Programmer to Programmer ™

Professional .NET Network Programming

Andrew Krowczyk, Vinod Kumar, Nauman Laghari, Ajit Mungale,
Christian Nagel, Tim Parker, Srinivasa Sivakumar

Wrox technical support at: **support@wrox.com**
Updates and source code at: **www.wrox.com**
Peer discussion at: **p2p.wrox.com**

What You Need to Use This Book

A prerequisite to running the samples in this book is that you have a machine with the .NET Framework installed upon it. This means that you'll need to be running either:

- Windows 2000 Professional (or better)
- Windows XP

It is also recommended that you use a version of Visual Studio .NET with this book.

In this book, prior knowledge of network programming is not assumed, so basic and more advanced networking concepts are appropriately covered.

All the code examples in this book are in C#, and a working knowledge of the language is assumed.

Summary of Contents

7

Multicast Sockets

In 1994, you may recall, the Rolling Stones transmitted a live concert over the Internet for free. This was made possible due to multicasting, the same technology that makes it possible to watch astronauts in space, to hold meetings over the Internet, and more.

Unicasting would be inappropriate for these applications, as for events attended by thousands of clients, the load on the server and the network would be excessive. Multicasting means that the server only has to send messages just once, and they will be distributed to a whole group of clients. Only systems that are members of the group participate in the network transfers.

In the last few chapters, we discussed socket programming using connection-oriented and connectionless protocols. Chapter 6 showed how we can send broadcasts with the UDP protocol. In this chapter, the UDP protocol again rears its head, but now we are using multicasts.

Multicasts can be used for group communications over the Internet, where every node participating in the multicast has to join the group set up for the purpose. Routers can forward messages to all interested nodes.

In this chapter, we will create two Windows applications using multicasting features. With one application it will be possible to chat with multiple systems, where everyone is both a sender and a receiver. The second application – in the form of a picture show – demonstrates how large data packets can be sent to multiple clients without using a large percentage of the network bandwidth.

In particular, we will:

- Compare unicasts, broadcasts, and multicasts
- Examine the architecture of multicasting
- Implement multicast sockets with .NET
- Create a multicast chat application
- Create a multicast picture show application

Unicasts, Broadcasts, and Multicasts

The Internet Protocol supports three kinds of IP addresses:

- **Unicast** – unicast network packets are sent to a single destination
- **Broadcast** – broadcast datagrams are sent to all nodes in a subnetwork
- **Multicast** – multicast datagrams are sent to all nodes, possibly on different subnets, that belong to a group

The TCP protocol provides a connection-oriented communication where two systems communicate with each other; with this protocol, we can only send unicast messages. If multiple clients connect to a single server, all clients maintain a separate connection on the server. The server needs resources for each of these simultaneous connections, and must communicate individually with every client. Don't forget that the UDP protocol can also be used to send unicast messages, where, unlike TCP, connection-less communication is used, making it is faster than TCP, although without TCP's reliability.

> *Sending unicast messages with TCP is covered in Chapter 5; using UDP for unicast is discussed in Chapter 6.*

Broadcast addresses are identified by IP addresses where all bits of the host are set to 1. For instance, to send messages to all hosts in a subnet with a mask of `255.255.255.0` in a network with the address `192.168.0`, the broadcast address would be `192.168.0.255`. Any host with an IP address beginning `192.168.0` will then receive the broadcast messages. **Broadcasts** are *always* performed with connection-less communication using the UDP protocol. The server sends the data regardless of whether any client is listening. Performance reasons mean it wouldn't be possible to set up a separate connection to every client. Connection-less communication means that the server does not have to allocate resources for every single client – no matter how many clients are listening, the same server resources will be consumed.

Of course, there are disadvantages to the connection-less mechanism. For one, there is no guarantee that the data is received by anyone. If we wanted to add reliability, we would have to add a handshaking mechanism of our own at a higher level than UDP.

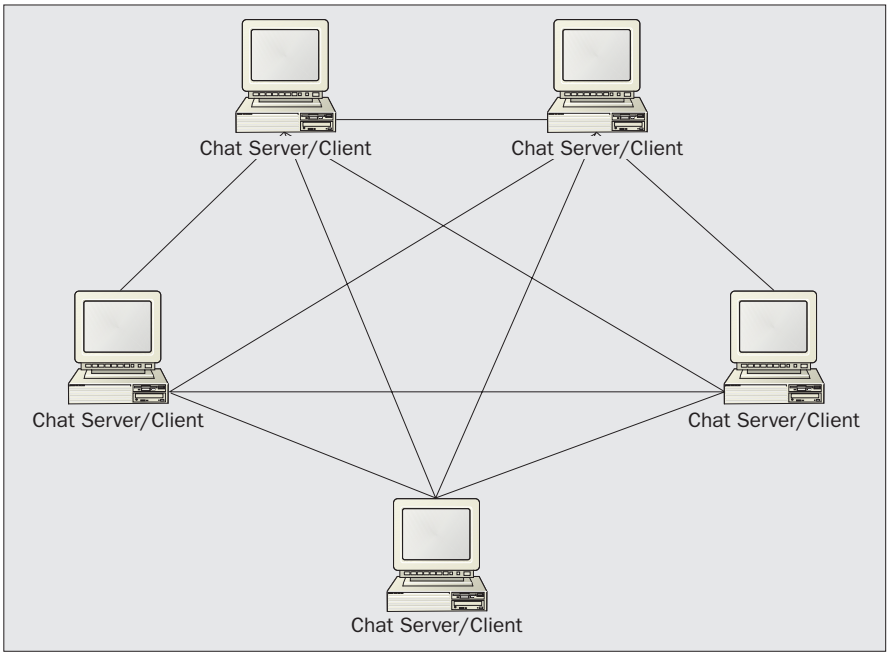
Broadcasts introduce a performance issue for every system on the destination subnet, because each system on that subnet has to check whether the receiving packet is of interest. A broadcast can be of interest to any system in the network, and it passes all the way up to the transport layer in the protocol stack of each system before its relevancy can be determined. There is another issue with broadcasts: they don't cross subnets. Routers don't let broadcasts cross them – we would soon reach network saturation if routers forwarded broadcasts, so this is desired behavior. Thus, broadcasts can be only used inside a particular subnet.

Broadcast communication is useful if multiple nodes in the same subnet should get information simultaneously. NTP (Network Time Protocol) is an example where broadcasts are useful.

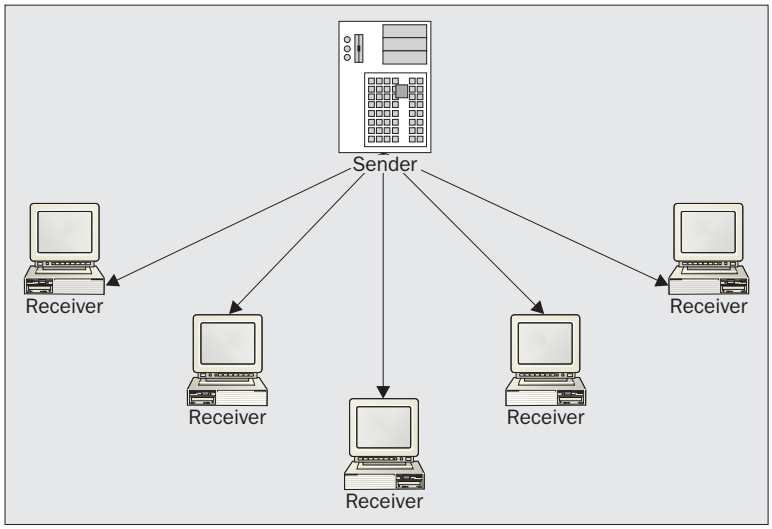
Multicast addresses are identified by IP class D addresses (in the range `224.0.0.0` to `239.255.255.255`). Multicast packets can pass across different networks through routers, so it is possible to use multicasts in an Internet scenario as long as your network provider supports multicasting. Hosts that want to receive particular multicast messages must register their interest using IGMP (Internet Group Management Protocol). Multicast messages are not then sent to networks where no host has joined the multicast group. Class D IP addresses are used for multicast groups, to differentiate them from normal host addresses, allowing nodes to easily detect if a message is of interest.

Application Models with Multicasts

There are many types of application where multicasts are of good use. One such scenario is when every system in a group wants to send data to every other system in the group (many-to-many). Multicasting means that each system doesn't need to create a connection to every other system, and a multicast address can be used instead. A peer-to-peer chat application, as seen in the picture below, would benefit from such behavior. The chat sender sends a message to every node of the group by sending a single message to the network:



Another scenario where multicasts play an important role is if one system wants to send data to a group of systems (one-to-many). This can be useful for sending audio, video, or other large data types. The server only sends the data once, to the multicast address, and a large number of systems can listen. The Rolling Stones concert in November 1994 was the first time audio and video of a live rock concert was transmitted over the Internet using multicast. This was a big success, and it demonstrated the usefulness of multicasting. The same technology is used in a local network to install applications on hundreds of PCs simultaneously without the servers having to send a big installation package to every client system separately:



Architecture of Multicast Sockets

Multicast messages are sent using the UDP protocol to a group of systems identified by a class D subnet address. Certain class D address ranges are reserved for specific uses, as we will see soon.

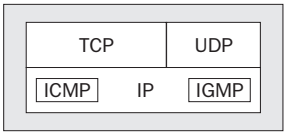
In addition to UDP, the **Internet Group Management Protocol** (IGMP) is used to register clients that want to receive messages for a specific group. This protocol is built into the IP module, and allows clients to leave a group as well as join.

In this section, we'll cover foundation issues and other important factors of multicasting:

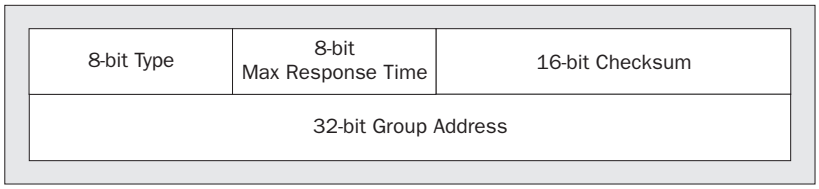
- The IGMP protocol
- Multicast addresses
- Routing
- Scoping
- Scalability
- Reliability
- Security

The IGMP Protocol

IGMP is used by IP hosts to report group memberships to any immediately neighboring routers that are multicast enabled. Similarly to the ICMP protocols, IGMP is implemented in the IP module as the picture below shows. IGMP messages are encapsulated in IP datagrams with the IP protocol number 2. In Chapter 1, we saw the protocol number listed in the IP header, where 2 denotes IGMP, 1 is for ICMP, 6 for TCP, and 17 for UDP.



