Multicast over TCP/IP HOWTO

Table of Contents

Table of Contents

Multicast over TCP/IP HOWTO

Juan−Mariano de Goyeneche `<jmseyas@dit.upm.es>`

v1.0, 20 March 1998

This HOWTO tries to cover most aspects related to multicast over TCP/IP networks. So, a lot of information within it is not Linux−specific (just in case you don't use GNU/Linux... yet). Multicast is currently an active area of research and, at the time of writing, many of the "standards" are merely drafts. Keep it in mind while reading the lines that follow.

1.Introduction.

2.Multicast Explained.

3.Kernel requirements and configuration.

4.The MBone.

5.Multicast applications.

6.Multicast programming.

7.The internals.

8.Routing Policies and Forwarding Techniques.

9.Multicast Transport Protocols.

10.References.

11.Copyright and Disclaimer.

12.Acknowledgements.

1.Introduction.

I'll try to give here the most wide range, up to date and accurate information related to multicasting over TCP/IP networks that I can. Any feedback is very welcome. If you find any mistakes in this document, have any comments about its contents or an update or addition, please send them to me at the address listed at the top of this howto.

1.1 What is Multicast.

Multicast is... a need. Well, at least in some scenarios. If you have information (a *lot* of information, usually) that should be transmitted to various (but usually not *all*) hosts over an internet, then Multicast is the answer. One common situation in which it is used is when distributing real time audio and video to the set of hosts which have joined a distributed conference.

Multicast is much like radio or TV in the sense that only those who have tuned their receivers (by selecting a particular frequency they are interested on) receive the information. That is: you hear the channel you are interested in, but not the others.

1.2 The problem with Unicast.

Unicast is anything that is not broadcast nor multicast. All right, the definition is not very bright... When you send a packet and there is only one sender process −yours− and one recipient process (the *one* you are sending the packet to), then this is unicast. TCP is, by its own nature, unicast oriented. UDP supports a lot more paradigms, but if you are sending UDP packets and there is only one precess supposed to receive them, this is unicast too.

For years unicast transmissions proved to be enough for the Internet. It was not until 1993 when the first implementation of multicast saw the light in the 4.4 BSD release. It seems nobody needed it until then. Which were those new problems that multicast addressed?

Needless to say that the Internet has changed a lot since the "early days". Particularly, the appearance of the Web strongly transformed the situation: people didn't just want connections to remote hosts, mail and FTP. First they wanted to see the pictures people placed in their home pages, but later they also wanted to *see* and *hear* that people.

With today's technology it is possible to afford the "cost" of making a unicast connection with everyone who wants to see your web page. However, if you are to send audio and video, which needs a *huge* amount of bandwidth compared with web applications, you have −you *had*, until multicast came into scene− two options: to establish a separate unicast connection with *each* of the recipients, or to use broadcast. The first solution is not affordable: if we said that a *single* connection sending audio/video consumes a huge bandwidth, imagine having to establish hundreds or, may be, thousands of those connections. Both the sending computer and your network would collapse.

Broadcast seems to be *a* solution, but it's not certainly *the* solution. If you want all the hosts in your LAN to attend the conference, you may use broadcast. Packets will be sent only once and every host will receive them as they are sent to the broadcast address. The problem is that perhaps only a *few* of the hosts and not *all* are interested in those packets. Furthermore: perhaps some hosts are really interested in your conference, but they are outside of your LAN, a few routers away. And you know that broadcast works fine inside a LAN, but problems arise when you want broadcast packets to be routed across different LANs.

The best solution seems to be one in which you send packets to a certain special address (a certain frequency in radio/TV transmissions). Then, all hosts which have decided to join the conference will be aware of packets with that destination address, read them when they traverse the network, and pass them to the IP layer to be demultiplexed. This is similar to broadcasting in that you send only one broadcast packet and all the hosts in the network recognize and read it; it differs, however, in that not all multicast packets are read and processed, but only those that were previously registered in the kernel as being "of interest".

Those special packets are routed at kernel level like any packet because they *are* IP packets. The only difference might reside in the routing algorithm which tells the kernel where to route or not to route them.

2.Multicast Explained.

2.1 Multicast addresses.

As you probably know, the range of IP addresses is divided into "classes" based on the high order bits of a 32 bits IP address:

```
   Bit -->  0                           31            Address Range:
           +-+----------------------------+
           |0|       Class A Address      |       0.0.0.0 - 127.255.255.255
           +-+----------------------------+
           +-+-+--------------------------+
           |1 0|     Class B Address      |     128.0.0.0 - 191.255.255.255
           +-+-+--------------------------+
           +-+-+-+------------------------+
           |1 1 0|   Class C Address      |     192.0.0.0 - 223.255.255.255
           +-+-+-+------------------------+
           +-+-+-+-+----------------------+
           |1 1 1 0|  MULTICAST Address   |     224.0.0.0 - 239.255.255.255
           +-+-+-+-+----------------------+
           +-+-+-+-+-+--------------------+
           |1 1 1 1 0|     Reserved       |     240.0.0.0 - 247.255.255.255
           +-+-+-+-+-+--------------------+
```

The one which concerns us is the "Class D Address". Every IP datagram whose destination address starts with "1110" is an IP Multicast datagram.

The remaining 28 bits identify the multicast "*group*" the datagram is sent to. Following with the previous analogy, you have to tune your radio to hear a program that is transmitted at some specific frequency, in the same way you have to "tune" your kernel to receive packets sent to an specific multicast group. When you do that, it's said that the host has *joined* that group in the interface you specified. More on this later.

There are some special multicast groups, say "well known multicast groups", you should not use in your particular applications due the special purpose they are destined to:

- 224.0.0.1 is the *all−hosts* group. If you ping that group, all multicast capable hosts on the network should answer, as every multicast capable host *must* join that group at start−up on all it's multicast

capable interfaces.
- 224.0.0.2 is the *all−routers* group. All multicast routers must join that group on all it's multicast capable interfaces.
- 224.0.0.4 is the *all DVMRP routers*, 224.0.0.5 the *all OSPF routers*, 224.0.013 the *all PIM routers*, etc.

All this special multicast groups are regularly published in the "Assigned Numbers" RFC.

