNRP*, либо спрашивает его соседей, не могут ли они сделать это.

В конечном итоге с узлом, опубликовавшим идентификатор *PNRP*, устанавливается соединение, и служба *PNRP* получает возможность преобразовать его.

Обратите внимание, что все это не требует участия администратора. Все, что понадобится сделать – позаботиться о том, чтобы равноправные участники знали, что следует делать с именами после их преобразования с помощью своей локальной службы *PNRP*.

Равноправные участники сети могут применять *PNRP* для нахождения идентификаторов *PNRP*, совпадающих с определенным идентификатором *P2P*. Эту возможность можно использовать для реализации простейшего механизма, позволяющего обнаруживать незащищенные имена равноправных участников. Дело в том, что в случае отображения несколькими равноправными участниками незащищенного имени с одинаковым классификатором, их идентификатор *P2P ID* будет совпадать.

Конечно, из-за того, что любой равноправный участник может использовать незащищенное имя, нет никакой гарантии, что конечная точка, с которой будет устанавливаться соединение, будет именно той, которая ожидается, поэтому такое решение подходит лишь для реализации механизма обнаружения по локальной сети.

Облака *PNRP*

Выше было рассказано, как *PNRP* осуществляет регистрацию и преобразование имен равноправных участников в облаке (группе). Облако (или группа) поддерживается *seed*-сервером, которым может быть любой сервер с запущенной службой *PNRP*, которая поддерживает запись хотя бы об одном равноправном участнике. Для службы *PNRP* доступны облака двух типов:

Облака локальных соединений (*Link local*)	В такие облака входят компьютеры с подключением к локальной сети. Каждый ПК может подключаться к более чем одному облаку такого типа при условии наличия у него нескольких сетевых адаптеров
Глобальные облака (*Global*)	В такие облака по умолчанию входят компьютеры с подключением к Интернету, хотя также можно определять приватное глобальное облако. Разница состоит в том, что для глобального облака с подключением к Интернету *Microsoft* поддерживает *seed*-сервер, а для определяемого самостоятельно приватного глобального облака должен использоваться собственный *seed*-сервер. В последнем случае необходимо позаботиться о том, чтобы все равноправные участники могли подключаться к нему, настроив соответствующим образом параметры политики

В прошлых выпусках *PNRP* существовали облака еще и третьего типа, которые назывались облаками локальных сайтов (*Site local*). Они больше уже не применяются и потому не рассматриваются.

Для выяснения, в какие облака входит данный компьютер, служит следующая команда (*cmd)*:

```
netsh p2p pnrp cloud show list
```

На рисунке 5 показано, как обычно выглядит результат запуска этой команды:

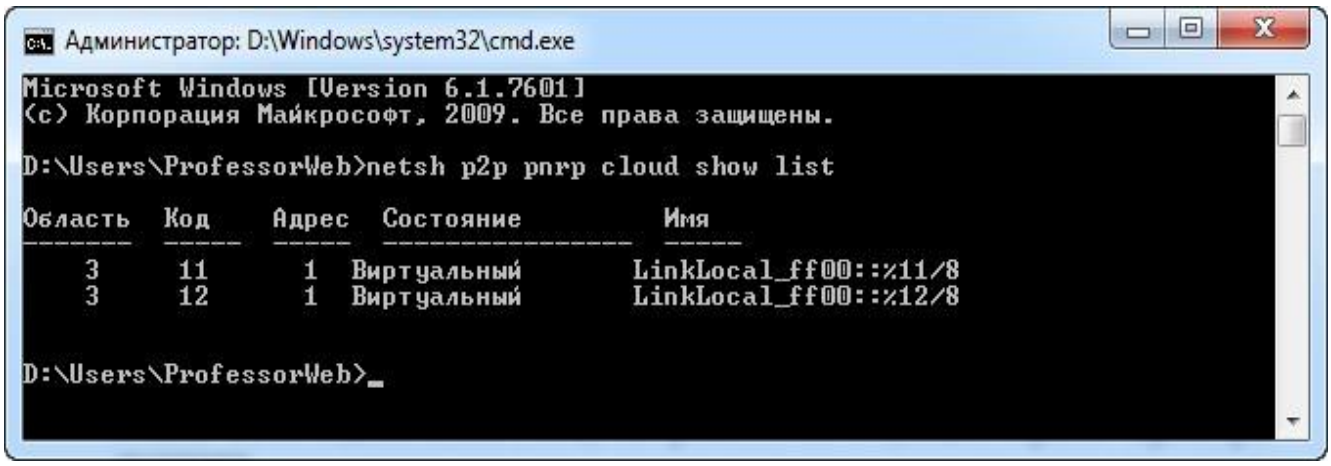

Рисунок 5 – Результат выполнения команды

В данном случае результат указывает, что доступно два облака локальных соединений (*Link local*).

Понять это можно как по значению в столбце *Name*(Имя), так и по значению в столбце *Scope*(Область), в котором для облаков локальных соединений всегда отображается значение 3, а для глобальных облаков – значение 1. Для подключения к глобальному облаку необходимо иметь глобальный адрес *IPv6*. Компьютер, на котором запускалась команда *netshc p2p*, таковым не обладал, потому для него оказалось доступным только локальное облако.

При наличии проблем с сетевой связью следует проверить, не препятствует ли брандмауэр передаче локального сетевого трафика через порты *UDP* 3540 или 1900. *UDP*-порт 3540 используется протоколом *PNRP*, а *UDP*-порт 1900 – протоколом *SSDP* (*Simple Service Discovery Protocol* – простой протокол обнаружения служб), который, в свою очередь, используется службой *PNRP* (а также устройствами *UPnP*).

Особенности *PNRP* в *Windows* 7

В *Windows* 7 протокол *PNRP* предусматривает использование нового компонента под названием *DRT* (*Distributed Routing Table* – таблица распределенной маршрутизации).

Этот компонент отвечает за определение структуры используемых *PNRP* ключей, роль которой в реализации по умолчанию исполняет описанный выше идентификатор *PNRP*. С помощью *API*-интерфейса *DRT* можно определить

альтернативную схему ключей при условии, что они представляют собой 256-битные целочисленные значения (подобно идентификаторам *PNRP ID*).