An IGMP v2 message consists of 64 bits, and contains the type of the message, a maximum response time (used only for membership queries), a checksum, and the group address:



IGMP v2 is defined in RFC 2236 (http://www.ietf.org/rfc/rfc2236.txt).

The message types used for communication between a host and a router are defined by the first 8 bits of IGMP v2 message headers, and are described in the following table:

Hex Value	Message	Description
`0x11`	Membership Query	These are used by the router to see whether any group members exist. Two types of membership query can be differentiated by the group address in the 32-bit group address field. A **general query** has a group address in the IGMP header of all zeros, and asks which groups have members on an attached network. A **group-specific query** returns information on whether a particular group has members.
`0x16`	Version 2 Membership Report	When a host **joins a multicast group**, a membership report is sent to the router to inform the router that a system on the network is listening to multicast messages.

Table continued on following page

Hex Value	Message	Description
0x17	Leave Group	The last host of a multicast group inside a subnet must send a **leave group** message to all routers (224.0.0.2) when it leaves a group. A host may remember the hosts of the multicast group (received in membership reports in response to membership queries) so that it knows when it is the last one in the group, but this is not a requirement. If the group members are not remembered, every host leaving a group sends a leave group message. In any case the router checks if it was the last host in the group, and stops forwarding multicast messages if so.
0x12	Version 1 Membership Report	This report is used for compatibility reasons with IGMP v1.

IGMP Versions

Version 2 of IGMP added the leave group message so that a client can explicitly leave the group. Version 1 had to wait for a timeout that could take up to five minutes. During this time, unwanted multicast transmissions are sent to the network, and for large data such as audio or video streams, this can use up a substantial part of the available bandwidth. When leaving the group with IGMP v2, latency is reduced to just a few seconds.

IGMP v3 is still in draft stage, but it is already available with Windows XP and adds specific joins and leaves with the source address(es). This capability makes the Source-Specific Multicast (SSM) protocol possible. With an IGMP v2 multicast, every member of the group can send multicast messages to every other member. SSM makes it possible to restrict the sender (source) of the group to a specific host or multiple hosts, which is a great advantage in the one-to-many application scenario. IGMP v3 messages have a different layout to IGMP v2 messages, and the size of an IGMP v3 message depends on how many source addresses are used.

> *At the time of writing, IGMP v3 is available as a draft version. With its release, a new RFC will update RFC 2236.*

Multicast Addresses

A class D multicast address starts with the binary values 1110 in the first four bits, making the address range from 224.0.0.0 to 239.255.255.255.

However, not every address of this range is available for multicasting; for example, the multicast addresses 224.0.0.0–224.0.0.255 are special purpose, and routers do not pass them across networks. Unlike normal IP addresses, where every country has a local representation to assign IP addresses, only the Internet Assigned Names and Numbers Authority (IANA, http://www.iana.org) is responsible for assigning multicast addresses. RFC 3171 defines the use of specific ranges of IP multicast addresses and their purposes.

*RFC 3171 uses **CIDR** (Classless InterDomain Routing) addresses for a shorthand notation of a range of IP addresses. The CIDR notation `224.0.0/24` is similar to the address range with the dotted quad-notation `224.0.0.0–224.0.0.255`. In the CIDR notation, the first part shows the fixed range of the dotted quad-notation followed by the number of fixed bits, so `232/8` is the shorthand CIDR notation for `232.0.0.0–232.255.255.255`.*

As a quick overview of multicast addresses, let's look at the three main ways in which they can be allocated:

- Static
- Dynamic
- Scope-relative

Static Multicast Addresses

Static multicast addresses that are needed globally are assigned by IANA. A few examples are listed in the table below:

IP Address	Protocol	Description
`224.0.0.1`	All systems on this subnet	The addresses starting with 224.0.0 belong to the **Local Network Control Block**, and are never forwarded by a router. Examples of these are `224.0.0.1` to send a message to all systems on the subnet, or `224.0.0.2` to send a message to all routers on the subnet. The DHCP server answers messages on the IP address `224.0.0.12`, but only on a subnet.
`224.0.0.2`	All routers on this subnet	
`224.0.0.12`	DHCP Server	
`224.0.1.1`	NTP, Network Time Protocol	The addresses in the CIDR range `224.0.1/24` belong to the **Internetwork Control Block**. Messages sent to these addresses can be forwarded by a router. Examples are the Network Time Protocol and WINS requests.
`224.0.1.24`	WINS Server	

A static address is one of global interest, used for protocols that need well-known addresses. These addresses may be hard-coded into applications and devices.

A complete list of actually reserved multicast addresses and their owners in the ranges defined by RFC 3171 can be found at the IANA web site at
http://www.iana.org/assignments/multicast-addresses.

The IANA web site offers a form that allows us to request multicast addresses for applications that need a globally unique IP address; this can be found at
http://www.iana.org/cgi-bin/multicast.pl.

Dynamic Multicast Addresses

Often, a dynamic multicast address would fit the purpose rather than a fixed static address. These requested-on-demand addresses have a specific lifetime. The concept of requesting dynamic multicast addresses is similar to DHCP (Dynamic Host Configuration Protocol) requests, and indeed in the first versions of MADCAP (Multicast Address Dynamic Client Allocation Protocol) was based on DHCP. Later MADCAP versions are completely independent of DHCP as they have quite different requirements.

With MADCAP, the client sends a unicast or a multicast message to a MADCAP server to request a multicast address. The server answers with a lease-based address.

The MADCAP protocol is defined by RFC 2730.

A MADCAP server comes with Windows 2000 Server and can be configured as part of the DHCP Server services.

Scope-Relative Multicast Addresses

Scope-relative multicast addresses are multicast addresses that are used only within a local group or organization. The address range `239.0.0.0` to `239.255.255.255` is reserved for administrative scope-relative addresses. These addresses can be reused with other local groups. Routers are typically configured with filters to prevent multicast traffic in this address range from flowing outside of the local network.

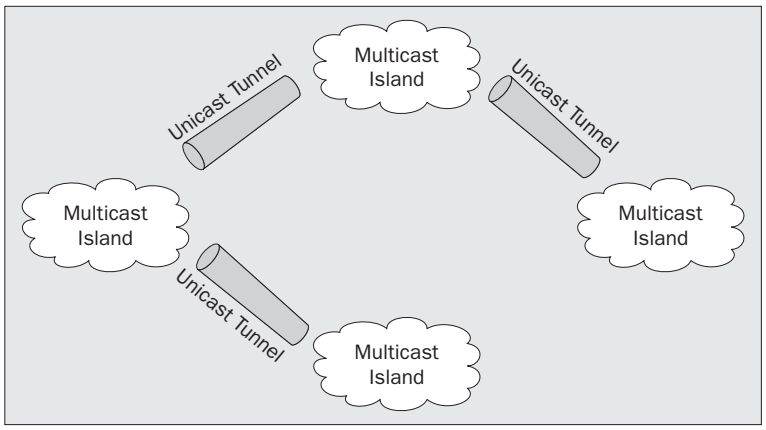
Administrative scope-relative addresses are defined in RFC 2365.

Another way to define the scope of multicast addresses is by using a TTL. We will look at TTLs in the next section.

Routing

Adding multicast capabilities to the Internet was not a straightforward task. When the multicast protocol was defined, router manufacturers didn't implement multicasting functionality because they didn't know if multicasting had a real use. Instead, they preferred to wait until they knew whether their customers actually wanted multicasting technology, creating a problem in that the technology couldn't take off, as it wasn't possible to use it over the Internet. To resolve this dilemma, the Multicast Backbone (**MBone)** was created in 1992. The MBone started with 40 subnetworks in four different countries, and now spans 3400 subnets in 25 countries.

The MBone connects together islands of subnetworks capable of multicasting through **tunnels** as can be seen in the next figure. Multicast messages are forwarded using a unicast connection between both tunnel ends to connect multiple islands across the Internet where multicasting is not supported:

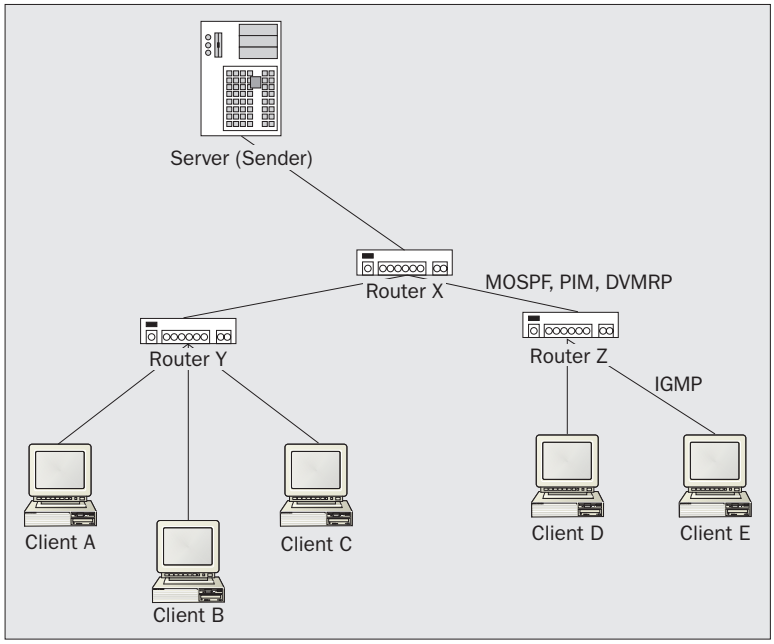


Today, routers are capable of routing multicast messages, but many Internet providers still don't support multicasts, and so the MBone is still a useful facility.

The web page reporting the actual status of multicast enabled networks in the Internet can be found at http://www.multicasttech.com/status.

Today, the MBone is used for audio and video multicasts, technical talks and seminars, NASA Space Shuttle missions, and so on. MBone tools (such as `sdr` or `multikit`) provide us with information about planned multicast events.

