



[Featured Articles: Fe  
Multiplayer and Networkir](#)

# Advanced WinSock Multiplayer Game Programming: Multicasting

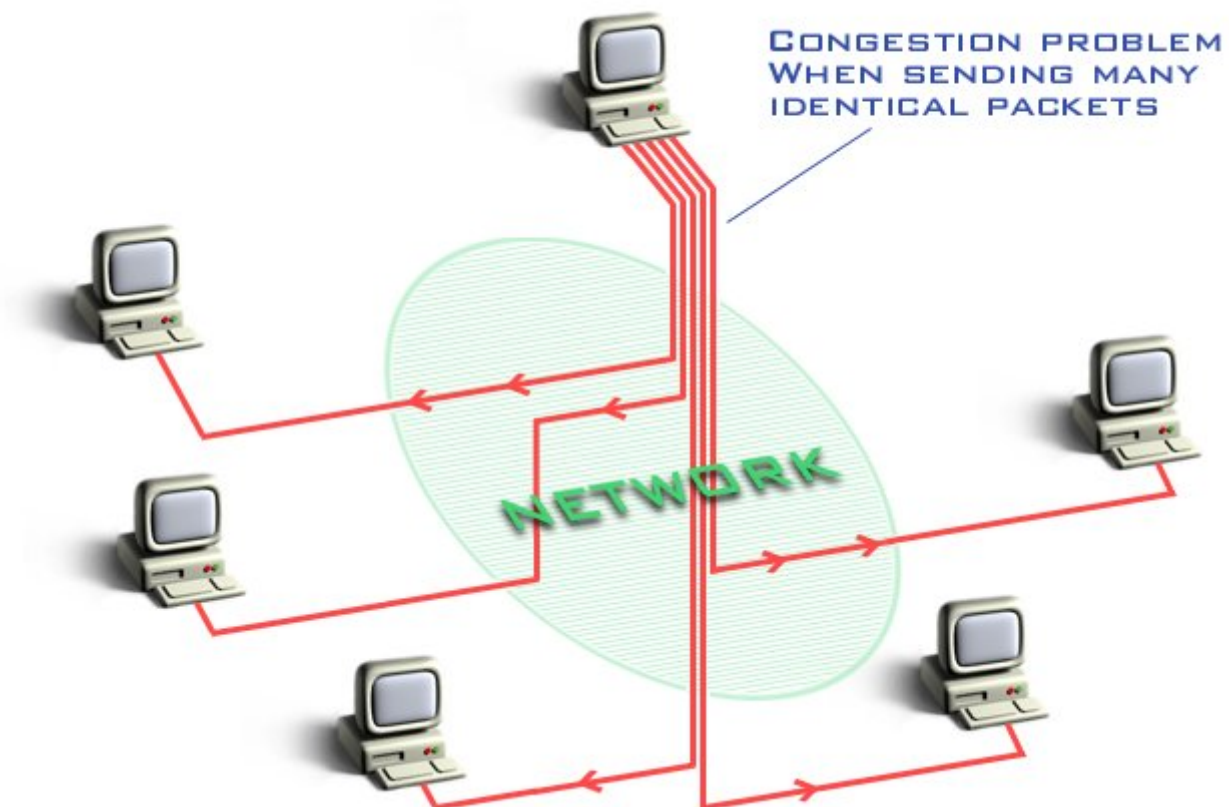
by [Denis Lukianov](#)

Combatting lag is a major problem in multiplayer network game development. As multiplayer game developers, we always strive to make things faster, leaner and meaner to reduce lag and free up bandwidth. This is why we often forsake the reliability of TCP for the speed that UDP provides. Multicasting is yet another step in the fight against latency, carrying many promises, including the transmission of very high quality streaming digital TV over Networks and in the future, the Internet. What is the magic behind multicasting and how can it be used in our games? In short, it can not only reduce server workload but is also a solution to the age old problem of players fining each other on networks without the game developer having to put up dedicated master servers (but more on that later).

Oh, and if DirectPlay uses multicasting extensively, then it's all more the reason for us to use it :).