Это означает возможность использования любой схемы, но при этом, разумеется, нужно самостоятельно заботиться о генерации и защите ключей. За счет применения этого компонента можно создавать совершенно новые топологии облаков, выходящие за рамки действия *PNRP*.

В *Windows* 7 также появилась новая опция для подключения к другим пользователям в приложении *Remote Assistance*, которая называется *Easy Connect*. Эта опция использует *PNRP* для обнаружения пользователей, к которым необходимо подключиться. После создания сеанса с помощью *Easy Connect* или других средств (например, отправки приглашения по электронной почте) пользователи могут открыть общий доступ к своим рабочим столам и оказывать помощь друг другу через интерфейс *Remote Assistance*.

Служба *People Near Me*

Служба *PNRP*, как было показано в предыдущем разделе, применяется для определения местонахождения равноправных участников сети *P2P*. Очевидно, что она является важной действующей технологией при продумывании процесса обнаружения, подключения и взаимодействия в приложении *P2P*, но сама по себе она, ни одного из этих этапов полностью не реализует.

В реализации этапа обнаружения ей помогает служба *People Near Me* (Соседние пользователи), которая позволяет находить равноправных участников, которые зарегистрировались в локальной сети (т.е. подключены к тому же локальному облаку, что и данный компьютер).

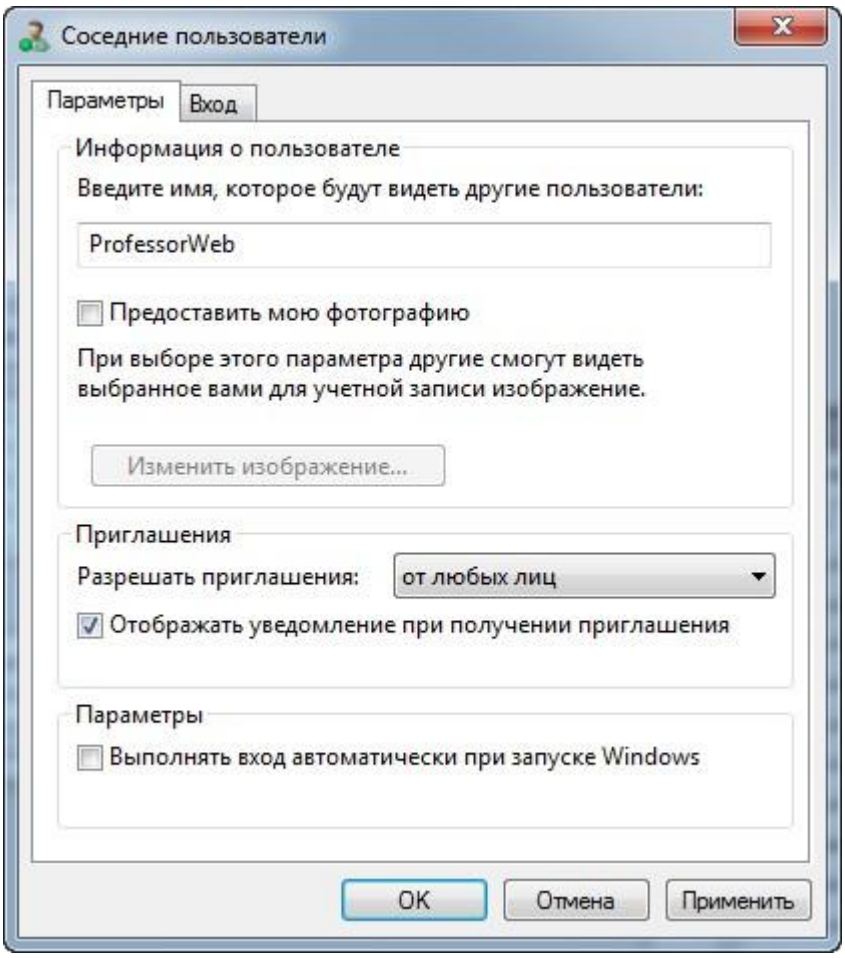

Рисунок 6 – Окно конфигурирования службы «Соседние пользователи»

Эта служба наверняка уже попадалась на глаза, поскольку она является встроенным компонентом *Windows Vista* и *Windows* 7, а также используется в приложении *Windows Meeting Space*, которое позволяет открывать равноправным участникам общий доступ к различным приложениям. Для конфигурирования этой службы применяется элемент *Change People Near Me* (Изменить соседних пользователей) панели управления, быстро перейти к которому можно путем ввода следующей команды (*cmd*):

```
collab.cpl
```

Этот элемент позволяет получить доступ к диалоговому окну, показанному на рисунке 6. После регистрации эта служба становится доступной в любом приложении, которое построено с учетом возможности ее использования:

На момент написания этих строк служба *PNM* было доступна только в семействе операционных систем *Windows Vista* и *Windows* 7.

Создание приложений *P2P*

Теперь, когда известно, что собой представляет технология *P2P*, и какие технологии доступны разработчикам приложений *.NET* для реализации приложений *P2P*, пришла пора посмотреть, как создавать такие приложения. Из приведенного ранее материала понятно, что в любом приложении *P2P* должен применяться протокол *PNRP* для публикации, распространения и преобразования имен равноправных участников. Поэтому в первую очередь здесь будет показано, как достичь этого в *.NET*, а затем – каким образом использовать службу *PNM* в качестве каркаса для приложения *P2P*. Последнее может быть выгодно, поскольку в случае применения *PNM* реализовать собственные механизмы обнаружения не понадобится.

Перед изучением этих тем сначала необходимо ознакомится с классами, которые предлагаются в следующих пространствах имен: *System.Net.PeerToPeer* и *System.Net.PeerToPeer.Collaboration*. Для работы с этими классами нужно обязательно ссылаться на сборку *System.Net.dll*.

Классы в пространстве имен *System.Net.PeerToPeer* инкапсулируют *API*-интерфейс для *PNRP* и позволяют взаимодействовать со службой *PNRP*. Их можно применять для решения двух основных задач: регистрация имен равноправных участников и преобразование имен равноправных участников.

Регистрация имен равноправных участников.

Для регистрации имени равноправного участника потребуется выполнить перечисленные ниже шаги:

1. Создайте защищенное или незащищенное имя равноправного участника с определенным классификатором.