How is a multicast packet sent to a client? The next picture shows a server that sends multicast messages to a specific multicast group address. A client that wants to receive multicast packets for the defined multicast address must join the multicast group. Suppose client E joins the multicast group by sending an IGMP request to the routers on its local network. All routers of a subnet can be reached using the IP address `224.0.0.2`. In the picture, only router Z is in client E's subnet. The router registers the client as a member of the multicast group and informs other routers using a different protocol in order to pass information on multicast members across routers. We will discuss the multicast protocol used by routers shortly: MOSPF, PIM, or DVRMP. Multicast-enabled routers pass the information about the member in this group on to other routers. The server just needs to send a UDP message to the group address. Because router X now knows that a client wants to receive those messages, it forwards the multicast message to router Z that in turn forwards the message to the subnetwork of client E. Client E can read the multicast message. The message is never transmitted to the network of router Y because no client of that network joined the multicast group.



The scope of the multicast determines how many times multicast messages are forwarded by routers. Let's look at how the scope can be influenced.

Scoping

We have already discussed scoping when looking at administrative scope-relative multicast addresses belonging to a specific class D address range assigned by IANA, but there's another way to scope multicast messages.

When the client sends a multicast group report to join a multicast group, it defines a TTL (time to live) value that is sent in the IP packet. The TTL value defines by how many hops the membership report should be forwarded. A TTL value of 1 means that the group report never leaves the local network. Every router receiving the group report decrements the TTL value by 1 and discards the packet when the TTL reaches 0.

There's a problem defining the exact number of hops to a sender, as different routes may need a different number of hops, and administrative scope-relative multicast addresses have some advantages.

Routing Protocols

Different protocols can be used by routers to forward multicast membership reports and to find the best way from the sender to the client. When the MBone was created, the Distance Vector Multicast Routing Protocol (DVMRP) was the only protocol used. Now, Multicast Open Shortest Path First (MOSPF) and Protocol-Independent Multicast (PIM) are also widely used protocols for multicasting.

DVMRP uses a reverse path-flooding algorithm, where the router sends a copy of the network packet out to all paths except the one from where the packet originated. If no node in a router's network is a member of the multicast group, the router sends a prune message back to the sending router so that it knows it does not need to receive packets for that multicast group. DVMRP periodically refloods attached networks to reach new nodes that may be added to the multicast group. Now you can see why DVMRP doesn't scale too well!

MOSPF is an extension of the OSPF (Open Shortest Path First) protocol. With MOSPF, all routers must be aware of all available links to networks hosting members of multicast groups. MOSPF calculates routes when multicast traffic is received. This protocol can only be used in networks where OSPF is used as a unicast routing protocol because MOSPF routes are exchanged between routers using OSPF. Also, MOSPF won't scale well if many multicast groups are used, or if the groups change often, because this can gobble up a lot of the router's processing power.

PIM uses two different algorithms for sending messages to group members. When the members are widely distributed across different networks, PIM-SM (Sparse Mode) is employed, and when a group uses only a few networks, PIM-DM (Dense Mode) is used. PIM-DM uses a reverse path-flooding algorithm similar to DVMRP, except that any unicast routing protocol can be used. PIM-SM defines a registration point for proper routing of packets.

Scalability

A great advantage of multicasting is scalability. If, say, we send 123 bytes to 1000 clients using unicast, the network is loaded with 123,000 bytes because the message must be sent once for every client. With multicast, sending the same 123 bytes to 1000 clients only requires 123 bytes to be sent to the network, because all clients will receive the same message.

On the other hand, if we send the 123 bytes using a broadcast, the network load would be low with 123 bytes similar to multicast. But broadcast not only has the disadvantage that messages can't cross different networks, but also that all other clients not interested in the broadcast message need to handle the broadcast packet up to the transport layer before it can determine that no socket is listening to the message, and the packet is discarded.

Multicasting is the most efficient and scalable way to send messages to multiple clients.

Reliability

IP multicasting doesn't offer any compatible transport-level protocol that is both reliable and implements a flow mechanism. UDP on the other hand guarantees neither that messages arrive, nor that they arrive in the correct order. In many scenarios we don't have a problem if some messages are lost, and when listening to a live concert on the net, users expect to miss a few packets – and it is preferable to a faithful reproduction that pauses while data is resent. However, a high quality listening application that caches a lot of data in advance would probably prefer a reliable mechanism. If we want to use multicasting to install an application on multiple workstations simultaneously, a reliable mechanism is essential. If some messages were lost, the installed application may not run, or could even produce harmful effects.

If guaranteed delivery is needed for a multicast, we must add custom handshaking using a reliable protocol. One way this can be done is by adding packet numbers and a checksum to the data that is sent as part of the multicast. If the receiver detects a corrupted packet because of an incorrect checksum, or a packet is missing, it sends a NACK message to the sender, and the sender can resend the corrupted or missing packet. Note that the use of NACK messages in the event of errors is far more scalable than the alternative of sending acknowledgement messages for every packet correctly received.

Windows XP uses NACK messages in this way, as part of its reliable multicasting, provided through Message Queuing. If we use Windows XP (or Windows .NET Server) on both the client and server, we wouldn't need to implement a NACK mechanism ourselves.

Security

What about multicasting and security? We can differentiate multicasting security issues according to whether the Internet or an intranet is used, and whether we're securing multicast communication within a group.

A firewall acts as a security gateway between the Internet and an intranet. We'll assume that our Internet provider supports the MBone on the Internet side, and that multicasting is enabled in the intranet. Our firewall would stop multicast messages passing from the Internet to the intranet, and vice versa, and we must explicitly enable the multicast address and the port to pass through the firewall.

Regarding secure communication within a multicast group, the IETF's working group on Multicast Security (MSEC) has made a proposal for multicasting group key management that promises to establish a standard way for secure communication among authorized group members. This proposal would prevent anyone outside the group reading the group's messages. At the time of writing, the IETF already has a draft document describing secure group key management. The release of this document is planned for December 2002.

Using Multicast Sockets with .NET

Now that we've covered the basic principles and issues of multicasting, let's start playing with the .NET classes that support it. We'll start with a look at the code needed for a sender and a receiver.

Sender

The sending application has no special tasks that we haven't already talked about in previous chapters, and we can simply use the `UdpClient` class to send multicast messages. The only difference to what we've done in previous chapters is that we must now use a multicast address. The `IPEndPoint` object `remoteEP` will point to the group address and the port number that will be used by the group:

```
IPAddress groupAddress = IPAddress.Parse("234.5.6.11");
int remotePort = 7777;
int localPort = 7777;
IPEndPoint remoteEP = new IPEndPoint(groupAddress, remotePort);
UdpClient server = new UdpClient(localPort);
server.Send(data, data.Length, remoteEP);
```

The multicast group address must also be made known to clients joining the group. We can do this using a fixed address defined in a configuration file that clients can access, but we can also use a MADCAP server to get a multicast address dynamically. In that case, we have to implement a way to tell the client about the dynamically assigned addresses. We could do this using a stream socket that the client connects to, and sending the multicast address to the client. We will implement a stream socket to tell the client about the multicast address later on, when we come to the picture show application.

Receiver

Clients have to join the multicast group. The method `JoinMulticastGroup()` of the `UdpClient` class already implements this. This method sets the socket options `AddMembership` and `MulticastTimeToLive`, and sends an IGMP group report message to the router. The first parameter of `JoinMulticastGroup()` denotes the IP address of the multicast group, and the second parameter represents the TTL value (the number of routers that should forward the report message).

```
UdpClient udpClient = new UdpClient();
udpClient.JoinMulticastGroup(groupAddress, 50);
```

To drop a group membership, we call `UdpClient.DropMulticastGroup()`, which takes an IP address parameter specifying the same multicast group address as used with `JoinMulticastGroup()`:

```
udpClient.DropMulticastGroup(groupAddress);
```

Using the Socket Class

Instead of using the `UdpClient` class, we can also use the `Socket` class directly. The following code does practically the same as the `UdpClient` class above. A UDP socket is created with the constructor of the `Socket` class, and then the socket options `AddMembership` and `MulticastTimeToLive` are set with the method `SetSocketOption()`. We have already used this method in Chapter 4, now we'll use it with multicast options. The first argument we pass is `SocketOptionLevel.IP`, because the IGMP protocol is implemented in the IP module. The second argument specifies a value of the `SocketOptionName` enumeration. The `AddMembership` value is used to send an IGMP group membership report, and `MulticastTimeToLive` sets the number of hops by which the multicast report should be forwarded. For the group membership report, we also have to specify the IP address of the multicast group. The IP address can be specified with the helper class `MulticastOption`:

```
public void SetupMulticastClient(IPAddress groupAddress, int timeToLive)
{
   Socket socket = new Socket(AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);

   MulticastOption multicastOption = new MulticastOption(groupAddress);
   socket.SetSocketOption(SocketOptionLevel.IP,
                          SocketOptionName.AddMembership,
                          multicastOption);

   socket.SetSocketOption(SocketOptionLevel.IP,
                          SocketOptionName.MulticastTimeToLive,
                          timeToLive);
}
```

Leaving the multicast group is done by calling `SetSocketOption()` with the enumeration value `SocketOptionName.DropMembership`:

```
public void StopMulticastClient(IPAddress groupAddress, int timeToLive)
{
   Socket socket = new Socket(AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);

   MulticastOption multicastOption = new MulticastOption(groupAddress);
   socket.SetSocketOption(SocketOptionLevel.IP,
                          SocketOptionName.DropMembership,
                          multicastOption);
}
```

The advantage of the `Socket` class over `UdpClient` is that we have more options available for multicasting. In addition to the options we have seen for joining and dropping a group, Windows XP has the enumeration value `SocketOptionName.AddSourceGroup` to join a multicast source group using SSM.