In any case, range 224.0.0.0 through 224.0.0.255 is reserved for local purposes (as administrative and maintenance tasks) and datagrams destined to them are never forwarded by multicast routers. Similarly, the range 239.0.0.0 to 239.255.255.255 has been reserved for "administrative scoping" (see section 2.3.1 for information on administrative scoping).

2.2 Levels of conformance.

Hosts can be in three different levels of conformance with the Multicast specification, according to the requirements they meet.

Level 0 is the "no support for IP Multicasting" level. Lots of hosts and routers in the Internet are in this state, as multicast support is not mandatory in IPv4 (it is, however, in IPv6). Not too much explanation is needed here: hosts in this level can neither send nor receive multicast packets. They must ignore the ones sent by other multicast capable hosts.

Level 1 is the "support for sending but not receiving multicast IP datagrams" level. Thus, note that it is not necessary to join a multicast group to be able to send datagrams to it. Very few additions are needed in the IP module to make a "Level 0" host "Level 1−compliant", as shown in section 2.3.

Level 2 is the "full support for IP multicasting" level. Level 2 hosts must be able to both send and receive multicast traffic. They must know the way to join and leave multicast groups and to propagate this information to multicast routers. Thus, they must include an Internet Group Management Protocol (IGMP) implementation in their TCP/IP stack.

2.3 Sending Multicast Datagrams.

By now, it should be obvious that multicast traffic is handled at the transport layer with UDP, as TCP provides point−to−point connections, not feasibles for multicast traffic. (Heavy research is taking place to define and implement new multicast−oriented transport protocols. See section Multicast Transport Protocols for details).

In principle, an application just needs to open a UDP socket and fill with a class D multicast address the destination address where it wants to send data to. However, there are some operations that a sending process must be able to control.

TTL.

The TTL (Time To Live) field in the IP header has a double significance in multicast. As always, it controls the live time of the datagram to avoid it being looped forever due to routing errors. Routers decrement the TTL of every datagram as it traverses from one network to another and when its value reaches 0 the packet is dropped.

The TTL in IPv4 multicasting has also the meaning of "threshold". Its use becomes evident with an example: suppose you set a long, bandwidth consuming, video conference between all the hosts belonging to your department. You want that huge amount of traffic to remain in your LAN. Perhaps your department is big enough to have various LANs. In that case you want those hosts belonging to each of *your* LANs to attend the conference, but in any case you want to collapse the entire Internet with your multicast traffic. There is a need to limit how "long" multicast traffic will expand across routers. That's what the TTL is used for. Routers have a TTL threshold assigned to each of its interfaces, and only datagrams with a TTL greater than the interface's threshold are forwarded. Note that when a datagram traverses a router with a certain threshold assigned, the datagram's TTL is *not* decremented by the value of the threshold. Only a comparison is made. (As before, the TTL is decremented by 1 each time a datagram passes across a router).

A list of TTL thresholds and their associated scope follows:

```
TTL     Scope
----------------------------------------------------------------------
   0    Restricted to the same host. Won't be output by any interface.
   1    Restricted to the same subnet. Won't be forwarded by a router.
 <32         Restricted to the same site, organization or department.
 <64 Restricted to the same region.
<128 Restricted to the same continent.
<255 Unrestricted in scope. Global.
```

Nobody knows what "site" or "region" mean exactly. It is up to the administrators to decide what this limits apply to.

The TTL−trick is not always flexible enough for all needs, specially when dealing with overlapping regions or trying to establish geographic, topologic and bandwidth limits simultaneously. To solve this problems, administratively scoped IPv4 multicast regions were established in 1994. (see D. Meyer's "*Administratively Scoped IP Multicast*" Internet draft). It does scoping based on multicast addresses rather than on TTLs. The range 239.0.0.0 to 239.255.255.255 is reserved for this administrative scoping.

Loopback.

When the sending host is Level 2 conformant and is also a member of the group datagrams are being sent to, a copy is looped back by default. This does not mean that the interface card reads its own transmission, recognizes it as belonging to a group the interface belongs to, and reads it from the network. On the contrary, is the IP layer which, by default, recognizes the to−be−sent datagram and copies and queues it on the IP input queue before sending it.

This feature is desirable in some cases, but not in others. So the sending process can turn it on and off at wish.

Interface selection.

Hosts attached to more than one network should provide a way for applications to decide which network interface will be used to output the transmissions. If not specified, the kernel chooses a default one based on system administrator's configuration.

2.4 Receiving Multicast Datagrams.

Joining a Multicast Group.

Broadcast is (in comparison) easier to implement than multicast. It doesn't require processes to give the kernel some rules regarding what to do with broadcast packets. The kernel just knows what to do: read and deliver *all* of them to the proper applications.

With multicast, however, it is necessary to advise the kernel which multicast groups we are interested in. That is, we have to ask the kernel to "join" those multicast groups. Depending on the underlying hardware, multicast datagrams are filtered by the hardware or by the IP layer (and, in some cases, by both). Only those with a destination group previously registered via a join are accepted.

Essentially, when we join a group we are telling the kernel: "OK. I know that, by default, you ignore multicast datagrams, but remember that I am interested in *this* multicast group. So, do read and deliver (to any process interested in them, not only to me) any datagram that you see in this network interface with this multicast group in its destination field".

Some considerations: first, note that you don't just join a group. You join a group *on* a particular network interface. Of course, it is possible to join the same group on more than one interface. If you don't specify a concrete interface, then the kernel will choose it based on its routing tables when datagrams are to be sent. It is also possible that more than one process joins the same multicast group on the same interface. They will all receive the datagrams sent to that group via that interface.

As said before, any multicast−capable hosts join the *all−hosts* group at start−up , so "pinging" 224.0.0.1 returns all hosts in the network that have multicast enabled.

Finally, consider that for a process to receive multicast datagrams it has to ask the kernel to join the group *and* bind the port those datagrams were being sent to. The UDP layer uses both the destination address and port to demultiplex the packets and decide which socket(s) deliver them to.

Leaving a Multicast Group.

When a process is no longer interested in a multicast group, it informs the kernel that *it* wants to leave that group. It is important to understand that this doesn't mean that the kernel will no longer accept multicast datagrams destined to that multicast group. It will still do so if there are more precesses who issued a "multicast join" petition for that group and are still interested. In that case *the host* remains member of the group, until all the processes decide to leave the group.

Even more: if you leave the group, but remain bound to the port you were receiving the multicast traffic on, and there are more processes that joined the group, you will still receive the multicast transmissions.