## The Idea Behind Multicasting

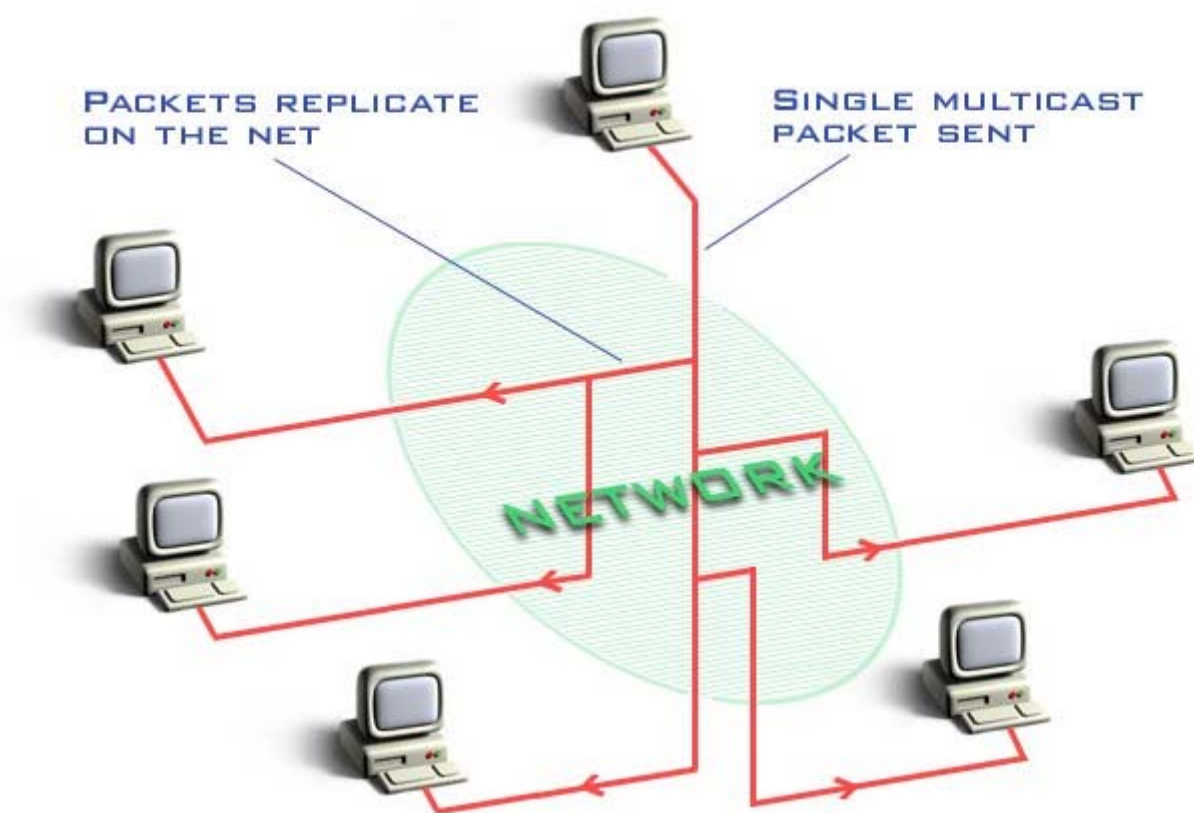
The theory goes something like this. In the most commonly used networking client-server model, when a client sends input to the server, this input updates the game state and then the server tells all the other clients about what has happened by sending the same information to all the clients:



As you can see there is a traffic problem on the server's Network connection. If, say, there were 32 players connected to the server at the time, then the same information would be sent 32 times (once to each player). If there were 20 bytes of data to be sent to each of the 32 players then 640 bytes would have to be sent through the server's Network connection. If that were to happen every time any of the 32 players pressed a key or moved the mouse, a huge amount of traffic is generated. Naturally, there is no replacement for good code practice and sending only the data that is needed, but multicasting can seriously help.

So how can multicasting help? Well, Multicasting can dramatically reduce the amount of data that needs to be sent by taking the task of packet replication away from the game server to the actual Network infrastructure. In multicasting, packets can be sent to groups of Network addresses, instead of individual addresses.

This is similar to the way email works - when we want to send the same email message to multiple email addresses, we don't send the message to every address from our computer. Instead we send the message once, telling the server to replicate the message to all the other addresses.



## The Darker Side

Of course, there are reasons why multicasting is not commonly used:

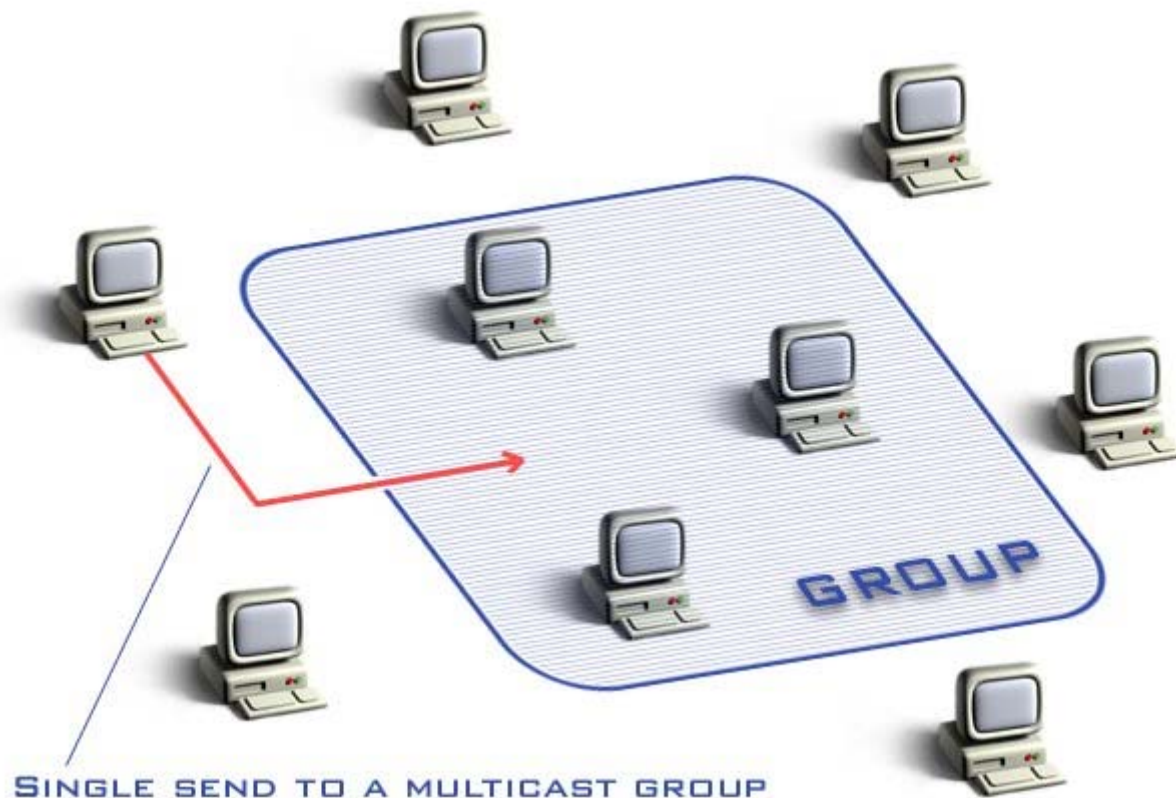
- Some ISPs and networks don't support multicasting yet. Bastards. So if you want to implement multicasting in a game, you're better off adding it as an option. Internet multicasting is rarely supported, but hopefully it will be in the future.
- Multicasting only makes a worthwhile gain in performance when network data is replicated, realistically only worth bothering when there is support for more than about three players.
- Multicasting requires some more coding and programmers are lazy to even look into it. As you will see, in fact it requires very little additional code. The corporate "Quality Digital TV via Multicasting" idea seems to put game programmers off the subject altogether, I suspect it has something to do with hacker ethics, so long live the .org's!

## How Multicasting Works

You may have heard of broadcasting. Broadcasting forwards data to every address on the network. Unlike broadcasting, multicasting only forwards to those addresses who have explicitly registered interest in the data.

On an IP Network supporting multicasting there are such things as multicast groups. If you want to receive multicast data packets, you must join a multicast group. Although it is should be possible to send data packets to a multicast group regardless of membership, it is often better to join a group before sending to it for reasons I won't venture into. If you are a member of a group to which you are sending multicast datapackets, you will receive a copy of the data packets. Also, a client will not receive all data packets from a multicast group, but only those which are sent to the port that the socket is bound to.

So a sensible idea would be for all the game clients to join a multicast group and wait for data on the same port. Then the server, by sending a single packet of data to that multicast group, would be sending to all the clients as the packets are replicated somewhere along the way.



We've seen the light, we've seen the darkness, so let us onto the code...

## Joining a Multicast Group and Receiving Multicast Data Packets

To receive multicast packets sent to a multicast group, your game will need to join or become a member of that multicast group. To request becoming a member of a multicast group is a lot simpler than you may at first imagine. You need to first *bind()* your UDP socket to a local port (elementary, my dear friend):

```
SOCKADDR_IN addrLocal;  
// We want to use the Internet address family
```

```

addrLocal.sin_family = AF_INET;
// Use any local address
addrLocal.sin_addr.s_addr = INADDR_ANY;
// Use arbitrary port - but the same as on other clients/servers
addrLocal.sin_port = htons(uiPort);
// Bind socket to our address
if(SOCKET_ERROR == bind(hUDPSocket, (LPSOCKADDR)&addrLocal,
                        sizeof(struct sockaddr)))
    {cout << "Euston, we have a problem";}
// Ready to switch to multicasting mode