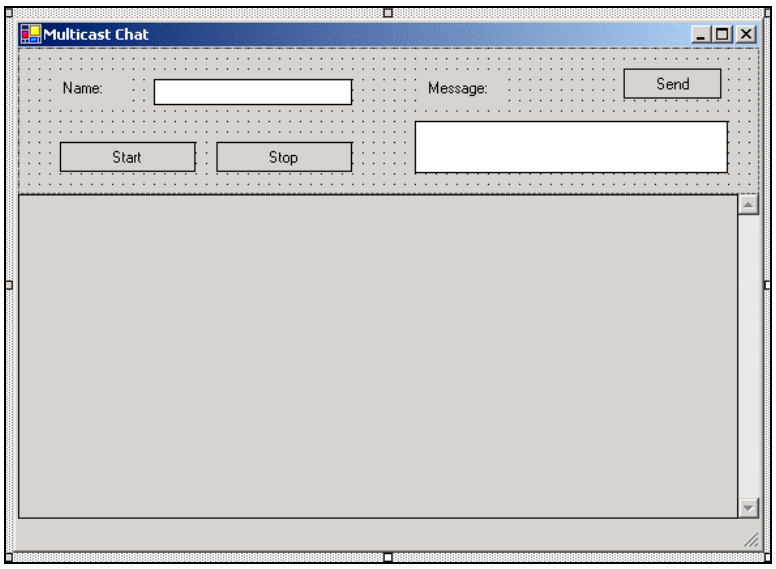
Creating a Chat Application

Now we can start writing a full multicast application. The first of our sample applications is a simple chat application that multiple users can use to send messages to all other chat clients. In this application, every system acts as both client and server. Every user can enter a message that is sent to all multicast participants.

The chat application is created as a Windows Forms project called `MulticastChat`, creating an executable with the name `MulticastChat.exe`. The main form class in this application is `ChatForm.cs`.

User Interface

The user interface of the chat application allows the user to enter a chat name, and join the network communication by pressing the Start button. When this button is pressed, the multicast group is joined, and the application starts listening to the group address. Messages are entered in the text box below the Message label, and are sent to the group when the Send button is pressed:



The table below shows the major controls of the form with their name and any non-default property values:

Control Type	Name	Properties
TextBox	textName	Text = ""
Button	buttonStart	Text = "Start"
Button	buttonStop	Enabled = false Text = "Stop"
Button	buttonSend	Enabled = false Text = "Send"
TextBox	textMessage	Multiline = true Text = ""
TextBox	textMessages	Multiline = true ReadOnly = true Scrollbars = Vertical Text = ""
StatusBar	statusBar	

`ChatForm` is the main class of the application as you can see in the code below. The code shows the .NET namespaces and private fields that will be used by all the methods that we'll add to the class as we progress:

```
using System;
using System.Configuration;
using System.Collections.Specialized;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.Threading;
using System.Windows.Forms;
```

```
namespace Wrox.Networking.Multicast
{
   public class ChatForm : System.Windows.Forms.Form
   {
      private bool done = true;       // Flag to stop listener thread
      private UdpClient client;       // Client socket
      private IPAddress groupAddress; // Multicast group address
      private int localPort;          // Local port to receive messages
      private int remotePort;         // Remote port to send messages
      private int ttl;

      private IPEndPoint remoteEP;
      private UnicodeEncoding encoding = new UnicodeEncoding();

      private string name;           // user name in chat
      private string message;        // message to send

      //...
```

Configuration Settings

The multicast address and port numbers should be easily configurable, so we'll create an XML application configuration file called `MulticastChat.exe.config` with the following content. This configuration file must be placed in the same directory as the executable resides (which will be `Debug\bin` when running from Visual Studio .NET with debugging).

```
<?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="32" />
   </appSettings>
</configuration>
```

The value of the `GroupAddress` key must be a class D IP address as discussed earlier in the chapter. `LocalPort` and `RemotePort` use the same values to make the chat application both a receiver and a sender with the same port number.

> *To start this application twice on the same system for testing purposes, you will have to copy the application together with the configuration file into two different directories. Because two running applications on one system may not listen to the same port number, you will also have to change the port numbers for `LocalPort` and `RemotePort` in the configuration file to two different ports. Each application will need `RemotePort` to be set to the value of `LocalPort` in the second application.*

We've set the TTL value in this file to 32. If you don't want to forward group membership reports across routers in your multicast environment, change this value to 1.

This configuration file is read by the `ChatForm` class constructor using the class `System.Configuration.ConfigurationSettings`. If the configuration file doesn't exist, or it is incorrectly formatted, an exception is thrown that we catch to display an error message:

```
        public ChatForm()
        {
            //
            // Required for Windows Form Designer support
            //
            InitializeComponent();

            try
            {
                // Read the application configuration file
                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,
                                "Error in application configuration file!",
                                "Error Multicast Chat", MessageBoxButtons.OK,
                                MessageBoxIcon.Error);
                buttonStart.Enabled = false;
            }
        }

    }
}
```

Joining the Multicast Group

In the `Click` handler of the Start button, we read the name that was entered in the text box `textName` and write it to the `name` field. Next, we create a `UdpClient` object and join the multicast group by calling the method `JoinMulticastGroup()`. Then we create a new `IPEndPoint` object referencing the multicast address and the remote port for use with the `Send()` method to send data to the group:

```
        private void OnStart(object sender, System.EventArgs e)
        {
            name = textName.Text;
            textName.ReadOnly = true;

            try
            {
                // Join the multicast group
                client = new UdpClient(localPort);
                client.JoinMulticastGroup(groupAddress, ttl);

                remoteEP = new IPEndPoint(groupAddress, remotePort);
```

In the next section of code, we create a new thread that will receive messages sent to the multicast address because the `UdpClient` class doesn't support asynchronous operations. An alternative way of achieving asynchronous operations would be to use the raw `Socket` class. The `IsBackground` property of the thread is set to `true` so that the thread will be stopped automatically when the main thread quits.

After starting the thread, we send an introduction message to the multicast group. To convert a string to the byte array that the `Send()` method requires, we call `UnicodeEncoding.GetBytes()`:

```
            // Start the receiving thread
            Thread receiver = new Thread(new ThreadStart(Listener));
            receiver.IsBackground = true;
            receiver.Start();

            // Send the first message to the group
            byte[] data = encoding.GetBytes(name + " has joined the chat");
            client.Send(data, data.Length, remoteEP);
```

The last action performed by the `OnStart()` method is to enable the Stop and Send buttons, and disable the Start button. We also write a handler for the `SocketException` that could occur if the application is started twice listening on the same port:

```
            buttonStart.Enabled = false;
            buttonStop.Enabled = true;
            buttonSend.Enabled = true;
         }
         catch (SocketException ex)
         {
            MessageBox.Show(this, ex.Message, "Error MulticastChat",
                            MessageBoxButtons.OK, MessageBoxIcon.Error);
         }
      }
```

Receiving Multicast Messages

In the method of the listener thread that we created earlier, we wait in the `client.Receive()` method until a message arrives. With the help of the class `UnicodeEncoding`, the received byte array is converted to a string.

The returned message should now be displayed in the user interface. There is an important issue to pay attention to when using threads and Windows controls. In native Windows programming, it is possible to create Windows controls from different threads, but only the thread that created the control may invoke methods on it, so all function calls on the Windows control must occur in the creation thread. In Windows Forms, the same model is mapped to the .NET Windows Forms classes. All methods of Windows Forms controls must be called on the creation thread – with the exception of the method `Invoke()` and its asynchronous variants, `BeginInvoke()` and `EndInvoke()`. These can be called from any thread, as it forwards the method that should be called to the creation thread of the Windows control. That creation thread then calls the method.

So instead of displaying the message in the text box directly, we call the `Invoke()` method of the Form class to forward the call to the creation thread of the Form class. Because this is the same thread that created the text box, this fulfills our requirements.

The `Invoke()` method requires an argument of type `Delegate`, and because any delegate derives from this class, every delegate can be passed to this method. We want to invoke a method that doesn't take parameters: `DisplayReceivedMessage()`, and there's already a predefined delegate in the .NET Framework to invoke a method without parameters: `System.Windows.Forms.MethodInvoker`. This delegate accepts methods without parameters, such as our method `DisplayReceivedMessage()`.

```
// Main method of the listener thread that receives the data
private void Listener()
{
   done = false;

   try
   {
      while (!done)
      {
         IPEndPoint ep = null;

         byte[] buffer = client.Receive(ref ep);
         message = encoding.GetString(buffer);

         this.Invoke(new MethodInvoker (DisplayReceivedMessage));
      }
   }
   catch (Exception ex)
   {
      MessageBox.Show(this, ex.Message, "Error MulticastChat",
                      MessageBoxButtons.OK, MessageBoxIcon.Error);

   }
}
```

In the `DisplayReceivedMessage()`implementation, we write the received message to the `textMessages` text box, and write some informational text to the status bar:

```
private void DisplayReceivedMessage()
{
   string time = DateTime.Now.ToString("t");
   textMessages.Text = time + "   " + message + "\r\n" +
                       textMessages.Text;
   statusBar.Text = "Received last message at " + time;
}
```

Sending Multicast Messages

Our next task is to implement the message sending functionality in the `Click` event handler of the Send button. As we have already seen, a string is converted to a byte array using the `UnicodeEncoding` class:

```
private void OnSend(object sender, System.EventArgs e)
{
   try
   {
      // Send a message to the group
      byte[] data = encoding.GetBytes(name + ": " + textMessage.Text);
      client.Send(data, data.Length, remoteEP);
      textMessage.Clear();
      textMessage.Focus();
   }
   catch (Exception ex)
   {
      MessageBox.Show(this, ex.Message, "Error MulticastChat",
                      MessageBoxButtons.OK, MessageBoxIcon.Error);
   }
}
```

Dropping the Multicast Membership

The `Click` event handler for the Stop button, the method `OnStop()`, stops the client listening to the multicast group by calling the `DropMulticastGroup()` method. Before the client stops receiving the multicast data, a final message is sent to the group saying that the user has left the conversation:

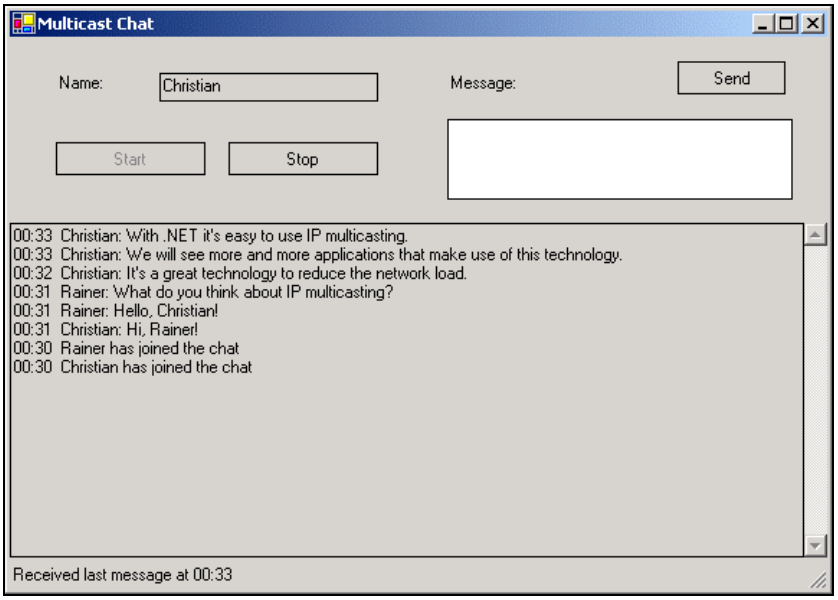
```
private void OnStop(object sender, System.EventArgs e)
{
   StopListener();
}

private void StopListener()
{
   // Send a leaving message to the group
   byte[] data = encoding.GetBytes(name + " has left the chat");
   client.Send(data, data.Length, remoteEP);

   // Leave the group
   client.DropMulticastGroup(groupAddress);
   client.Close();

   // Tell the receiving thread to stop
   done = true;

   buttonStart.Enabled = true;
   buttonStop.Enabled = false;
   buttonSend.Enabled = false;
}
```

Because the multicast group should be left not only when the user presses the Stop button, but also when they exit the application, we will handle the `Closing` event of the form in the `OnClosing()` method. If the server has not already been stopped, we again call the `StopListener()` method to stop listening to the group after sending a final message to the group:

```
        private void OnClosing(object sender,
                               System.ComponentModel.CancelEventArgs e)
        {
           if (!done)
              StopListener();
        }
```

Starting the Chat Application

Now we can start the chat application on multiple systems and start the conversation. Note that newer messages appear towards the top of the chat window:



A Picture Show Application

The second application we are going to look at is a picture show application. This application illustrates a multicast scenario where a single application sends messages to multiple clients. This application is a little more challenging as the messages that can be sent are of larger sizes which don't necessarily fit into datagram packets.

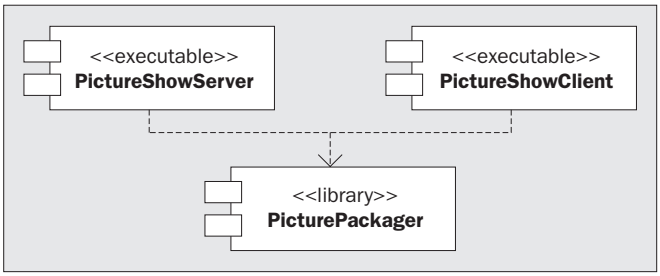
The picture show server allows pictures on its file system to be selected for multicasting to all clients that have joined the multicast group.

> *The picture show is a large application, and so rather than try to cover it in its entirety, we'll focus on the most important aspects related to network communication. The complete working application is available for download from the Wrox web site.*

The Picture Show Solution

The complete picture show application consists of three Visual Studio .NET projects:

- `PictureShowServer` – a Windows server application
- `PictureShowClient` – a Windows client application
- `PicturePackager` – a class library used by both client and server



To run the application, the `PictureShowServer.exe` executable and the `PicturePackager.dll` library must be copied to the server. The client systems need `PictureShowClient.exe`, `PicturePackager.dll`, and the application configuration file `PictureShowClient.exe.config`.

The server application must be started and initialized before the client applications can run. You must also set the server name in the client application to that of your server.

The first thing to do is consider how we're going to send pictures to our application's users.

Creating a Picture Protocol

Pictures can be sent to the multicast group that are too large to fit in a datagram packet. We have to split the picture into multiple packages. Also, we want to send some more data in addition to the picture stream. So we need some layout of the data that are sent across the wire.

To make it easy to parse the data so that the format can easily be extended for future versions of the application, we'll use an XML format for the packages that contain the picture data, and make use of classes from the `System.Xml` namespace.

> *To save space, we could define a custom binary format for the data packet, but when sending a picture, the XML overhead is not large compared to the size of the picture. Using XML gives us the advantage of an existing parser, and this format allows us to easily add more elements in future versions.*

One of our XML packages will fit into a single IP datagram. The root element is `<PicturePackage>` with an attribute called `Number` that identifies the fragments that belong together. `<Name>` and `<Data>` are child elements of the `<PicturePackage>` element. The `<Name>` element is informational and can be used for a name of the picture; the `<Data>` element is the base-64 encoded binary data segment of the picture. The attribute `SegmentNumber` allows us to work out how to put the segments together to create a complete picture; `LastSegmentNumber` informs us how many segments are needed to do so:

```
<PicturePackage Number="4">
   <Name>hello.jpg</Name>
   <Data SegmentNumber="2" LastSegmentNumber="12" Size="2400">
      <!-- base-64 encoded picture data -->
   </Data>
</PicturePackage>
```

The `PicturePackager` assembly contains two classes: the `PicturePackager` utility class and the entity class `PicturePackage`. A single `PicturePackage` object corresponds to an XML `<PicturePackage>` file like that we see above. We can produce the XML representation for a single picture segment using the `GetXml()` method on the class.

The `PicturePackage` class offers two constructors. The first is for use on the server, creating a picture fragment from native data types (`int`, `string`, and `byte[]`).The second constructor is designed for use on the client, and creates a fragment from an XML source.

`PicturePackager` is a utility class that has static methods only. It splits up a complete picture into multiple segments with the `GetPicturePackages()` method, and recreates a complete picture by merging the constituent segments with the `GetPicture()` method.

Picture Packages

So, the `PicturePackage` class represents one segment of a complete picture, and here it is reproduced in full. It starts by defining read-only properties that map to XML elements of the above format:

```
using System;
using System.Xml;
using System.Text;

namespace Wrox.Networking.Multicast
{
   public class PicturePackage
   {
      private string name;
      private int id;
      private int segmentNumber;
      private int numberOfSegments;
      private byte[] segmentBuffer;

      public string Name
      {
         get
         {
            return name;
         }
      }

      public int Id
      {
         get
         {
            return id;
         }
      }
```

```
        public int SegmentNumber
        {
           get
           {
              return segmentNumber;
           }
        }

        public int NumberOfSegments
        {
           get
           {
              return numberOfSegments;
           }
        }

        public byte[] SegmentBuffer
        {
           get
           {
              return segmentBuffer;
           }
        }
```

Next, we come to the two constructors. One takes multiple arguments to create a `PicturePackage` object by the sender, and the other takes XML data received from the network to recreate the object on the receiver:

```
        // Creates a picture segment from data types
        // Used by the server application
        public PicturePackage(string name, int id, int segmentNumber,
                              int numberOfSegments, byte[] segmentBuffer)
        {
           this.name = name;
           this.id = id;
           this.segmentNumber = segmentNumber;
           this.numberOfSegments = numberOfSegments;
           this.segmentBuffer = segmentBuffer;
        }

        // Creates a picture segment from XML code
        // Used by the client application
        public PicturePackage(XmlDocument xml)
        {
           XmlNode rootNode = xml.SelectSingleNode("PicturePackage");
           id = int.Parse(rootNode.Attributes["Number"].Value);

           XmlNode nodeName = rootNode.SelectSingleNode("Name");
           this.name = nodeName.InnerXml;

           XmlNode nodeData = rootNode.SelectSingleNode("Data");
           numberOfSegments = int.Parse(nodeData.Attributes[
                                        "LastSegmentNumber"].Value);
```

```
            segmentNumber = int.Parse(nodeData.Attributes[
                                      "SegmentNumber"].Value);
            int size = int.Parse(nodeData.Attributes["Size"].Value);
            segmentBuffer = Convert.FromBase64String(nodeData.InnerText);
         }
```

The only other item in this class is the `GetXml()` method, which converts the picture segment into an `XmlDocument` object with the help of classes from the `System.Xml` namespace. The XML representation is returned as a string:

```
         // Return XML code representing a picture segment
         public string GetXml()
         {
            XmlDocument doc = new XmlDocument();

            // Root element <PicturePackage>
            XmlElement picturePackage = doc.CreateElement("PicturePackage");

            // <PicturePackage Number="number"></PicturePackage>
            XmlAttribute pictureNumber = doc.CreateAttribute("Number");
            pictureNumber.Value = id.ToString();
            picturePackage.Attributes.Append(pictureNumber);

            // <Name>pictureName</Name>
            XmlElement pictureName = doc.CreateElement("Name");
            pictureName.InnerText = name;
            picturePackage.AppendChild(pictureName);

            // <Data SegmentNumber="" Size=""> (base-64 encoded fragment)
            XmlElement data = doc.CreateElement("Data");
            XmlAttribute numberAttr = doc.CreateAttribute("SegmentNumber");
            numberAttr.Value = segmentNumber.ToString();
            data.Attributes.Append(numberAttr);

            XmlAttribute lastNumberAttr =
                         doc.CreateAttribute("LastSegmentNumber");
            lastNumberAttr.Value = numberOfSegments.ToString();
            data.Attributes.Append(lastNumberAttr);

            data.InnerText = Convert.ToBase64String(segmentBuffer);
            XmlAttribute sizeAttr = doc.CreateAttribute("Size");
            sizeAttr.Value = segmentBuffer.Length.ToString();
            data.Attributes.Append(sizeAttr);

            picturePackage.AppendChild(data);

            doc.AppendChild(picturePackage);

            return doc.InnerXml;
         }
      }
   }
```

More detailed information about XML and the .NET classes that work with it can be found in the Wrox book Professional XML for .NET Developers *(ISBN 1-86100-531-8).*

Picture Packager

The `PicturePackager` class is a utility class consisting of static methods only. It is used by both sender and receiver. The `GetPicturePackages()` method splits up an image into multiple packages in the form of an array of `PicturePackage` objects where every segment of the picture is represented by a single `PicturePackage` object:

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;

namespace Wrox.Networking.Multicast
{

   public class PicturePackager
   {

      protected PicturePackager()
      {
      }

      // Return picture segments for a complete picture
      public static PicturePackage[] GetPicturePackages(string name,
                                                        int id,
                                                        Image picture)
      {
         return GetPicturePackages(name, id, picture, 4000);
      }