The idea is that joining a multicast group only tells the IP and data link layer (which in some cases explicitly tells the hardware) to accept multicast datagrams destined to that group. It is not a per−process membership, but a per−host membership.

Mapping of IP Multicast Addresses to Ethernet/FDDI addresses.

Both Ethernet and FDDI frames have a 48 bit destination address field. In order to avoid a kind of multicast ARP to map multicast IP addresses to ethernet/FDDI ones, the IANA reserved a range of addresses for multicast: every ethernet/FDDI frame with its destination in the range 01−00−5e−00−00−00 to 01−00−5e−ff−ff−ff (hex) contains data for a multicast group. The prefix 01−00−5e identifies the frame as multicast, the next bit is always 0 and so only 23 bits are left to the multicast address. As IP multicast groups are 28 bits long, the mapping can not be one−to−one. Only the 23 least significant bits of the IP multicast group are placed in the frame. The remaining 5 high−order bits are ignored, resulting in 32 different multicast groups being mapped to the same ethernet/FDDI address. This means that the ethernet layer acts as an imperfect filter, and the IP layer will have to decide whether to accept the datagrams the data−link layer passed to it. The IP layer acts as a definitive perfect filter.

Full details on IP Multicasting over FDDI are given in RFC 1390: "*Transmission of IP and ARP over FDDI Networks*". For more information on mapping IP Multicast addresses to ethernet ones, you may consult `draft−ietf−mboned−intro−multicast−03.txt`: "*Introduction to IP Multicast Routing*".

If you are interested in IP Multicasting over Token−Ring Local Area Networks, see RFC 1469 for details.

3.Kernel requirements and configuration.

Linux is, of course (you doubted it?), full Level−2 Multicast−Compliant. It meets all requirements to send, receive and act as a router (mrouter) for multicast datagrams.

If you want just to send and receive, you must say yes to "*IP: multicasting*" when configuring your kernel. If you also want your Linux box to act as a multicast router (mrouter) you also need to enable multicast routing in the kernel by selecting "*IP: forwarding/gatewaying*", "*IP: multicast routing*" and "*IP: tunneling*", the latter because new versions of `mrouted` relay on IP tunneling to send multicast datagrams encapsulated into unicast ones. This is necessary when establishing tunnels between multicast hosts separated by unicast−only networks and routers. (The `mrouted` is a daemon that implements the multicast routing algorithm −the routing policy− and instructs the kernel on how to route multicast datagrams).

Some kernel versions label multicast routing as "*EXPERIMENTAL*", so you should enable "*Prompt for* *development and/or incomplete code/drivers*" in the "*Code maturity level options*" section.

If, when running the `mrouted`, traffic generated in the same network your Linux box is connected to is correctly forwarded to the other network, but you can't see the other's network traffic on your local network, check whether you are receiving ICMP protocol error messages. Almost sure you forgot to turn on IP tunneling in your Linux router. It's a kind of stupid error when you know it but, believe me, its quite time−consuming when you don't, and there is no apparent reason that explains what is going wrong. A sniffer proves to be quite useful in these situations!

(You can see more on multicast routing on section Routing Policies and Forwarding Techniques; `mrouted` and tunnels are also explained in sections The MBone and Multicast applications).

Once you have compiled and installed your new kernel, you should provide a default route for multicast traffic. The goal is to add a route to the network 224.0.0.0.

The problem most people seem to face in this stage of the configuration is with the value of the mask to supply. If you have read Terry Dawson's excellent NET−3−HOWTO, it should not be difficult to guess the correct value, though. As explained there, the netmask is a 32 bit number filled with all−1s in the network part of your IP address, and with all−0s in the host part. Recall from section 2.1 that a class D multicast address has no netwok/host sections. Instead it has a 28−bit group identifier and a 4−bit class D identifier. Well, this 4 bits are the network part and the remaining 28 the host part. So the netmask needed is 11110000000000000000000000000000 or, easier to read: 240.0.0.0. Then, the full command should be:

```
route add 224.0.0.0 netmask 240.0.0.0 dev eth0
```

Depending on how old your `route` program is, you might need to add the `-net` flag after the `add`.

Here we supposed that `eth0` was multicast−capable and that, when not otherwise specified, we wanted multicast traffic to be output there. If this is not your case, change the `dev` parameter as appropriate.

The `/proc` filesystem proves here to be useful once again: you can check `/proc/net/igmp` to see the groups your host is currently subscribed to.

4.The MBone.

Using a new technology usually carries some advantages and disadvantages. The advantages of multicast are −I think− clear. The main disadvantage is that hundreds of hosts and, specially, routers don't support it yet. As a consequence, people who started working on multicast, bought new equipment, modified their operating systems, and built *multicast islands* in their local places. Then they discovered that it was difficult to communicate with people doing similar things because if only one of the routers between them didn't support multicast there was nothing to do...

The solution was clear: they decided to build a *virtual multicast network* in the top of the Internet. That is: sites with multicast routers between them could communicate directly. But sites joined across unicast routers would send their island's multicast traffic encapsulated in unicast packets to other multicast islands. Routers in the middle would not have problems, as they would be dealing with unicast traffic. Finally, in the receiving site, traffic would be de−encapsulated, and sent to the island in the original multicast way. Two ends converting from multicast to unicast, and then again to multicast define what is called a multicast *tunnel*.

The *MBone* or *Multicast Backbone* is that virtual multicast network based on multicast islands connected by multicast tunnels.

Several activities take place in the MBone daily, but it deserves to be remarked the profusion of tele−conferences with real time audio and video taking place across the whole Internet. As an example, it was recently transmitted (live) the talk Linus Torvalds gave to the Silicon Valley Linux Users Group.

For more information on the MBone, see:

http://www.mediadesign.co.at/newmedia/more/mbone−faq.html

5.Multicast applications.