2. Настройте процесс регистрации этого имени, предоставив следующие сведения:

✓ Номер порта *TCP*.

✓ Облако или облака, в которых должно быть зарегистрировано имя участника (если не указано, *PNRP* будет регистрировать имя равноправного участника во всех доступных облаках).

✓ Комментарий длиной до 39 символов.

✓ Дополнительные данные объемом до 4096 байт.

✓ Информация о том, должны ли для имени равноправного участники автоматически генерироваться конечные точки (поведение по умолчанию, при котором конечные точки автоматически генерируются на основе *IP*-адреса или адресов равноправного члена и номера порта, если таковой указан).

✓ Коллекция конечных точек.

3. Зарегистрируйте имя равноправного участника в локальной службе *PNRP*.

После выполнения третьего шага имя равноправного члена становится доступным для всех соседей в выбранном облаке (или облаках). Регистрация равноправного члена продолжается то тех пор, пока не будет прекращена явным образом, или до тех пор, пока не будет завершен процесс, зарегистрировавший имя равноправного участника.

Для создания имени равноправного участника применяется класс *PeerName*. Экземпляр этого класса создается из строкового представления идентификатора *P2P ID* в форме `авторитетный_источник.классификатор`, либо из строки классификатора и типа *PeerNameType*. Можно использовать как тип *PeerNameType.Secured*, так и тип *PeerNameType.Unsecured*. Например:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Secured);
```

Поскольку в незащищенном имени равноправного участника значением авторитетного источника является 0, следующие строки кода эквивалентны:

```
PeerName pn=new PeerName("Peer classifier", PeerNameType.Unsecured);
PeerName pn = new PeerName("0.Peer classifier");
```

После создания экземпляр *PeerName*, можно использовать вместе с номером порта для инициализации объекта *PeerNameRegistration*:

```
PeerNameRegistration pnr = new PeerNameRegistration(pn, 8080);
```

В качестве альтернативного варианта, можно установить для объекта *PeerNameRegistration*, создаваемого с использованием его параметра по умолчанию, свойство *PeerName* и (необязательно) свойство *Port*. Кроме того, желаемый экземпляр *Cloud* можно передать либо в третьем параметре

конструктору *PeerNameRegistration*, либо указать с помощью свойства *Cloud*. Экземпляр *Cloud* можно получить либо из имени облака, либо с применением одного из следующих статических членов класса *Cloud*:

Cloud.Global — Это статическое свойство позволяет получить ссылку на глобальное облако, которое в зависимости от конфигурации политики равноправных участников может быть и приватным глобальным облаком

Cloud.AllLinkLocal — Это статическое поле позволяет получить облако, содержащее все облака локальных соединений, доступные для данного равноправного участника

Cloud.Available — Это статическое поле позволяет получить облако, содержащее все облака, доступные для данного равноправного участника, т.е. локальные и глобальные (если таковые имеются)

После создания экземпляра *PeerNameRegistration* можно при желании установить его свойства *Comment* и *Data*. При этом следует помнить об ограничениях этих свойств. При попытке установить для свойства *Comment* строку длиной более 39 символов *Unicode*, будет сгенерировано исключение *PeerToPeerException*, а при попытке установить для свойства *Data* байтовый массив размером более 4096 байт – исключение *ArgumentOutOfRangeException*.

Можно также с помощью свойства *EndPointCollection* добавить конечные точки. Это свойство представляет собой коллекцию типа *System.Net.IPEndPointCollectionc* объектами *System.Net.IPEndPoint*. В случае применения свойства *EndPointCollection* может понадобиться установить свойство *UseAutoEndPointSelection* в *false*, чтобы предотвратить автоматическую генерацию конечных точек.

После того как все готово к регистрации имени равноправного участника, можно вызвать метод *PeerNameRegistration.Start()*. Для удаления записи о регистрации имени равноправного члена из службы *PNRP* служит метод *PeerNameRegistration.Stop()*.

Ниже приведен пример регистрации защищенного имени равноправного участника вместе с комментарием:

```
PeerName pn = new PeerName("Peer classifier", PeerNameType.Unsecured);
PeerNameRegistration pnr = new PeerNameRegistration(pn, 8080);
pnr.Comment = "Комментарий";
```

```
pnr.Start();
```

Преобразование имен равноправных участников

Для преобразования имени равноправного участника должны быть выполнены следующие шаги:

✓ Сгенерируйте имя равноправного члена на основе известного идентификатора *P2P*, либо идентификатора *P2P ID*, полученного посредством операции обнаружения.

✓ Воспользуйтесь распознавателем (*resolver*) для преобразования имени равноправного участника и получения коллекции записей с именами равноправных членов. Распознаватель можно ограничить отдельным облаком и/или максимальным количеством возвращаемых результатов.

✓ Из любой полученной записи с именем равноправного участника извлеките само имя, а также необходимую информацию о конечной точке, комментарии и дополнительные данные, если таковые присутствуют.

Подобно процессу регистрации имени равноправного участника, этот процесс начинается с создания объекта *PeerName*. Отличие состоит в том, что здесь используется имя равноправного участника, которое уже было зарегистрировано одним или более удаленными участниками. Простейшим способом для получения списка активных членов в локальном облаке является регистрация незащищенного имени для каждого равноправного участника с одним и тем же классификатором и применение этого имени на стадии преобразования.

Однако для глобальных облаков пользоваться такой стратегией не рекомендуется, поскольку незащищенные имена могут быть легко подделаны. Для преобразования имен равноправных участников используется класс *PeerNameResolver*. Получив в распоряжение экземпляр этого класса, можно выбрать, как должно выполняться преобразование – синхронно с применением метода *Resolve()* или же асинхронно с помощью метода *ResolveAsync()*.

Метод *Resolve()* можно вызывать с указанием как лишь одного параметра *PeerName*, так и дополнительно экземпляра *Cloud*, в котором должно

производиться преобразование, максимального количества возвращаемых записей с именами равноправных участников в виде целого числа, или того и другого вместе. Этот метод возвращает экземпляр *PeerNameRecordCollection*, представляющий собой коллекцию объектов *PeerNameRecord*.

Например, ниже показан пример преобразования незащищенного имени равноправного участника во всех локальных облаках с возвратом максимум 5 результатов:

```
PeerName pn = new PeerName("0.Peer classifier");
PeerNameResolver pnres = new PeerNameResolver();
PeerNameRecordCollection pnrc=pnres.Resolve(pn, Cloud.AllLinkLocal, 5);
```

Метод *ResolveAsync()* предусматривает использование стандартной схемы вызова асинхронного метода, а именно – передачу уникального объекта *userState*, после чего ожидание поступления событий *ResolveProgressChanged*, свидетельствующих об обнаружении равноправных участников и выполнении преобразования, и события *ResolveCompleted*, свидетельствующего об окончании процесса обнаружения участников и преобразования.

Метод *ResolveAsyncCancel()* позволяет отменить любой незавершенный асинхронный запрос. В обработчиках события *ResolveProgressChanged* используется параметр аргументов события *ResolveProgressChangedEventArgs*, унаследованный от стандартного класса *System.ComponentModel.ProgressChangedEventArgs*.

Свойство *PeerNameRecord* объекта аргумента события, получаемого в обработчике событий, можно применять для получения ссылки на запись с именем равноправного участника, которая была обнаружена.

Аналогичным образом для события *ResolveCompleted* требуется обработчик, принимающий параметр типа *ResolveCompletedEventArgs*, который наследуется от *AsyncCompletedEventArgs*. Этот тип имеет параметр *PeerNameRecordCollection*, который можно использовать для получения полного списка записей с именами равноправных членов, которые были обнаружены.

В следующем коде показан пример реализации обработчиков для этих событий:

```
private pnres_ResolveProgressChanged(object sender,
                                     ResolveProgressChangedEventArgs e)
{
  //Использование e.ProgressPercentage (унаследованного от базовых
  //аргументов события)
  //Обработка PeerNameRecord из e.PeerNameRecord
}
private pnres_ResolveCompleted(object sender,
                                     ResolveCompletedEventArgs e)
{
  //Проверка наличия e.IsCancelled и e.Error (унаследованных
  //от базовых аргументов события)
  //Обработка PeerNameRecordCollection из e.PeerNameRecordCollection
}
```

После получения одного или более объектов *PeerNameRecord* можно переходить к их обработке. Этот класс *PeerNameRecord* предоставляет в распоряжение свойства *Comment* и *Data* для изучения комментариев и дополнительных данных, которые были добавлены при регистрации имени равноправного участника (если были), свойство *PeerName* для извлечения объекта *PeerName* и записи с именем участника и, что наиболее важно, свойство *EndPointCollection*.

Как и в случае с *PeerNameRegistration*, здесь это свойство тоже представляет собой коллекцию типа *System.Net.IPEndPointCollection*, содержащую объекты *System.Net.IPEndPoint*. Эти объекты можно использовать для подключения к предоставляемым равноправным участником конечным точкам любым желаемым образом.

Безопасный доступ кода в *System.Net.PeerToPeer*

Пространство имен *System.Net.PeerToPeer* также включает два следующих класса, которые можно использовать вместе с моделью

CAS (*Code Access Security* – безопасный доступ кода):

PnrpPermission, унаследованный от класса *CodeAccessPermission* и *Pnrp-PermissionAttribute*, унаследованный от класса *CodeAccessSecurityAttribute*.

Эти классы удобно применять для обеспечения возможности настройки доступа *PNRP* с помощью полномочий обычным для *CAS* образом.

Пример приложения *P2P*

Давайте рассмотрим пример приложения *P2P*, использующего графический интерфейс *WPF*, в котором для конечной точки равноправного участника используется служба *WCF*.

Конфигурационные настройки этого приложения содержатся в конфигурационном файле *App.config* и указывают, как должно выглядеть имя равноправного участника и какой порт должен прослушиваться:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="username" value="Вася Пупкин" />
    <add key="port" value="4590" />
  </appSettings>
</configuration>
```

После сборки этого приложения можно приступить к его тестированию, либо скопировав его на другие компьютеры в локальной сети и запустив все его экземпляры, либо запустив множество его экземпляров на одном компьютере. В последнем варианте нужно будет поменять используемый для каждого экземпляра порт, внеся соответствующие изменения в отдельные конфигурационные файлы (понадобится скопировать содержимое каталога *Debug* на свой локальный компьютер и по очереди отредактировать каждый конфигурационный файл).

В обоих вариантах результаты будут выглядеть яснее, если изменить также имя пользователя в каждом экземпляре.

После запуска всех экземпляров приложения можно щелкнуть на кнопке *Refresh* (Обновить), чтобы получить список доступных равноправных участников асинхронным образом. Обнаружив в этом списке нужного участника, можно с помощью щелчка на кнопке *Message* отправить ему стандартное сообщение.

На рисунке 7 показан пример того, как приложение выглядит в действии при запуске четырех его экземпляров на одной машине. На рисунке 7 видно, что

один равноправный участник только что отправил сообщение другому равноправному участнику, и оно отобразилось в диалоговом окне:

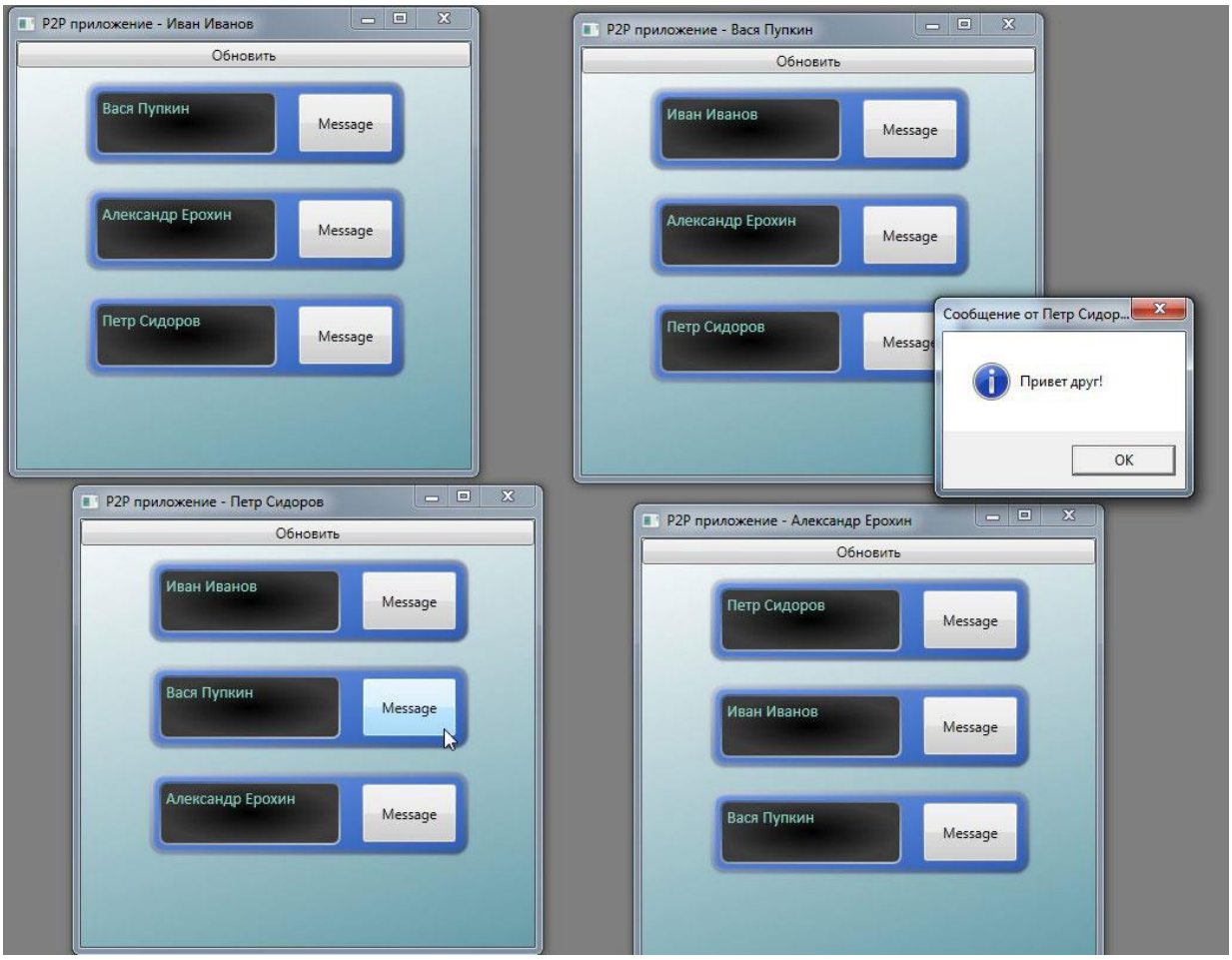

Рисунок 7 – Пример приложения при запуске четырех его экземпляров на одной машине

Итак, создайте новый проект приложения *WPF* по имени *P2P*. Изначально нужно будет добавить в проект ссылки на сборки *System.Configuration.dll*, *System.Net.dll*, *System.Runtime.Serialization.dll*, *System.ServiceModel.dll*, а также указать пространства имен, которые мы будем использовать:

```
using System.Net;
using System.Net.PeerToPeer;
using System.ServiceModel;
using System.Configuration;
```

Добавьте в проект *XML*-файл *App.config*, который описан выше, а также следующие файлы классов (предоставляют информацию о пире, создают *WCF*-контракт):

```
//Файл PeerEntry.cs
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Net.PeerToPeer;

namespace P2P
{
  class PeerEntry
  {
    public PeerName PeerName { get; set; }
    public IP2PService ServiceProxy { get; set; }
    public string DisplayString { get; set; }
    public bool ButtonsEnabled { get; set; }
  }
}
//Файл P2PService.cs
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace P2P
{
 [ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
 public class P2PService : IP2PService
 {
  private MainWindow hostReference;
  private string username;

  public P2PService(MainWindow hostReference, string username)
  {
   this.hostReference=hostReference;
   this.username = username;
  }

  public string GetName()
  {
   return username;
  }

  public void SendMessage(string message, string from)
  {
   hostReference.DisplayMessage(message, from);
  }
 }
}
//Файл IP2PService.cs
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;
using System.Runtime.Serialization;
using System.ServiceModel;

namespace P2P
{
 [ServiceContract]
 public interface IP2PService
 {
  [OperationContract]
  string GetName();

  void SendMessage(string message, string from);
 }
}
```