      // Return picture segments for a complete picture
      public static PicturePackage[] GetPicturePackages(string name, int id,
                                                        Image picture,
                                                        int segmentSize)
      {
         // Save the picture in a byte array
         MemoryStream stream = new MemoryStream();
         picture.Save(stream, ImageFormat.Jpeg);

         // Calculate the number of segments to split the picture
         int numberSegments = (int)stream.Position / segmentSize + 1;

         PicturePackage[] packages = new PicturePackage[numberSegments];

         // Create the picture segments
         int sourceIndex = 0;
         for (int i=0; i < numberSegments; i++)
         {
            // Calculate the size of the segment buffer
            int bytesToCopy = (int)stream.Position - sourceIndex;
            if (bytesToCopy > segmentSize)
               bytesToCopy = segmentSize;
```

```
            byte[] segmentBuffer = new byte[bytesToCopy];
            Array.Copy(stream.GetBuffer(), sourceIndex, segmentBuffer,
                       0, bytesToCopy);

            packages[i] = new PicturePackage(name, id, i + 1,
                                             numberSegments, segmentBuffer);

            sourceIndex += bytesToCopy;
         }

         return packages;
      }
```

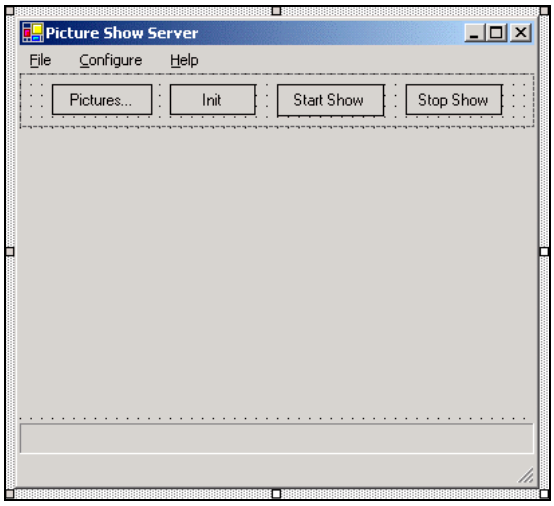
The receiver uses the inverse `GetPicture()` method that takes all `PicturePackage` objects for a single picture and returns the complete image object:

```
      // Returns a complete picture from the segments passed in
      public static Image GetPicture(PicturePackage[] packages)
      {
         int fullSizeNeeded = 0;
         int numberPackages = packages[0].NumberOfSegments;
         int pictureId = packages[0].Id;

         // Calculate the size of the picture data and check for consistent
         //  picture IDs
         for (int i=0; i < numberPackages; i++)
         {
            fullSizeNeeded += packages[i].SegmentBuffer.Length;
            if (packages[i].Id != pictureId)
               throw new ArgumentException(
                  "Inconsistent picture ids passed", "packages");
         }

         // Merge the segments to a binary array
         byte[] buffer = new byte[fullSizeNeeded];
         int destinationIndex = 0;
         for (int i = 0; i < numberPackages; i++)
         {
            int length = packages[i].SegmentBuffer.Length;
            Array.Copy(packages[i].SegmentBuffer, 0, buffer,
                       destinationIndex, length);
            destinationIndex += length;
         }

         // Create the image from the binary data
         MemoryStream stream = new MemoryStream(buffer);
         Image image = Image.FromStream(stream);

         return image;
      }
   }
}
```

The `System.Drawing` and `System.Xml` assemblies must be referenced by the `PicturePackager` assembly.

Picture Show Server

Now we've got the assembly to split and merge pictures, we can start on the implementation of the server application. The server project is `PictureShowServer`, and its main form class is `PictureServerForm`, contained in the file `PictureShowServer.cs` shown here:



The controls on this dialog are listed in the following table:

Control Type	Name	Comments
`MainMenu`	`mainMenu`	The menu has the main entries File, Configure, and Help. The File menu has the submenus Init, Start, Stop, and Exit. The Configure menu allows configuration of the Multicast Session, the Show Timings, and Pictures. The Help menu just offers an About option.
`Button`	`buttonPictures`	The Pictures... button will allow the user to configure the pictures that should be presented.
`Button`	`buttonInit`	The Init button will publish the multicast address and port number to clients using a TCP socket.
`Button`	`buttonStart`	The Start button starts sending the pictures to the multicast group address.
`Button`	`buttonStop`	The Stop button stops the picture show prematurely.
`PictureBox`	`pictureBox`	The picture box will show the picture that is currently being transferred to clients.

Control Type	Name	Comments
ProgressBar	progressBar	The progress bar indicates how many pictures of the show have been transferred.
StatusBar	statusBar	The status bar shows information about what's currently going on.
ImageList	imageList	The image list holds all images that make up the show.

The `PictureServerForm` class contains the `Main()` method, and there are three other dialog classes, and the `InfoServer` class.

The other three dialogs, `ConfigurePicturesDialog`, `MulticastConfigurationDialog`, and `ConfigureShowDialog`, are used to configure application settings. The `ConfigurePicturesDialog` dialog allows us to select image files from the server's file system to make up the picture show. `MulticastConfigurationDialog` sets up the multicast address and port number, and the interface where the pictures should be sent to in case the server system has multiple network cards. `ConfigureShowDialog` allows the time between pictures to be specified.

The other class, `InfoServer.cs`, starts its own thread that acts as an answering server for a client application. This thread returns information about the group address and port number to clients.

Let's have a look at the code for the start up class for the application, which is called `PictureServerForm` and resides in the `PictureShowServer.cs` source file. Firstly, we'll see the namespaces and fields that the class will require for the methods covered in subsequent sections:

```
using System;
using System.Drawing;
using System.Windows.Forms;
using System.Net;
using System.Net.Sockets;
using System.Text;
using System.IO;
using System.Xml;
using System.Threading;

namespace Wrox.Networking.Multicast
{
   public class PictureServerForm : System.Windows.Forms.Form
   {
      private string[] fileNames;     // Array of picture filenames
      private object filesLock = new object();  // Lock to synchronize
                                                //  access to filenames

      private UnicodeEncoding encoding = new UnicodeEncoding();

      // Multicast group address, port, and endpoint
      private IPAddress groupAddress = IPAddress.Parse("231.4.5.11");
      private int groupPort = 8765;
      private IPEndPoint groupEP;
      private UdpClient udpClient;
```

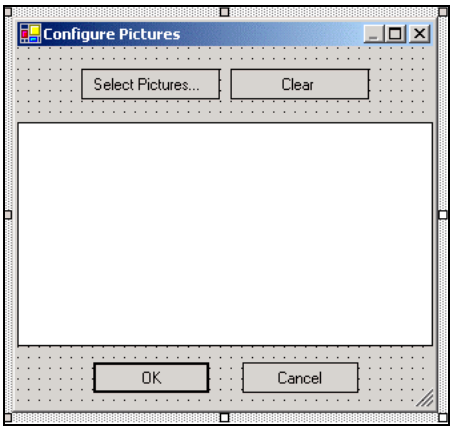
```
        private Thread senderThread;    // Thread to send pictures

        private Image currentImage;     // Current image sent

        private int pictureIntervalSeconds = 3; // Time between sending
                                                // pictures
        //...
```

Opening Files

One of the first actions the picture show server application must do is configure the pictures that are to be presented. `OnConfigurePictures()` is the handler for the Configure | Pictures menu item and the click event of the Pictures... button:

```
        private void OnConfigurePictures(object sender, System.EventArgs e)
        {
           ConfigurePicturesDialog dialog = new ConfigurePicturesDialog();
           if (dialog.ShowDialog() == DialogResult.OK)
           {
              lock (filesLock)
              {
                 fileNames = dialog.FileNames;
                 progressBar.Maximum = filenames.Length;
              }
           }
        }
```

This dialog opens a configure pictures dialog, which offers a preview of the pictures in a `ListView` with a `LargeIcon` view:



The controls used on this Form are detailed in the next table:

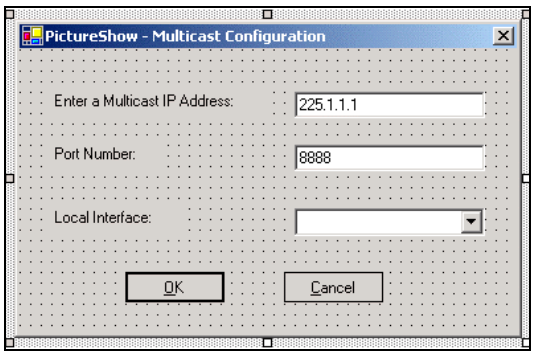
Control Type	Name	Comments
Button	buttonSelect	The Select Pictures... button displays the openFileDialog so the user can select the pictures for the show.
OpenFileDialog	openFileDialog	
Button	buttonClear	The Clear button removes all selected images.
Button	buttonOK	The OK or Cancel buttons close the dialog. If OK is clicked, the selected files are sent to the main form, and if Cancel is clicked, all file selections are ignored.
Button	buttonCancel	
ImageList	imageList	The ImageList Windows Forms component collects all selected images for display in the listViewPictures list view showing a preview for the user.
ListView	listViewPictures	

`OnFileOpen()` is the handler for the `Click` event of the Select Pictures... button. It is where we create `Image` objects from the files selected by `OpenFileDialog` for adding to the `ImageList` associated with the `ListView`:

```
        private void OnFileOpen(object sender, System.EventArgs e)
        {
           if (openFileDialog.ShowDialog() == DialogResult.OK)
           {
              fileNames = openFileDialog.FileNames;

              int imageIndex = 0;
              foreach (string fileName in fileNames)
              {
                 using (Image image = Image.FromFile(fileName))
                 {
                    imageList.Images.Add(image);

                    listViewPictures.Items.Add(fileName, imageIndex++);
                 }
              }
           }
        }
```

Configuring Multicasting

Another configuration dialog of the server application allows configuring of the multicast address and multicast port number:



This dialog also allows us to select the local interface that should be used for sending multicast messages. This can be useful if the system has multiple network cards, or is connected both to a dial-up network and a local network.

The combo box listing the local interfaces is filled at form startup. First we call `Dns.GetHostName()` to retrieve the hostname of the local host, and then `Dns.GetHostByName()` to get an `IPHostEntry` object containing all the IP addresses of the local host. If the host has multiple IP addresses, the string `"Any"` is added to the combo box to allow the user to send multicast messages across all network interfaces. If there is only one network interface, the combo box is disabled, as it will not be possible to select a different interface.

Here is the constructor in `MulticastConfigurationDialog.cs`