Most people dealing with multicast, sooner or later decide to connect to the MBone, and then they usually need an `mrouted`. You'll also need it if you don't have a multicast−capable router and you want multicast traffic generated in one of your subnets to be "heard" on another. `mrouted` does circunvect the problem of sending multicast traffic across unicast routers −it encapsulates multicast datagrams into unicast ones (IP into IP)− but this is not the only feature it provides. Most important, it instructs the kernel on how to route (or not−to−route) multicast datagrams based on their source and destination. So, even having a multicast capable router, `mrouted` can be used to tell it *what* to do with the datagrams (note I said *what*, and not *how*; `mrouted` says "forward this to the network connected to that interface", but actual forwarding is performed by the kernel). This distinction between actual−forwarding and the algorithm that decides who and how to forward is very useful as it allows to write forwarding code only once and place it into the kernel. Forwarding algorithms and policies are then implemented in user space daemons, so it is very easy to change from one policy to another without the need of kernel re−compilation.

You can get a version of *mrouted* ported to Linux from:

ftp://www.video.ja.net/mice/mrouted/Linux/. This site is mirrored all across the world. Be sure to read the ftp://www.video.ja.net/mice/README.mirrors file to choose the one nearest you.

Next, we'll focus specially on multicast applications written to connect to the MBone, which have been ported to Linux. The list is picked up from Michael Esler's "Linux Multicast Information" page http://www.cs.virginia.edu/~mke2e/multicast/. I recommend you that page for lots of information and resources on multicast and Linux.

Audio Conferencing

- NeVoT − Network Voice Terminal http://www.fokus.gmd.de/step/nevot
- RAT − UCL Robust−Audio Tool http://www−mice.cs.ucl.ac.uk/mice/rat

- vat – LBL visual audio tool http://www−nrg.ee.lbl.gov/vat/

Video Conferencing

- ivs – Inria video conferencing system http://www.inria.fr/rodeo/ivs.html
- nv – Network video tool ftp://ftp.parc.xerox.com/pub/net−research/
- nv w/ Meteor – Release of nv w/ support for the Matrox Meteor (UVa) ftp://ftp.cs.virginia.edu/pub/gwtts/Linux/nv−meteor.tar.gz
- vic – LBL video conferencing tool http://www−nrg.ee.lbl.gov/vic/
- vic w/ Meteor – Release of vic w/ support for the Matrox Meteor (UVa) ftp://ftp.cs.virginia.edu/pub/gwtts/Linux/vic2.7a38−meteor.tar.gz

Other Utilities

- mmphone Multimedia phone service http://www.eit.com/software/mmphone/phoneform.html
- wb – LBL shared white board http://www−nrg.ee.lbl.gov/wb/
- webcast – Reliable multicast application for linking Mosaic browsers http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/webcast.html

Session Tools

I placed session tools later because I think they deserve some explanation. When a conference takes places, several multicast groups and ports are assigned to each service you want for your conference (audio, video, shared white−boards, etc...) Announces of the conferences that will take place, along with information on multicast groups, ports and programs that will be used (vic, vat, ...) are periodically multicasted to the MBone. Session tools "hear" this information and present you in an easy way which conferences are taking (or will take) place, so you can decide which interest you. Also, they facilitate the task of joining a session. Instead of launching each program that will be used and telling which multicast group/port to join, you usually just need to click and the session tool launches the proper programs suppling them all information needed to join the conference. Session tools usually let you announce your own conferences on the MBone.

- gwTTS – University of Virginia tele−tutoring system http://www.cs.Virginia.EDU/~gwtts
- isc – Integrated session controller http://www.fokus.gmd.de/step/isc
- mmcc – Multimedia conference control ftp://ftp.isi.edu/confctrl/mmcc
- sd – LBL session directory tool ftp://ftp.ee.lbl.gov/conferencing/sd
- sd−snoop – Tenet Group session directory snoop utility ftp://tenet.berkeley.edu/pub/software
- sdr – UCL's next generation session directory ftp://cs.ucl.ac.uk/mice/sdr

---

6.Multicast programming.

Multicast programming... or writing your own multicast applications.

Several extensions to the programming API are needed in order to support multicast. All of them are handled via two system calls: `setsockopt()` (used to pass information to the kernel) and `getsockopt()` (to retrieve information regarded multicast behavior). This does *not* mean that 2 new system calls were added to support multicast. The pair `setsockopt()`/`getsockopt()` has been there for years. Since 4.2 BSD at least. The addition consists on a new set of options (multicast options) that are passed to these system calls, that the kernel must understand.

The following are the `setsockopt()`/`getsockopt()` function prototypes:

```
int getsockopt(int s, int level, int optname, void* optval, int* optlen);

int setsockopt(int s, int level, int optname, const void* optval, int optlen);
```

The first parameter, `s`, is the socket the system call applies to. For multicasting, it must be a socket of the family `AF_INET` and its type may be either `SOCK_DGRAM` or `SOCK_RAW`. The most common use is with `SOCK_DGRAM` sockets, but if you plan to write a routing daemon or modify some existing one, you will probably need to use `SOCK_RAW` ones.

The second one, `level`, identifies the layer that is to handle the option, message or query, whatever you want to call it. So, `SOL_SOCKET` is for the socket layer, `IPPROTO_IP` for the IP layer, etc... For multicast programming, `level` will always be `IPPROTO_IP`.

`optname` identifies the option we are setting/getting. Its value (either supplied by the program or returned by the kernel) is `optval`. The optnames involved in multicast programming are the following:

	setsockopt()	getsockopt()
IP_MULTICAST_LOOP	yes	yes
IP_MULTICAST_TTL	yes	yes
IP_MULTICAST_IF	yes	yes
IP_ADD_MEMBERSHIP	yes	no
IP_DROP_MEMBERSHIP	yes	no

`optlen` carries the size of the data structure `optval` points to. Note that in `getsockopt()` it is a value–result rather than a value: the kernel writes the value of `optname` in the buffer pointed by `optval` and informs us of that value's size via `optlen`.

Both `setsockopt()` and `getsockopt()` return 0 on success and –1 on error.

6.1 IP_MULTICAST_LOOP.

You have to decide, as the application writer, whether you want the data you send to be looped back to your host or not. If you plan to have more than one process or user "listening", loopback must be enabled. On the other hand, if you are sending the images your video camera is producing, you probably don't want loopback, even if you want to see yourself on the screen. In that latter case, your application will probably receive the images from a device attached to the computer and send them to the socket. As the application already "has" that data, it is improbable it wants to receive it again on the socket. Loopback is by default enabled.

Regard that `optval` is a pointer. You can't write:

```
setsockopt(socket, IPPROTO_IP, IP_MULTICAST_LOOP, 0, 1);
```

to disable loopback. Instead write:

```
u_char loop;
setsockopt(socket, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, sizeof(loop));
```

and set `loop` to 1 to enable loopback or 0 to disable it.