Большая часть работы в этом приложении происходит в обработчике события *Window_Loaded()* окна *MainWindow*. Ниже показана *XAML*-разметка окна и исходный код приложения:

```
<Window x:Class="P2P.MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:P2P="clr-namespace:P2P"
  Title="MainWindow" Height="400" Width="400" Loaded="Window_Loaded"
                                              Closing="Window_Closing">
<Window.Resources>
  <LinearGradientBrush x:Key="bevelBrush" EndPoint="0.369,-1.362"
                                              StartPoint="0.631,2.362">
     <GradientStop Color="#FF001E56" Offset="0"/>
     <GradientStop Color="#FFFFFFFF" Offset="1"/>
  </LinearGradientBrush>
  <DataTemplate x:Key="PeerEntryDataTemplate">
     <Grid>
        <Grid.ColumnDefinitions>
           <ColumnDefinition Width="Auto" />
           <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
           <RowDefinition />
        </Grid.RowDefinitions>
        <Rectangle  RadiusX="10"  RadiusY="10"  Grid.ColumnSpan="2"
        Stroke="{DynamicResource bevelBrush}" StrokeThickness="4" >
           <Rectangle.Fill>
              <LinearGradientBrush EndPoint="0.369,-1.362"
                                              StartPoint="0.631,2.362">
                  <GradientStop Color="#FF1346A6" Offset="0"/>
                  <GradientStop Color="#FF85ADF6" Offset="1"/>
              </LinearGradientBrush>
           </Rectangle.Fill>
           <Rectangle.BitmapEffect>
              <BlurBitmapEffect Radius="3"/>
           </Rectangle.BitmapEffect>
```

```
        </Rectangle>
        <TextBlock  Margin="10"  Text="{Binding  Path=DisplayString}"
        Padding="4"  TextWrapping="Wrap"  Width="150"  Opacity="0.995"
        FontFamily="Calibri" FontSize="14" Foreground="#FF8ED1C3" >
          <TextBlock.Background>
             <RadialGradientBrush>
                <GradientStop Color="#FF000000" Offset="0"/>
                <GradientStop Color="#FF3C3C3C" Offset="1"/>
             </RadialGradientBrush>
          </TextBlock.Background>
        </TextBlock>
        <Rectangle RadiusX="6" RadiusY="6" Margin="8" Fill="{x:Null}"
        StrokeThickness="2" >
          <Rectangle.Stroke>
             <LinearGradientBrush              EndPoint="0.631,2.362"
             StartPoint="0.369,-1.362">
                <GradientStop Color="#FF001E56" Offset="0"/>
                <GradientStop Color="#FFFFFFFF" Offset="1"/>
             </LinearGradientBrush>
          </Rectangle.Stroke>
        </Rectangle>
        <StackPanel Grid.Column="1">
           <Button     Name="MessageButton"     Margin="10,10,10,10"
           Height="50" IsEnabled="{Binding Path=ButtonsEnabled}" Con-
           tent="Message"    BorderBrush="{DynamicResource    bevel-
           Brush}"/>
        </StackPanel>
      </Grid>
  </DataTemplate>
</Window.Resources>

<Window.Background>
  <LinearGradientBrush EndPoint="0.444,-0.183" StartPoint="0.778,1.12">
     <GradientStop Color="#FF6199A5" Offset="0"/>
     <GradientStop Color="#FFFFFFFF" Offset="1"/>
  </LinearGradientBrush>
</Window.Background>

<StackPanel>
  <Button Name="RefreshButton" Click="RefreshButton_Click">Обновить<
                                                          /Button>
  <ListBox Name="PeerList" ItemTemplate="{DynamicResource PeerEntry-
  DataTemplate}"       ButtonBase.Click="PeerList_Click"       Back-
  ground="{x:Null}" BorderBrush="{x:Null}">
     <ListBox.ItemContainerStyle>
         <Style TargetType="ListBoxItem">
            <Setter Property="Margin" Value="10" />
            <Setter Property="HorizontalAlignment" Value="Center" />
         </Style>
     </ListBox.ItemContainerStyle>
     <P2P:PeerEntry DisplayString="Обновите, чтобы увидеть пиров."
                                          ButtonsEnabled="False" />
  </ListBox>
```

```
</StackPanel>
</Window>


public partial class MainWindow : Window
{
  private P2PService localService;
  private string serviceUrl;
  private ServiceHost host;
  private PeerName peerName;
  private PeerNameRegistration peerNameRegistration;

  public MainWindow()
  {
    InitializeComponent();
  }

private void Window_Loaded(object sender, RoutedEventArgs e)
{
//Получение конфигурационной информации из app.config
string port = ConfigurationManager.AppSettings["port"];
string username = ConfigurationManager.AppSettings["username"];
string machineName = Environment.MachineName;
string serviceUrl = null;

//Установка заголовка окна
this.Title = string.Format("P2P приложение - {0}", username);

//Получение URL-адреса службы с использованием адресаIPv4
//и порта из конфигурационного файла
foreach (IPAddress address in Dns.GetHostAddresses
                                          (Dns.GetHostName()))
{
  if (address.AddressFamily==System.Net.Sockets.AddressFamily.InterNetwork)
    {
      serviceUrl = string.Format("net.tcp://{0}:{1}/P2PService", address, port);
      break;
    }
}

//Выполнение проверки, не является ли адрес null
if (serviceUrl == null)
{
   //Отображение ошибки и завершение работы приложения
   MessageBox.Show(this, "Не удается определить адрес конечной точки
                       WCF.", "Networking Error", MessageBoxButton.OK,
                       MessageBoxImage.Stop);
   Application.Current.Shutdown();
}
//Регистрация и запуск службы WCF
localService = new P2PService(this, username);
host = new ServiceHost(localService, new Uri(serviceUrl));
NetTcpBinding binding = new NetTcpBinding();
```

```
binding.Security.Mode = SecurityMode.None;
host.AddServiceEndpoint(typeof(IP2PService), binding, serviceUrl);
try
{
  host.Open();
}
catch (AddressAlreadyInUseException)
{
   //Отображение ошибки и завершение работы приложения
   MessageBox.Show(this, "Не удается начать прослушивание,
                     порт занят.", "WCF Error", MessageBoxButton.OK,
                     MessageBoxImage.Stop);
   Application.Current.Shutdown();
}

//Создание имени равноправного участника (пира)
peerName = new PeerName("P2P Sample", PeerNameType.Unsecured);

//Подготовка процесса регистрации имени равноправного участника в локальном облаке
peerNameRegistration=new PeerNameRegistration(peerName, int.Parse(port));
peerNameRegistration.Cloud = Cloud.AllLinkLocal;

//Запуск процесса регистрации
peerNameRegistration.Start();
}

private void Window_Closing(object sender, System.ComponentModel.
                                                    CancelEventArgs e)
{
   //Остановка регистрации
   peerNameRegistration.Stop();

   //Остановка WCF-сервиса
   host.Close();
}

private void RefreshButton_Click(object sender, RoutedEventArgs e)
{
   //Создание распознавателя и добавление обработчиков событий
   PeerNameResolver resolver = new PeerNameResolver();
   resolver.ResolveProgressChanged +=
            new EventHandler<ResolveProgressChangedEventArgs>
            (resolver_ResolveProgressChanged);
   resolver.ResolveCompleted +=
            new EventHandler<ResolveCompletedEventArgs>
            (resolver_ResolveCompleted);

   //Подготовка к добавлению новых пиров
   PeerList.Items.Clear();
   RefreshButton.IsEnabled = false;

   //Преобразование незащищенных имен пиров асинхронным образом
   resolver.ResolveAsync(new PeerName("0.P2P Sample"), 1);
```

```
}

void resolver_ResolveCompleted(object sender, ResolveCompletedEventArgs e)
{
   //Сообщение об ошибке, если в облаке не найдены пиры
   if (PeerList.Items.Count == 0)
   {
     PeerList.Items.Add(
       new PeerEntry
       {
          DisplayString = "Пиры не найдены.",
          ButtonsEnabled = false
       });
   }
   //Повторно включаем кнопку "обновить"
   RefreshButton.IsEnabled = true;
}

void resolver_ResolveProgressChanged(object sender,
                                     ResolveProgressChangedEventArgs e)
{
   PeerNameRecord peer = e.PeerNameRecord;

   foreach (IPEndPoint ep in peer.EndPointCollection)
   {
      if (ep.Address.AddressFamily ==
                    System.Net.Sockets.AddressFamily.InterNetwork)
      {
         try
         {
            string endpointUrl = string.Format
              ("net.tcp://{0}:{1}/P2PService", ep.Address, ep.Port);
            NetTcpBinding binding = new NetTcpBinding();
            binding.Security.Mode = SecurityMode.None;
            IP2PService serviceProxy =
                      ChannelFactory<IP2PService>.CreateChannel
                       (binding, new EndpointAddress(endpointUrl));
            PeerList.Items.Add(
               new PeerEntry
               {
                  PeerName = peer.PeerName,
                  ServiceProxy = serviceProxy,
                  DisplayString = serviceProxy.GetName(),
                  ButtonsEnabled = true
               });
         }
         catch (EndpointNotFoundException)
         {
            PeerList.Items.Add(
               new PeerEntry
               {
                  PeerName = peer.PeerName,
                  DisplayString = "Неизвестный пир",
```

```
                    ButtonsEnabled = false
                });
            }
        }
    }
}

private void PeerList_Click(object sender, RoutedEventArgs e)
{
  //Убедимся, что пользователь щелкнул по кнопке с именем MessageButton
  if (((Button)e.OriginalSource).Name == "MessageButton")
  {
     //Получение пира и прокси, для отправки сообщения
     PeerEntry peerEntry=((Button)e.OriginalSource).DataContext as PeerEntry;
     if (peerEntry != null && peerEntry.ServiceProxy != null)
     {
       try
       {
         peerEntry.ServiceProxy.SendMessage("Привет друг!",
                    ConfigurationManager.AppSettings["username"]);
       }
       catch (CommunicationException)
       {

       }
     }
  }
}

internal void DisplayMessage(string message, string from)
{
   //Показать полученное сообщение (вызывается из службы WCF)
   MessageBox.Show(this, message, string.Format("Сообщение от {0}",
        from), MessageBoxButton.OK, MessageBoxImage.Information);
}
}
```