```
public MulticastConfigurationDialog()
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent();

   string hostname = Dns.GetHostName();
   IPHostEntry entry = Dns.GetHostByName(hostname);
   IPAddress[] addresses = entry.AddressList;

   foreach (IPAddress address in addresses)
   {
      comboBoxLocalInterface.Items.Add(address.ToString());
   }
   comboBoxLocalInterface.SelectedIndex = 0;

   if (addresses.Length > 1)
   {
      comboBoxLocalInterface.Items.Add("Any");
   }
   else
   {
      comboBoxLocalInterface.Enabled = false;
   }
}
```

Another interesting aspect of this class is the validation of the multicast address. The text box `textBoxIPAddress` has the handler `OnValidateMulticastAddress()` assigned to the event `Validating`. The handler checks that the IP address entered is in the valid range of multicast addresses:

```
        private void OnValidateMulticastAddress(object sender,
                                System.ComponentModel.CancelEventArgs e)
        {
           try
           {
              IPAddress address = IPAddress.Parse(textBoxIPAddress.Text);

              string[] segments = textBoxIPAddress.Text.Split('.');
              int network = int.Parse(segments[0]);

              // Check address falls in correct range
              if ((network < 224) || (network > 239))
                 throw new FormatException("Multicast addresses have the" +
                                           "range 224.x.x.x to 239.x.x.x");

              // Check address is not a reserved class D
              if ((network == 224) && (int.Parse(segments[1]) == 0)
                  && (int.Parse(segments[2]) == 0))
                 throw new FormatException("The Local Network Control Block" +
                           "cannot be used for multicasting groups");
           }
           catch (FormatException ex)
           {
              MessageBox.Show(ex.Message);
              e.Cancel = true;
           }
        }
```

We now return to the events in the main `PictureServerForm` class in the file `PictureShowServer.cs`. When the Init button is pressed on the main dialog, we want the listening server to start up so that it can send the group address and port number to requesting clients. We do this by creating an `InfoServer` object with the IP address and the port number, and then invoke `Start()`. This method starts a new thread to handle client requests, as we will see next. `OnInit()` finishes by calling some helper methods that enable the Start button, and disable the Init button:

```
// PictureShowServer.cs

        private void OnInit(object sender, System.EventArgs e)
        {
           InfoServer info = new InfoServer(groupAddress, groupPort);
           info.Start();

           UIEnableStart(true);
           UIEnableInit(false);
        }
```

The `InfoServer` class does all the work of responding to client requests by sending the multicast group address and port number to the clients in a separate thread. The class constructor initializes the `InfoServer` object with the group address and the group port. The `Start()` method (invoked by the `OnInit()` in the `PictureShowServer` class) creates the new thread. The main method of the newly created thread, `InfoMain()`, sets up a TCP stream socket where we simply place the multicast address and the port number separated by a colon (`:`) for sending to the client as soon as one connects to the server:

```
// InfoServer.cs

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

namespace Wrox.Networking.Multicast
{
   public class InfoServer
   {
      private IPAddress groupAddress;
      private int groupPort;
      private UnicodeEncoding encoding = new UnicodeEncoding();

      public InfoServer(IPAddress groupAddress, int groupPort)
      {
         this.groupAddress = groupAddress;
         this.groupPort = groupPort;
      }

      public void Start()
      {
         // Create a new listener thread
         Thread infoThread = new Thread(new ThreadStart(InfoMain));
         infoThread.IsBackground = true;
         infoThread.Start();
      }

      protected void InfoMain()
      {
         string configuration = groupAddress.Address.ToString() + ":" +
                                groupPort.ToString();

         // Create a TCP streaming socket that listens to client requests
         Socket infoSocket = new Socket(AddressFamily.InterNetwork,
                                        SocketType.Stream,
                                        ProtocolType.Tcp);

         try
         {
            infoSocket.Bind(new IPEndPoint(IPAddress.Any, 8777));
            infoSocket.Listen(5);
```

```
            while (true)
            {
               // Send multicast configuration information to clients
               Socket clientConnection = infoSocket.Accept();
               clientConnection.Send(encoding.GetBytes(configuration));
               clientConnection.Shutdown(SocketShutdown.Both);
               clientConnection.Close();
            }
         }
         finally
         {
            infoSocket.Shutdown(SocketShutdown.Both);
            infoSocket.Close();
         }
      }
   }
}
```

Chapter 4 delved into stream sockets in more detail.

The `UIEnableStart()` helper method enables or disables the Start button to prevent the user pressing the wrong button. This method is called in the method `OnInit()`, and, as you would expect, is very similar to `UIEnableInit`:

```
// PictureShowServer.cs

      private void UIEnableStart(bool flag)
      {
         if (flag)
         {
            buttonStart.Enabled = true;
            buttonStart.BackColor = Color.SpringGreen;
            miFileStart.Enabled = true;
         }
         else
         {
            buttonStart.Enabled = false;
            buttonStart.BackColor = Color.LightGray;
            miFileStart.Enabled = false;
         }
      }
```

Sending Pictures

`OnStart()` is the method that handles the `Click` event of the Start button, and is where the sending thread is initialized and started. The start method of this thread is `SendPictures()`, which we look at next.

```
// PictureShowServer.cs

      private void OnStart(object sender, System.EventArgs e)
      {
         if (fileNames == null)
```

```
            {
               MessageBox.Show("Select pictures before starting the show!");
               return;
            }

            // Initialize picture sending thread
            senderThread = new Thread(new ThreadStart(SendPictures));
            senderThread.Name = "Sender";
            senderThread.Priority = ThreadPriority.BelowNormal;
            senderThread.Start();

            UIEnableStart(false);
            UIEnableStop(true);
         }
```

The list of filenames that we receive from the Configure Pictures dialog is used by the `SendPictures()` method of the sending thread. Here we load a file from the array `fileNames` to create a new `Image` object that is passed to the `SendPicture()` method. Then the progress bar is updated with the help of a delegate to reflect the ongoing progress. When building the multicast chat application, we discussed issues relating to the use of Windows controls in multiple threads, and now we have to use the `Invoke()` method again:

```
// PictureShowServer.cs

      private void SendPictures()
      {
         InitializeNetwork();

         lock (filesLock)
         {
            int pictureNumber = 1;
            foreach (string fileName in fileNames)
            {
               currentImage = Image.FromFile(filename);
               Invoke(new MethodInvoker(SetPictureBoxImage);

               SendPicture(image, fileName, pictureNumber);
               Invoke(new MethodInvokerInt(IncrementProgressBar),
                                    new object[] {1});

               Thread.Sleep(pictureIntervalSeconds);
               pictureNumber++;
            }
         }
         Invoke(new MethodInvoker(ResetProgress));
         Invoke(new MethodInvokerBoolean(UIEnableStart),
                new object[] {true});
         Invoke(new MethodInvokerBoolean(UIEnableStop),
                new object[] {false});
      }
```

We've already discussed the `MethodInvoker` delegate in the `System.Windows.Forms` namespace when we used it in the multicast chat application. The `MethodInvoker` delegate allows us to call methods that take no arguments. Now we also need methods that take an `int`, a `string`, or a `bool` argument to set specific values and enable or disable certain user interface elements. The delegates for these purposes are placed at the top of the `PictureShowServer.cs` file:

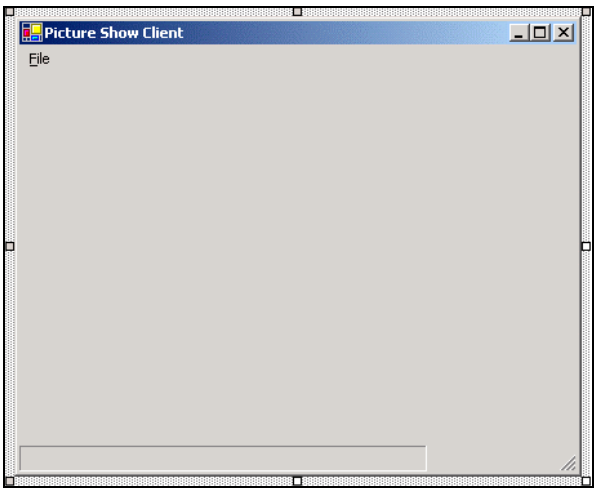
```
namespace Wrox.Networking.Multicast
{
  public delegate void MethodInvokerInt(int x);
  public delegate void MethodInvokerString(string s);
  public delegate void MethodInvokerBoolean(bool flag);
  //...
```

The `SendPicture()` method splits up a single picture image using the `PicturePackager` utility class. Every individual picture package is converted to a byte array with an `Encoding` object of type `UnicodeEncoding`. This byte array is then sent to the multicast group address by the `Send()` method of `UdpClient`:

```
// PictureShowServer.cs

      private void SendPicture(Image image, string name, int index)
      {
         string message = "Sending picture " + name;
         Invoke(new MethodInvokerString(SetStatusBar),
                                        new Object[] {message});

         PicturePackage[] packages =
               PicturePackager.GetPicturePackages(name, index, image, 1024);

         // Send all segments of a single picture to the group
         foreach (PicturePackage package in packages)
         {
            byte[] data = encoding.GetBytes(package.GetXml());
            int sendBytes = udpClient.Send(data, data.Length);
            if (sendBytes < 0)
               MessageBox.Show("Error sending");