To know whether a socket is currently looping−back or not use something like:

```
u_char loop;
int size;

getsockopt(socket, IPPROTO_IP, IP_MULTICAST_LOOP, &loop, &size)
```

6.2 IP_MULTICAST_TTL.

If not otherwise specified, multicast datagrams are sent with a default value of 1, to prevent them to be forwarded beyond the local network. To change the TTL to the value you desire (from 0 to 255), put that value into a variable (here I name it "ttl") and write somewhere in your program:

```
u_char ttl;
setsockopt(socket, IPPROTO_IP, IP_MULTICAST_TTL, &ttl, sizeof(ttl));
```

The behavior with `getsockopt()` is similar to the one seen on IP_MULTICAST_LOOP.

6.3 IP_MULTICAST_IF.

Usually, the system administrator specifies the default interface multicast datagrams should be sent from. The programmer can override this and choose a concrete outgoing interface for a given socket with this option.

```
struct in_addr interface_addr;
setsockopt (socket, IPPROTO_IP, IP_MULTICAST_IF, &interface_addr, sizeof(interface_addr));
```

>From now on, all multicast traffic generated in this socket will be output from the interface chosen. To revert to the original behavior and let the kernel choose the outgoing interface based on the system administrator's configuration, it is enough to call `setsockopt()` with this same option and `INADDR_ANY` in the interface field.

In determining or selecting outgoing interfaces, the following `ioctls` might be useful: `SIOCGIFADDR` (to get an interface's address), `SIOCGIFCONF` (to get the list of all the interfaces) and `SIOCGIFFLAGS` (to get an interface's flags and, thus, determine whether the interface is multicast capable or not –the `IFF_MULTICAST` flag–).

If the host has more than one interface and the IP_MULTICAST_IF option is not set, multicast transmissions are sent from the default interface, although the remainding interfaces might be used for multicast *forwarding* if the host is acting as a multicast router.

6.4 IP_ADD_MEMBERSHIP.

Recall that you need to tell the kernel which multicast groups you are interested in. If no process is interested in a group, packets destined to it that arrive to the host are discarded. In order to inform the kernel of your interests and, thus, become a member of that group, you should first fill a `ip_mreq` structure which is passed later to the kernel in the `optval` field of the `setsockopt()` system call.

The ip_mreq structure (taken from `/usr/include/linux/in.h`) has the following members:

```
struct ip_mreq
{
        struct in_addr imr_multiaddr;   /* IP multicast address of group */
        struct in_addr imr_interface;   /* local IP address of interface */
};
```

(Note: the "physical" definition of the structure is in the file above specified. Nonetheless, you should not include `<linux/in.h>` if you want your code to be portable. Instead, include `<netinet/in.h>` which, in turn, includes `<linux/in.h>` itself).

The first member, `imr_multiaddr`, holds the group address you want to join. Remember that memberships are also associated with interfaces, not just groups. This is the reason you have to provide a value for the second member: `imr_interface`. This way, if you are in a multihomed host, you can join the same group in several interfaces. You can always fill this last member with the wildcard address (`INADDR_ANY`) and then the kernel will deal with the task of choosing the interface.

With this structure filled (say you defined it as: `struct ip_mreq mreq;`) you just have to call `setsockopt()` this way:

```
setsockopt (socket, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(mreq));
```

Notice that you can join several groups to the same socket, not just one. The limit to this is `IP_MAX_MEMBERSHIPS` and, as of version 2.0.33, it has the value of 20.

6.5 IP_DROP_MEMBERSHIP.

The process is quite similar to joining a group:

```
struct ip_mreq mreq;
setsockopt (socket, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(mreq));
```

where `mreq` is the same structure with the same data used when joining the group. If the `imr_interface` member is filled with `INADDR_ANY`, the first matching group is dropped.

If you have joined a lot of groups to the same socket, you don't need to drop memberships in all of them in order to terminate. When you close a socket, all memberships associated with it are dropped by the kernel. The same occurs if the process that opened the socket is killed.

Finally, keep in mind that a process dropping membership for a group does not imply that the host will stop receiving datagrams for that group. If another socket joined that group in that same interface previously to this `IP_DROP_MEMBERSHIP`, *the host* will keep being a member of that group.

Both ADD_MEMBERSHIP and DROP_MEMBERSHIP are nonblocking operations. They should return immediately indicating either success or failure.

7.The internals.

This section's aim is to provide some information, not needed to reach a basic understanding on how multicast works nor to be able to write multicast programs, but which is very interesting, gives some insight on the underlying multicast protocols and implementations, and may be useful to avoid common errors and misunderstandings.

7.1 IGMP.

When talking about `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP`, we said that the information provided by this "commands" was used by the kernel to choose which multicast datagrams accept or discard. This is true, but it is not all the truth. Such a simplification would imply that multicast datagrams for *all* multicast groups around the world would be received by our host, and then it would check the memberships issued by processes running on it to decide whether to pass the traffic to them or to throw it out. As you can imagine, this is a complete bandwidth waste.

What actually happens is that hosts instruct their routers telling them which multicast groups they are interested in; then, those routers tell their up–stream routers they want to receive that traffic, and so on. Algorithms employed for making the decision of *when* to ask for a group's traffic or saying that it is not desired anymore, vary a lot. There's something, however, that never changes: *how* this information is transmitted. **IGMP** is used for that. It stands for Internet Group Management Protocol. It is a new protocol, similar in many aspects to ICMP, with a protocol number of 2, whose messages are carried in IP datagrams, and which all level 2−compliant host are required to implement.

As said before, it is used both by hosts giving membership information to its routers, and by routers to communicate between themselves. In the following I'll cover only the hosts−routers relationships, mainly because I was unable to find information describing router to router communication other than the mrouted source code (rfc 1075 describing the Distance Vector Multicast Routing Protocol is now obsoleted, and `mrouted` implements a modified DVMRP not yet documented).

IGMP version 0 is specified in RFC−988 which is now obsoleted. Almost no one uses version 0 now.

IGMP version 1 is described in RFC−1112 and, although it is updated by RFC−2236 (IGMP version 2) it is in wide use still. The Linux kernel implements the full IGMP version 1 and parts of version 2 requirements, but not all.