В обработчике события *Window_Loaded()* сначала производится загрузка конфигурационной информации и установка в заголовке окна имени пользователя.

Далее с использованием адреса хоста равноправного участника и сконфигурированного порта определяется конечная точка, в которой должна предоставляться служба *WCF*. Привязкой этой службы будет *NetTcpBinding*, поэтому в *URL*-адресе конечной точки применяется протокол *net.tcp*.

Затем производится проверка полученного *URL*-адреса конечной точки, после чего выполняется регистрация и запуск службы *WCF*. Обратите внимание

на использование одиночного (*singleton*) экземпляра класса службы для налаживания простых коммуникаций между приложением-хостом и службой (для отправки и получения сообщений). Кроме того, для простоты режим безопасности в конфигурации привязки отключен.

После щелчка на кнопке Обновить в обработчике события *RefreshButton_Click()* вызывается метод *PeerNameResolver.ResolveAsync()* для обнаружения соседних равноправных участников асинхронным образом.

Остальная часть кода отвечает за отображение и взаимодействие с соседними равноправными участниками; ее можно изучить самостоятельно.

Предоставление конечных точек *WCF* через облака *P2P* является замечательным способом для обеспечения возможности установки местонахождения служб внутри предприятия, а также налаживания связи между равноправными участниками сети подобно тому, как было сделано в рассмотренном примере.

Пространство имен *System.Net.PeerToPeer.Collaboration*

Классы в пространстве имен *System.Net.PeerToPeer.Collaboration* предоставляют каркас, который можно использовать для создания приложений, предусматривающих работу со службой *People Neat Me* и *API*-интерфейсом совместной работы посредством *P2P* (*P2P Collaboration API*). Как упоминалось ранее, это было возможно только в ОС *Windows Vista* или *Windows* 7.

Классы, определенные в этом пространстве имен, можно использовать для взаимодействия с соседними равноправными участниками и приложениями, включая выполнение следующих функций:

✓ осуществление входа (sign in) и выхода (sign out);

✓ обнаружение других равноправных участников;

✓ управление контактами и выявление присутствия других равноправных участников в сети.

Классы из этого пространства имен можно также применять для приглашения других пользователей присоединиться к работе в каком-то приложении или для обеспечения возможности обмена данными между

пользователями и приложениями. Однако для того чтобы делать это, необходимо создавать собственные приложения, поддерживающие *PNM*.

Осуществление входа и выхода

Одним из наиболее важных классов в пространстве имен *System.Net.PeerToPeer.Collaboration* является *PeerCollaboration*. Это статический класс с множеством статических методов, которые можно применять для самых различных целей, как будет показано в этом и следующем разделах. В частности, для осуществления входа и выхода из службы *People Near Me* предназначены его методы *SignIn()* и *SignOut()*. Оба метода принимают единственный параметр типа *PeerScope*, который может иметь одно из перечисленных ниже значений:

PeerScope.None	В случае использования этого значения методы *SignIn()* и *SignOut()* не будут иметь никакого эффекта
PeerScope.NearMe	Указание этого значения позволяет осуществлять вход и выход из локальных облаков
PeerScope.Internet	Применение этого значения позволяет осуществлять вход и выход из глобального облака (это может понадобится для установки связи с контактным лицом, которого в текущий момент нет в локальной подсети)
PeerScope.All	Указание этого значения позволяет осуществлять вход и выход из всех доступных облаков

При необходимости вызов *SignIn()* будет приводить к отображению диалогового окна конфигурирования *People Near Me*. После входа в службу можно для свойства *PeerCollaboration.LocalPresenceInfo* установить значение типа *PeerPresenceInfo*. Это сделает доступной стандартную функциональность мгновенного обмена сообщениями (*IM*), например, возможность установки состояния в "отошел".