```

And then just make a call to `setsockopt()`, and here's a prototype for your convenience \*grin\*:

```

int WINAPI setsockopt(SOCKET s, int level, int optname,
                    const char FAR * optval, int optlen);

```

If you thought you were getting away with just 1 new line of code to learn, you were wrong... you're only getting away with 4 new lines =). There are special parameters to prepare for this call: `s` is your socket handle, `level` should be set to `IPPROTO_IP`, `optname` should be set to `IP_ADD_MEMBERSHIP` and a pointer to the `p_mreq` structure passed as `optval`, with its length in `optlen`. This is what the `p_mreq` structure looks like:

```

struct ip_mreq {
    struct in_addr imr_multiaddr; /* multicast group to join */
    struct in_addr imr_interface; /* interface to join on */
}

```

It has 2 fields, both of them are `in_addr` structures: `imr_multiaddr` specifies the address of the multicast group to join and `imr_interface` specifies the local address `INADDR_ANY`.

There are special (Class 'D') addresses allocated for multicast groups. These are in the range from 224.0.1.0 to 239.255.255.255. You can choose an address from the range as the target multicast group to join, and set the `imr_multiaddr` to this address. The full `setsockopt()` call would look something like this:

```

struct ip_mreq mreq;
mreq.imr_multiaddr.s_addr = inet_addr("234.5.6.7");
mreq.imr_interface.s_addr = INADDR_ANY;
nRet = setsockopt(hUDPSocket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                (char*)&mreq, sizeof(mreq));

```

And that's all there is to it, apart from a lot of error checking which I've decided to leave out for clarity (aka Laziness). The socket will now receive data packets sent to the multicast group on the specified port with calls to `recvfrom()`:

```

SOCKADDR_IN addrSrc;
nRet = recvfrom(hUDPSocket, (char *)&Data, sizeof(Data), 0,
              (struct sockaddr*)&addrSrc, sizeof(addrSrc));

```

When you're finished with the group and want to leave, just repeat the call with identical parameters apart from `IP_ADD_MEMBERSHIP` which should be replaced with `IP_DROP_MEMBERSHIP`.

```

nRet = setsockopt(hUDPSocket, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                (char*)&mreq, sizeof(mreq));

```

Now that we can join a multicast group and receive packets sent to it, the logical thing to do is to learn how to send packets to a multicast group.

## Sending Multicast Data Packets

Sending multicast data packets is accomplished with a call to *sendto()*, specifying a multicast group address as the destination IP address and the wanted port (on which your clients are tuned to listen for data). So there really a lot to learn apart from using the TTL (Time To Live) socket option.

All IP packets carry a TTL value to make sure that they are discarded if they don't reach a destination so they don't clog up the Network. In a multicast data packet, TTL specifies how far a multicast data packet can travel:

TTL Threshold	Description
TTL equal to 0	Restricted to the same host
TTL equal to 1	Restricted to the same subnet
TTL equal to 32	Restricted to the same site
TTL equal to 64	Restricted to the same region
TTL equal to 128	Restricted to the same continent
TTL equal to 255	Unrestricted in scope

[From MSDN]

Multicasting is nowhere as dangerous as broadcasting in terms of unwanted traffic that it can produce but caution is advised when using some of the higher TTL values.

To set a socket's multicast TTL value, *setsockopt()* can be used with *IPPROTO\_IP* as the protocol level and *IP\_MULTICAST\_TTL* as the socket option.

```
char TTL = 32 ; // Restrict to our school network, for example
setsockopt(hUDPSocket, IPPROTO_IP, IP_MULTICAST_TTL,
           (char *)&TTL, sizeof(TTL));
```

Once the TTL is set, just *sendto()* away:

```
SOCKADDR_IN  addrDest;
szHi[50];

addrDest.sin_family = AF_INET;
// Target multicast group address
addrDest.sin_addr.s_addr = inet_addr("234.5.6.7");
// Port on which client is set to receive data packets
addrDest.sin_port = htons(uiPort);
// Something unoriginal to send
strcpy(szHi, "Hello Multicast Group!");

nRet = sendto(hUDPSocket, (char *)szHi, sizeof(szHi), 0,
              (struct sockaddr*)&addrDest, sizeof(addrDest));
```

We can now join multicast groups, send and receive data from them, but how do we implement multicasting as an option in our game and what would we use it for?

## Uses of Multicasting in Games

I can think of two ways straight away - one is to use it for reducing (maybe even eliminating) the amount of repeated data that a server has to send out, but another interesting use is a global server-less interface for finding other players on the Network.