Now I'll try to give an informal description of the protocol. You can check RFC−2236 for an in−proof formal description, with lots of state diagrams and time−out boundaries.

All IGMP messages have the following structure:

```
  0                   1                   2                   3
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |      Type     | Max Resp Time |           Checksum            |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |                         Group Address                         |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

IGMP version 1 (hereinafter IGMPv1) labels the "Max Resp Time" as "Unused", zeroes it when sent, and ignores it when received. Also, it brakes the "Type" field in two 4−bits wide fields: "Version" and "Type". As IGMPv1 identifies a "Membership Query" message as 0x11 (version 1, type 1) and IGMPv2 as 0x11 too, the 8 bits have the same effective interpretation.

I think it is more instructive to give first the IGMPv1 description and next point out the IGMPv2 additions, as they are mainly that, additions.

For the following discussions it is important to remember that multicast routers receive *all* IP multicast datagrams.

IGMP version 1.

Routers periodically send *IGMP Host Membership Queries* to the all−hosts group (224.0.0.1) with a TTL of 1 (once every minute or two). All multicast−capable hosts hear them, but don't answer immediately to avoid an IGMP Host Membership Report storm. Instead, they start a random delay timer for each group they belong to *on the interface* they received the query.

Sooner or later, the timer expires in one of the hosts, and it sends an IGMP *Host Membership Report* (also with TTL 1) to the multicast address of the group being reported. As it is sent to the group, all hosts that joined the group −and which are currently waiting for their own timer to expire− receive it, too. Then, they stop their timers and don't generate any other report. Just one is generated −by the host that chose the smaller timeout−, and that is enough for the router. It only needs to know that there are members for that group in the subnet, not how many nor which.

When no reports are received for a given group after a certain number of queries, the router assumes that no members are left, and thus it doesn't have to forward traffic for that group on that subnet. Note that in IGMPv1 there are no "Leave Group messages".

When a host joins a *new* group, the kernel sends a report for that group, so that the respective process needs not to wait a minute or two until a new membership query is received. As you can see this IGMP packet is generated by the kernel as a response to the `IP_ADD_MEMBERSHIP` command, seen in section IP_ADD_MEMBERSHIP. Note the emphasis in the adjective "new": if a process issues an `IP_ADD_MEMBERSHIP` command for a group the host is already a member of, no IGMP packets are sent as we must already be receiving traffic for that group; instead, a counter for that group's use is incremented. `IP_DROP_MEMBERSHIP` generates no datagrams in IGMPv1.

Host Membership Queries are identified by Type 0x11, and Host Membership Reports by Type 0x12.

No reports are sent for the all−hosts group. Membership in this group is permanent.

IGMP version 2.

One important addition to the above is the inclusion of a *Leave Group* message (Type 0x17). The reason is to reduce the bandwidth waste between the time the last host in the subnet drops membership and the time the router times−out for its queries and decides there are no more members present for that group (leave latency). Leave Group messages should be addressed to the all−routers group (224.0.0.2) rather than to the group being left, as that information is of no use for other members (kernel versions up to 2.0.33 send them to the group; although it does no harm to the hosts, it's a waste of time as they have to process them, but don't gain useful information). There are certain subtle details regarding when and when−not to send Leave Messages; if interested, see the RFC.

When an IGMPv2 router receives a Leave Message for a group, it sends *Group−Specific Queries* to the group being left. This is another addition. IGMPv1 has no group−specific queries. All queries are sent to the all−hosts group. The Type in the IGMP header does not change (0x11, as before), but the "Group Address" is filled with the address of the multicast group being left.

The "Max Resp Time" field, which was set to 0 in transmission and ignored on reception in IGMPv1, is meaningful only in "Membership Query" messages. It gives the maximum time allowed before sending a report in units of 1/10 second. It is used as a tune mechanism.

IGMPv2 adds another message type: 0x16. It is a "Version 2 Membership Report" sent by IGMPv2 hosts if they detect an IGMPv2 router is present (an IGMPv2 host knows an IGMPv1 router is present when it receives a query with the "Max Response" field set to 0).

When more than one router claims to act as querier, IGMPv2 provides a mechanism to avoid "discussions": the router with the lowest IP address is designed to be querier. The other routers keep timeouts. If the router with lower IP address crashes or is shutdown, the decision of who will be the querier is taken again after the timers expire.

7.2 Kernel corner.

This sub–section gives some start–points to study the multicast implementation of the Linux kernel. It does not explain that implementation. It just says where to find things.

The study was carried over version 2.0.32, so it could be a bit outdated by the time you read it (network code seems to have changed *A LOT* in 2.1.x releases, for instance).

Multicast code in the Linux kernel is always surrounded by `#ifdef CONFIG_IP_MULTICAST` / `#endif` pairs, so that you can include/ exclude it from your kernel based on your needs (this inclusion/exclusion is done at compile time, as you probably know if reading that section... `#ifdefs` are handled by the preprocessor. The decision is made based in what you selected when doing either a `make config`, `make menuconfig` or `make xconfig`).

You might want multicast features, but if your Linux box is not going to act as a multicast router you will probably not want multicast router features included in your new kernel. For this you have the multicast routing code surrounded by `#ifdef CONFIG_IP_MROUTE` / `#endif` pairs.

Kernel sources are usually placed in /usr/src/linux. However, the place may change so, both for accuracy and brevity, I will refer to the root directory of the kernel sources as just LINUX. Then, something like `LINUX/net/ipv4/udp.c` should be the same as `/usr/src/linux/net/ipv4/udp.c` if you unpacked the kernel sources in the `/usr/src/linux` directory.

All multicast interfaces with user programs shown in the section devoted to multicast programming were driven across the `setsockopt()`/ `getsockopt()` system calls. Both of them are implemented by means of functions that make some tests to verify the parameters passed to them and which, in turn, call another function that makes some additional tests, demultiplexes the call based on the `level` parameter to either system call, and then calls another function which... (if interested in all this jumps, you can follow them in `LINUX/net/socket.c` (functions `sys_socketcall()` and `sys_setsockopt()`, `LINUX/net/ipv4/af_inet.c` (function `inet_setsockopt()`) and `LINUX/net/ipv4/ip_sockglue.c` (function `ip_setsockopt()`)).

The one which interests us is `LINUX/net/ipv4/ip_sockglue.c`. Here we find `ip_setsockopt()` and `ip_getsockopt()` which are mainly a `switch` (after some error checking) verifying each possible value for `optname`. Along with unicast options, all multicast ones seen here are handled: `IP_MULTICAST_TTL`, `IP_MULTICAST_LOOP`, `IP_MULTICAST_IF`, `IP_ADD_MEMBERSHIP` and `IP_DROP_MEMBERSHIP`. Previously to the `switch`, a test is made to determine whether the options are multicast router specific, and if so, they are routed to the `ip_mroute_setsockopt()` and `ip_mroute_getsockopt()` functions (file `LINUX/net/ipv4/ipmr.c`).

In `LINUX/net/ipv4/af_inet.c` we can see the default values we talked about in previous sections (loopback enabled, TTL=1) provided when the socket is created (taken from function `inet_create()` in this file):