Для свойства *PeerPresenceInfo.DescriptiveText* можно установить строку *Unicode* длиной до 255 символов, а для свойства *PeerPresenceInfo.PresenceStatus* – значение перечисления *PeerPresenceStatus*. Значения этого перечисления описаны ниже:

Перечисление *PeerPresenceStatus*

Значение	Описание
PeerPresenceStatus.Away	Равноправного участника нет на месте

PeerPresenceStatus.BeRightBack	Равноправного участника нет на месте, но он скоро вернется
PeerPresenceStatus.Busy	Равноправный участник занят
PeerPresenceStatus.Idle	Равноправный участник неактивен
PeerPresenceStatus.Offline	Равноправного участника нет в сети
PeerPresenceStatus.Online	Равноправный участник находится в сети и доступен для связи
PeerPresenceStatus.OnThePhone	Равноправный участник разговаривает по телефону
PeerPresenceStatus.OutToLunch	Равноправного участника нет на месте, но он вернется после обеда

Обнаружение соседей

После подключения к локальному облаку можно получить список всех равноправных участников сети, которые находятся по соседству. Это делается с помощью метода *PeerCollaboration.GetPeersNearMe()*, который возвращает объект *PeerNearMeCollection*, содержащий в себе объекты *PeerNearMe*.

Свойство *Nickname* объекта *PeerName* позволяет получить имя обнаруженного соседа, свойство *IsOnline* – определить, находится ли он в текущий момент в сети, а (в случае низкоуровневых операций) свойство *PeerEndPoints* – определить имеющие к нему отношение конечные точки. С помощью свойства *PeerEndPoints* также можно выяснить статус объекта *PeerNearMe* в сети.

Для получения объекта *PeerPresenceInfo* необходимо передать методу *GetPresenceInfo()* информацию о конечной точке, как было описано в предыдущем разделе.

Управление контактами и определение присутствия соседей в сети

Контакты представляют собой способ, которым можно запоминать других равноправных участников сети. Обнаруженных посредством службы *People Near Me* пользователей сети можно добавить в свои контакты и после этого связываться с ними в любой момент, когда они будут появляться в сети. Связываться с контактами можно как через локальные, так и через глобальные облака (при условии наличия *IPv6*-подключения к Интернету).

Чтобы добавить обнаруженного пользователя в контакты, необходимо вызвать метод *PeerNearMe.AddToContactManager()*. При вызове этого метода можно закрепить за контактом отображаемое имя, псевдоним (*nickname*) и адрес электронной почты. Однако обычно управление контактами осуществляется с помощью класса *ContactManager*.

В любом случае при манипулировании контактами приходится иметь дело с объектами *PeerContact. Класс PeerContact*, как и *PeerNearMe*, унаследован от абстрактного базового класса *Peer*, но имеет больше свойств и методов, чем *PeerNearMe*.

Так, например, он включает в себя свойства *DisplayName* и *EmailAddress*, которые позволяют более подробно описывать обнаруженного *PNM* пользователя. Другое отличие между этими двумя классами связано с тем, что *PeerContact* имеет более явные отношения с классом *System.Net.PeerToPeer.PeerName*. С помощью свойства *PeerContact.PeerName* можно извлечь объект *PeerName*, после чего приступить к взаимодействию с любыми конечными точками, которые этот объект *PeerName* предоставляет, используя описанные ранее приемы.

Информация о локальном соседе также доступна в статическом свойстве *ContactManager.LocalContact*. Это позволит далее иметь дело со свойством *PeerContact* и деталями о локальном соседе.

Для добавления объектов *PeerNearMe* в локальный список контактов применяется либо метод *ContactManager.CreateContact()*, либо метод *CreateContactAsync()*, а объектов *PeerName* – метод *GetContact()*. Удаление контактов, представляемых объектами *PeerNearMe* или *PeerName*, производится с помощью метода *DeleteContact()*.

И, наконец, существуют события, которые можно обрабатывать в ответ на изменения в контактах. Например, событие *PresenceChanged* можно обрабатывать в ответ на изменения в информации о присутствии любого из известных *ContactManager* контактов.

Задания для самостоятельного выполнения

1. Реализовать пример приложения, который рассмотрен.

2. Усовершенствовать приложение, добавив дополнительные функции, по согласованию с преподавателем.

Варианты индивидуальных заданий к лабораторной работе.

ВАРИАНТ 1. Реализовать приложение для отправки файла по запросу.

ВАРИАНТ 2. Реализовать приложение для отправки текстовых сообщений по запросу.

ВАРИАНТ 3. Реализовать приложение для отправки изображений по запросу.

КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Структура сетевого приложения в пиринговых сетях.

2. Организация и методы для поиска соседних узлов.

3. Получение состояния соседних пиров.

4. Методы для добавления обнаруженного пользователя в контакты.

Список литературы

1. Вайсфельд, М. Объектно-ориентированное программирование / М. Вайсфельд. – М.; СПб.; Нижний Новгород: Питер, 2014. – 304с.

2. Иванова Т. И., Корпоративные сети связи. – М., Эко – Трендз, 2001г. – 284с.

3. Мельников Д. А., Информационные процессы в компьютерных сетях. Протоколы, стандарты, интерфейсы, модели. – М.: КУДИЦ – ОБРАЗ,1999г. – 256с.

4. Олифер, В. Г. Основы компьютерных сетей / В. Г.Олифер. – М.; СПб.; Нижний Новгород: Питер М, 2014. – 352 с. – (Учебное пособие).

5. Пятибратов А.П., Вычислительные системы, сети и телекоммуникации, Учебник 2-е издание, М., Финансы и статистика,2002г. – 512с.

6. Фейт, С. TCT/IP: Архитектура, протоколы, реализация (включая IP версии 6 и IP Security) / С. Фейт. – 2-е изд. – М.: Лори, 2014. – 424с.

7. .NET. Сетевое программирование / В. Кумар [и др.]. – [б. м.]: Лори, 2014. – 400с.

8. Интернет: http://www.osp.ru/nets/ – журнал «СЕТИ» издательства Открытые системы.

9. Интернет: http://www.citforum.ru/nets/ – раздел «СЕТЕВЫЕ ТЕХНОЛОГИИ» на веб-сайте ЦНИТ МГУ

ОГЛАВЛЕНИЕ

Учебное издание

ТЕХНИЧЕСКАЯ ИНФОРМАТИКА (СЕТЕВОЕ ПРОГРАММИРОВАНИЕ)
Методические указания к лабораторным работам

Издается в авторской редакции

Компьютерная верстка *Л.В. Савицкая*
ИЛ № *****. Сер. АЮ от **.**.**. Подписано в печать
Формат 60x90/16. Усл. печ. л.7,7

Опубликовано
на Образовательном портале ПГУ им. Т.Г. Шевченко moodle@spsu.ru