            Thread.Sleep(300);
         }

         message = "Picture " + name + " sent";
         Invoke(new MethodInvokerString(SetStatusBar),
                new object[] { message });
      }
```

Picture Show Client

The multicast picture show client has a simple user interface consisting of a menu, a picture box, and a status bar on a form. The status bar has three panels showing not only status messages but also the multicast address and port numbers. The File menu entry has Start, Stop, and Exit submenus:



The components on this form are described in the following table:

Control Type	Name	Comments
`MainMenu`	`mainMenu`	The main menu has a single File menu entry with Start, Stop, and Exit submenus.
`PictureBox`	`pictureBox`	The picture box displays a picture as soon as all fragments that make it up have been received.
`StatusBar`	`statusBar`	The status bar is split into three panels with the `Panels` property. The first panel (`statusBarPanelMain`) shows normal status text, the second and the third (`statusBarPanelAddress` and `statusBarPanelPort`) display the group address and port number.

The form class is `PictureClientForm` in the file `PictureShowClient.cs`. It houses the methods `GetMulticastConfiguration()` (to request the multicast group information from the server) `OnStart()` (to join the multicast group), and `Listener()` (to start a new thread). It also contains the method `DisplayPicture()`, which is called for each picture in the show, and `OnStop()`, which terminates the listening thread.

Let's start with the namespaces and private fields that the client `PictureClientForm` class requires:

```
// PictureShowClient.cs

using System;
using System.Drawing;
using System.Collections;
using System.Windows.Forms;
using System.Collections.Specialized;
using System.Net;
```

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

namespace Wrox.Networking.Multicast
{
  public delegate void MethodInvokerInt(int i);
  public delegate void MethodInvokerString(string s);

  public class PictureClientForm : System.Windows.Forms.Form
  {
    private IPAddress groupAddress; // Multicast group address
    private int groupPort;          // Multicast group port
    private int ttl;
    private UdpClient udpClient;    // Client socket for receiving
    private string serverName;      // Hostname of the server
    private int serverInfoPort;     // Port for group information

    private bool done = false;      // Flag to end receiving thread

    private UnicodeEncoding encoding = new UnicodeEncoding();

    // Array of all pictures received
    private SortedList pictureArray = new SortedList();
```

Receiving the Multicast Group Address

As in the multicast chat application, we use a configuration file for setting up the client picture show application. In this case, it is used to configure the name and address of the server so clients can connect to the TCP socket that returns the multicast address and port number:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <appSettings>
      <add key="ServerName" value="localhost" />
      <add key="ServerPort" value="7777" />
      <add key="TTL" value="32" />
   </appSettings>
</configuration>
```

The values in this configuration file are read by the `PictureClientForm` form class constructor, which also invokes the `GetMulticastConfiguration()` method:

```
// PictureShowClient.cs

      public PictureClientForm()
      {
         //
         // Required for Windows Form Designer support
         //
         InitializeComponent();
```

```
            try
            {
               // Read the application configuration file
               NameValueCollection configuration =
                     ConfigurationSettings.AppSettings;
               serverName = configuration["ServerName"];
               serverInfoPort = int.Parse(configuration["ServerPort"]);
               ttl = int.Parse(configuration["TTL"]);
            }
            catch
            {
               MessageBox.Show("Check the configuration file");
            }

            GetMulticastConfiguration();
         }
```

We connect to the server in the `GetMulticastConfiguration()` method. After connecting, we can call `Receive()`, as the server immediately starts to send once a connection is received. The byte array received is converted to a string using an object of the `UnicodeEncoding` class. The string contains the multicast address and the port number separated by a colon (`:`), so we split the string and set the member variables `groupAddress` and `groupPort`.

As mentioned above, the status bar has two additional panels, `statusBarPanelAddress` and `statusBarPanelPort`, where the multicast address and multicast port number are displayed:

```
// PictureShowClient.cs

      private void GetMulticastConfiguration()
      {
         Socket socket = new Socket(AddressFamily.InterNetwork,
                                    SocketType.Stream, ProtocolType.Tcp);
         try
         {
            // Get the multicast configuration info from the server
            IPHostEntry server = Dns.GetHostByName(serverName);
            socket.Connect(new IPEndPoint(server.AddressList[0],
                           serverInfoPort));
            byte[] buffer = new byte[512];
            int receivedBytes = socket.Receive(buffer);
            if (receivedBytes < 0)
            {
               MessageBox.Show("Error receiving");
               return;
            }
            socket.Shutdown(SocketShutdown.Both);

            string config = encoding.GetString(buffer);
            string[] multicastAddress = config.Split(':');
            groupAddress = new IPAddress(long.Parse(multicastAddress[0]));
            groupPort = int.Parse(multicastAddress[1]);
```

```
            statusBarPanelAddress.Text = groupAddress.ToString();
            statusBarPanelPort.Text = groupPort.ToString();
         }
         catch (SocketException ex)
         {
            if (ex.ErrorCode == 10061)
            {
               MessageBox.Show(this, "No server can be found on "
                              +  serverName + ", at port " + serverInfoPort,
                               "Error Picture Show", MessageBoxButtons.OK,
                               MessageBoxIcon.Error);
            }
            else
            {
               MessageBox.Show(this, ex.Message, "Error Picture Show",
                               MessageBoxButtons.OK, MessageBoxIcon.Error);
            }
         }
         finally
         {
            socket.Close();
         }
      }
```

Joining the Multicast Group

Once we have the multicast address and port number, we can join the multicast group. The constructor of the `UdpClient` class creates a socket listening to the port of the group multicast, and then we join the multicast group by calling `JoinMulticastGroup()`.

The task of receiving the messages and packaging the pictures together belongs to the newly created listener thread that invokes the `Listener()` method:

```
// PictureShowClient.cs

      private void OnStart(object sender, System.EventArgs e)
      {
         udpClient = new UdpClient(groupPort);
         try
         {
            udpClient.JoinMulticastGroup(groupAddress, ttl);
         }
         catch (Exception ex)
         {
            MessageBox.Show(ex.Message);
         }

         Thread t1 = new Thread(new ThreadStart(Listener));
         t1.Name = "Listener";
         t1.IsBackground = true;
         t1.Start();
      }
```

Receiving the Multicast Picture Data

The listener thread waits in the `Receive()` method of the `UdpClient` object until data is received. The byte array received is converted to a string with the encoding object, and in turn a `PicturePackage` object is initialized, passing in the XML string that is returned from the encoding object.

We have to merge all the picture fragments of a single picture received together to create full images for the client. This is done using the `pictureArray` member variable of type `SortedList`. The key of the sorted list is the picture ID; the value is an array of `PicturePackage` objects that make up a complete picture.

We then check whether the `pictureArray` already contains an array of `PicturePackage`s for the received picture. If it does, the fragment is added to the array; if not, we allocate a new array of `PicturePackage`s.

After updating the status bar with information about the picture fragment, we invoke the `DisplayPicture()` method if we have already received all the fragments of a picture:

```
// PictureShowClient.cs

      private void Listener()
      {
         while (!done)
         {
            // Receive a picture segment from the multicast group
            IPEndPoint ep = null;
            byte[] data = udpClient.Receive(ref ep);

            PicturePackage package = new PicturePackage(
                                     encoding.GetString(data));

            PicturePackage[] packages;
            if (pictureArray.ContainsKey(package.Id))
            {
               packages = (PicturePackage[])pictureArray[package.Id];
               packages[package.SegmentNumber - 1] = package;
            }
            else
            {
               packages = new PicturePackage[package.NumberOfSegments];
               packages[package.SegmentNumber - 1] = package;
               pictureArray.Add(package.Id, packages);
            }

            string message = "Received picture " + package.Id + " Segment "
                           + package.SegmentNumber;
            Invoke(new MethodInvokerString(SetStatusBar),
                                           new object[] {message});

            // Check if all segments of a picture are received
            int segmentCount = 0;
            for (int i = 0; i < package.NumberOfSegments; i++)
            {
                if (packages[i] != null)
                segmentCount++;
```

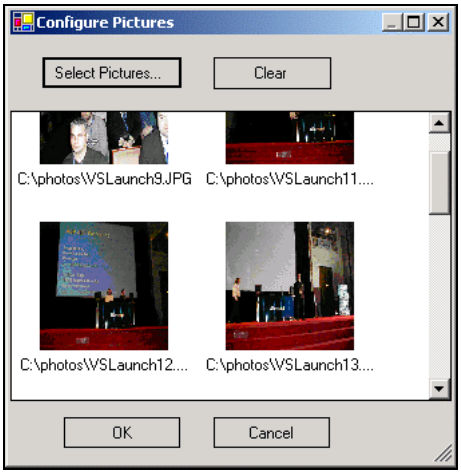
```
            }

            // All segments are received, so draw the picture
            if (segmentCount == package.NumberOfSegments)
            {
               this.Invoke(new MethodInvokerInt(DisplayPicture),
                           new object[] {package.Id});
            }
         }
      }
```

All we have to do in `DisplayPicture()` is recreate the picture with the help of the `PicturePackager` utility class. The picture is displayed in the picture box on the form. Because the picture fragments are no longer needed, we can now free some memory by removing the item representing the `PicturePackage` array of the sorted list collection:

```
      private void DisplayPicture(int id)
      {
         PicturePackage[] packages = (PicturePackage[])pictureArray[id];

         Image picture = PicturePackager.GetPicture(packages);

         pictureArray.Remove(id);

         pictureBox.Image = picture;
      }
```

Starting the Picture Show

Now we can start the server and client picture show applications. We first need to select the pictures using the Select Pictures… button on the server interface. The picture below shows the Configure Pictures dialog after some pictures have been selected:



Pressing the Init button on the main dialog starts the `InfoServer` thread, listening for client requests. Before starting this thread, the multicast addresses can be changed, but the defaults should be enough to get going. Once the `InfoServer` thread is underway, clients can join the session by selecting File | Start from their menus.

The screenshot below shows the server application in action:

The client application details picture segments as they are received in the status bar, to the right of the multicast group address and port number:

Summary

In this chapter, we've looked at the architecture and issues of multicasting, and seen how multicasting can be implemented with .NET classes.

Multicasting is a pretty young technology, and it has a bright future. I expect many issues, such as security and reliability, will be resolved by the standards fairly soon. For multicasting to be truly viable across the Internet, one or two improvements are required.

However, multicasting already has many useful applications, and its usage is set to grow a great deal in the coming years. Using multicasting to send data in a one-to-many scenario, or for many-to-many group applications, considerably reduces the network load when compared to unicasting.

After discussing the architecture of multicasting, we implemented two multicast applications. The first was a chat application in a many-to-many scenario; the second was a picture server in a one-to-many scenario sending large data to the multicast group. These demonstrated how easy the built-in multicast methods of the `UdpClient` class make multicasting in .NET.