The scenario: there are 2 people on a large Network running the same game that want to play together, but they don't know each other's IP addresses let alone the fact that the potential opponent exists. The common ways for connecting the 2 players:



- The players send out a broadcast message to the entire Network, however this would create huge traffic and will probably be restricted to sub Networks. Broadcasting on the Internet would create an enormous amount of traffic, so it is not allowed.
- The players connect to an intermediate, "known" master server IP, which tells them of each other's existence. These servers are costly to run and their uptime is often undependable.
- The players go to a chatroom hoping to find other players and play together. This will not connect all the players as some may be in different chatrooms. And the process of finding someone may take a while.

So here we are with an age old problem (how A finds B) on one hand and multicasting on the other. Multicasting groups always have the same address - a "known" address as in the case of a dedicated server, they are online 100% of the time - unlimited uptime, they don't cost anything to connect to or send information across. All game clients simply connects to a multicast group, multicast an "I want to play" message and the servers can then advertise their availability directly (instead of multicasting, to save bandwidth) to the clients who are members of the multicast group.

Sure, there are itsy-bitsy technical problems to sort out, but the idea is cool enough. And the TTL control allows us to query within a certain range of routers (see TTL table) so we can specifically only ask to send to our LAN, or a university network, or all servers within our country to respond. Don't you think that is COOL? I sure do.

The only problem (see "The Dark Side") remains multicast support by ISP's and Networks. So the best way to add multicasting to a game is still as an option (although I hope and pray that multicasting will be 100% supported in the future). But how do we integrate multicasting into our game as an option?

## Integrating Multicasting into Games

Ok then, where do we start? There are so many different types of multiplayer games that I won't even try to explain how to integrate multicasting into different types of games. Instead I'll just give a few possible ideas of solutions in a client-server relationship.

First of all, all the current network code should be kept as it is, when you add multicast support make sure you do not remove any existing code unless you really think it is necessary.

When adding multicast support, you can either do a parallel integration where multicasting runs along with existing code, or you could write two separate sets of network code and add a 'multicast on/off switch' for the user. The on/off switch would isolate servers using the other network code and add one more daunting and mysterious switch for the average newbie to get wrong. Parallel integration (bah, the things I learned in school last year) is my favorite as it will use multicasting only if it is supported and should be transparent to the user.

So let's stick with parallel integration - in this case the normal network code runs always, but the multicasting code only runs if multicasting is supported. How do we determine if multicasting is supported? Just read the error `setsockopt()` gives us when trying to join a group:

```
nRet = setsockopt(hUDPSocket, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                 (char*)&mreq, sizeof(mreq));
if(WSAEWOULDBLOCK == nRet)
{
    // Multicasting not supported. Damn.
}
```

The client-server relationship is a game of two halves. So what if the server supports multicasting while the client does not? How does the server know which clients are covered with a single send to the multicast group and which are not? The client first determines that it does not support multicasting, then connects to the server and tells the server whether it supports multicasting. The server usually keeps a list or array of clients, to which it is easy to add an extra boolean flag:

```
struct Client
{
    SOCKADDR_IN addrRemote;
    /* ... Game specific info here ...*/
    BOOL bSupportMulticast;
}
```

The server's function to send data sends a multicast to clients who support multicasting and normal UDP datagrams to those that do not. If, however, the server itself does not support multicasting then we must use the old method. Here's a useful code snippet for a server with multicasting as an option:

```
int SendToAll(char *Data)
{
    if(bServerSupportMulticast)
    {
        // First send multicast, then send individually
        // to those who don't support it
        SendMulticast(Data, addrMulticast);
        for(int index = 0; index < MAX_CLIENTS; index++)
        {
            if(Clients[index].Exist && !Clients[index].bSupportMulticast)
            {
                OldSendToClient(Data, Clients[index].addrRemote);
            }
        }
    }
    else
    {
        // Use the old method all the way regardless of support
        // as we ourselves don't support it
        for(int index = 0; index < MAX_CLIENTS; index++)
        {
            if(Clients[index].Exist)
            {
                OldSendToClient(Data, Clients[index].addrRemote);
            }
        }
    }
}
```

I hope I've shed some light on multicasting and its possible uses in games. If you've found this article the least bit interesting or have a problem, drop an email to [denis@voxelsoft.com](mailto:denis@voxelsoft.com). I hope to write another article soon, but for now, Happy Multicasting!

- Denis "Voxel" Lukianov

Thanks to Jan "Riva" Halfar for the wonderful diagrams.

**[Discuss this article in the forums](#)**

© 1999-2002 Gamedev.net. All rights reserved. [Terms of Use](#) [Privacy Policy](#)  
Comments? Questions? Feedback? [Send us an e-mail!](#)