```
#ifdef CONFIG_IP_MULTICAST
        sk->ip_mc_loop=1;
        sk->ip_mc_ttl=1;
        *sk->ip_mc_name=0;
        sk->ip_mc_list=NULL;
#endif
```

Also, the assertion of "closing a socket makes the kernel drop all memberships this socket had" is corroborated by:

```
#ifdef CONFIG_IP_MULTICAST
                /* Applications forget to leave groups before exiting */
                ip_mc_drop_socket(sk);
#endif
```

taken from `inet_release()`, on the same file as before.

Device independent operations for the Link Layer are kept in `LINUX/net/core/dev_mcast.c`.

Two important functions are still missing: the processing of input and output multicast datagrams. As any other datagrams, incoming datagrams are passed from the device drivers to the `ip_rcv()` function (`LINUX/net/ipv4/ip_input.c`). In this function is where the perfect filtering is applied to multicast packets that crossed the devices layer (recall that lower layers only perform best−effort filtering and is IP who 100% knows whether we are interested in that multicast group or not). If the host is acting as a multicast router, this function decides too whether the datagram should be forwarded and calls `ipmr_forward()` appropriately. (`ipmr_forward()` is implemented in `LINUX/net/ipv4/ipmr.c`).

Code in charge of out−putting packets is kept in `LINUX/net/ipv4/ip_output.c`. Here is where the `IP_MULTICAST_LOOP` option takes effect, as it is checked to see whether to loop back the packets or not (function `ip_queue_xmit()`). Also the TTL of the outgoing packet is selected based on whether it is a multicast or unicast one. In the former case, the argument passed to the `IP_MULTICAST_TTL` option is used (function (`ip_build_xmit()`).

While working with `mrouted` (a program which gives the kernel information about how to route multicast datagrams), we detected that all multicast packets originated on the local network were properly routed..., except the ones from the Linux box that was acting as the multicast router!! ip_input.c was working OK, but it seemed ip_output.c wasn't. Reading the source code for the output functions, we found that outgoing datagrams were not being passed to `ipmr_forward()`, the function that had to decide whether they should be routed or not. The packets were outputed to the local network but, as network cards are usually unable to read their own transmissions, those datagrams were never routed. We added the necessary code to the `ip_build_xmit()` function and everything was OK again. (Having the sources for your kernel is not a luxury or pedantry; it's a need!)

`ipmr_forward()` has been mentioned a couple of times. It is an important function as it solves one important misunderstanding that appears to be widely expanded. When routing multicast traffic, it is *not*`mrouted` who makes the copies and sends them to the proper recipients. `mrouted` receives all multicast traffic and, based on that information, computes the multicast routing tables and *tells the kernel* how to route: "datagrams for this group coming from that interface should be forwarded to those interfaces". This information is passed to the kernel by calls to `setsockopt()` on a raw socket opened by the mrouted daemon (the protocol specified when the raw socket was created *must* be IPPROTO_IGMP). This options are handled in the ip_mroute_setsockopt() function from LINUX/net/ipv4/ipmr.c. The first option (would be better to call them commands rather than options) issued on that socket must be MRT_INIT. All other commands are ignored (returning -EACCES) if MRT_INIT is not issued first. Only one instance of mrouted can be running at the same time in the same host. To keep track of this, when the first MRT_INIT is received, an important variable, struct sock* mroute_socket, is pointed to the socket MRT_INIT was received on. If mroute_socket is not null when attending an MRT_INIT this means another mrouted is already running and -EADDRINUSE is returned. All resting commands (MRT_DONE, MRT_ADD_VIF, MRT_DEL_VIF, MRT_ADD_MFC, MRT_DEL_MFC and MRT_ASSERT) return -EACCES if they come from a socket different than mroute_socket.

As routed multicast datagrams can be received/sent across either physical interfaces or tunnels, a common abstraction for both was devised: VIFs, Virtual InterFaces. mrouted passes vif structures to the kernel, indicating physical or tunnel interfaces to add to its routing tables, and multicast forwarding entries saying where to forward datagrams.

VIFs are added with MRT_ADD_VIF and deleted with MRT_DEL_VIF. Both pass a struct vifctl to the kernel (defined in /usr/include/linux/mroute.h) with the following information:

```
struct vifctl {
        vifi_t  vifc_vifi;              /* Index of VIF */
        unsigned char vifc_flags;       /* VIFF_ flags */
        unsigned char vifc_threshold;   /* ttl limit */
        unsigned int vifc_rate_limit;   /* Rate limiter values (NI) */
        struct in_addr vifc_lcl_addr;   /* Our address */
        struct in_addr vifc_rmt_addr;   /* IPIP tunnel addr */
};
```

With this information a vif_device structure is built:

```
struct vif_device
{
        struct device   *dev;                   /* Device we are using */
        struct route    *rt_cache;              /* Tunnel route cache */
        unsigned long   bytes_in,bytes_out;
        unsigned long   pkt_in,pkt_out;         /* Statistics */
        unsigned long   rate_limit;             /* Traffic shaping (NI) */
        unsigned char   threshold;              /* TTL threshold */
        unsigned short  flags;                  /* Control flags */
        unsigned long   local,remote;           /* Addresses(remote for tunnels)*/
};
```

Note the dev entry in the structure. The device structure is defined in /usr/include/linux/netdevice.h file. It is a big structure, but the field that interests us is:

```
  struct ip_mc_list*    ip_mc_list;   /* IP multicast filter chain    */
```

The ip_mc_list structure –defined in /usr/include/linux/igmp.h– is as follows:

```
struct ip_mc_list
{
        struct device *interface;
        unsigned long multiaddr;
        struct ip_mc_list *next;
```

```
        struct timer_list timer;
        short tm_running;
        short reporter;
        int users;
};
```

So, the `ip_mc_list` member from the `dev` structure is a pointer to a linked list of `ip_mc_list` structures, each containing an entry for each multicast group the network interface is a member of. Here again we see membership is associated to interfaces. `LINUX/net/ipv4/ip_input.c` traverses this linked list to decide whether the received datagram is destined to any group the interface that received the datagram belongs to:

```
#ifdef CONFIG_IP_MULTICAST
                if(!(dev->flags&IFF_ALLMULTI) && brd==IS_MULTICAST
                   && iph->daddr!=IGMP_ALL_HOSTS
                   && !(dev->flags&IFF_LOOPBACK))
                {
                        /*
                         *      Check it is for one of our groups
                         */
                        struct ip_mc_list *ip_mc=dev->ip_mc_list;
                        do
                        {
                                if(ip_mc==NULL)
                                {
                                        kfree_skb(skb, FREE_WRITE);
                                        return 0;
                                }
                                if(ip_mc->multiaddr==iph->daddr)
                                        break;
                                ip_mc=ip_mc->next;
                        }
                        while(1);
                }
#endif
```

The `users` field in the `ip_mc_list` structure is used to implement what was said in section IGMP version 1: if a process joins a group and the interface is already a member of that group (ie, another process joined that same group in that same interface before) only the count of members (`users`) is incremented. No IGMP messages are sent, as you can see in the following code (taken from `ip_mc_inc_group()`, called by `ip_mc_join_group()`, both in `LINUX/net/ipv4/igmp.c`):

```
        for(i=dev->ip_mc_list;i!=NULL;i=i->next)
        {
                if(i->multiaddr==addr)
                {
                        i->users++;
                        return;
                }
        }
```

When dropping memberships, the counter is decremented and additional operations are performed only when the count reaches 0 (`ip_mc_dec_group()`).

`MRT_ADD_MFC` and `MRT_DEL_MFC` set or delete forwarding entries in the multicast routing tables. Both pass a `struct mfcctl` to the kernel (also defined in `/usr/include/linux/mroute.h`) with this information:

```
struct mfcctl
{
        struct in_addr mfcc_origin;            /* Origin of mcast      */
        struct in_addr mfcc_mcastgrp;          /* Group in question    */
        vifi_t  mfcc_parent;                   /* Where it arrived     */
        unsigned char mfcc_ttls[MAXVIFS];      /* Where it is going    */
};
```

With all this information in hand, `ipmr_forward()` "walks" across the VIFs, and if a matching is found *it* duplicates the datagram and calls `ipmr_queue_xmit()` which, in turn, uses the output device specified by the routing table and the proper destination address if the packet is to be sent across a tunnel (ie, the unicast destination address of the other end of the tunnel).

Function `ip_rt_event()` (not directly related to output, but which is in ip_output.c too) receives events related to a network device, like the device going up. This function assures that then the device joins the ALL−HOSTS multicast group.

IGMP functions are implemented in `LINUX/net/ipv4/igmp.c`. Important information for that functions appears in `/usr/include/linux/igmp.h` and `/usr/include/linux/mroute.h`. The IGMP entry in the `/proc/net` directory is created with `ip_init()` in `LINUX/net/ipv4/ip_output.c`.

8.Routing Policies and Forwarding Techniques.

One trivial algorithm to make worldwide multicast traffic available everywhere could be to send it... everywhere, despite someone wants it or not. As this does not seem quite optimized, several routing algorithms and forwarding techniques have been implemented.

DVMRP (Distance Vector Multicast Routing Protocol) is, perhaps, the one most multicast routers use now. It is a *dense mode* routing protocol, that is, it performs well in environments with high bandwidth and densely distributed members. However, in *sparse mode* scenarios, it suffers from scalability problems.

Together with DVMRP we can find other dense mode routing protocols, such as **MOSPF** (Multicast Extensions to OSPF −Open Shortest Path First−) and PIM−DM (Protocol−Independent Multicast Dense Mode).

To perform routing in sparse mode environments, we have **PIM−SM** (Protocol Independent Multicast Sparse Mode) and **CBT** (Core Based Trees).

OSPF version 2 is explained in RFC 1583, and MOSPF in RFC 1584. PIM−SM and CBT specifications can be found in RFC 2117 and 2201, respectively.

All this routing protocols use some type of multicast forwarding, such as *flooding*, *Reverse Path Broadcasting* (RPB), *Truncated Reverse Path Broadcasting* (TRPB), *Reverse Path Multicasting* (RPM) or *Shared Trees*.

It would be too long to explain them here and, as short descriptions for them are publicly available, I'll just recommend reading the `draft-ietf-mboned-in.txt` text. You can find it in the same places RFCs are available, and it explains in some detail all the above techniques and policies.

9.Multicast Transport Protocols.

So far we have been talking about multicast transmissions using UDP. This is the usual practice, as it is impossible to do it with TCP. However, intense research is taking place since a couple of years in order to develop some new multicast transport protocols.

Several of these protocols have been implemented and are being tested. A good lesson from them is that it seems no multicast transport protocol is general and good enough for all types of multicast applications.

If transport protocols are complex and difficult to tune, imagine dealing with delays (in multimedia conferences), data loss, ordering, retransmissions, flow and congestion control, group management, etc, when the receiver is not one, but perhaps hundreds or thousands of sparse hosts. Here scalability is an issue, and new techniches are implemented, such as not giving acknowledges for every packet received but, instead, send *negative acknowledges* (NACKs) for data not received. RFC 1458 gives the proposed requirements for multicast protocols.

Giving descriptions of those multicast protocols is out of the scope of this section. Instead, I'll give you the names of some of them and point you to some sources of information: Real−Time Transport Protocol (RTP) is concerned with multi−partite multimedia conferences, **Scalable Reliable Multicast** (SRM) is used by the `wb` (the distributed White−Board tool, see section Multicast applications), **Uniform Reliable Group Communication Protocol** (URGC) enforces reliable and ordered transactions based in a centralized control, **Muse** was developed as an application specific protocol: to multicast news articles over the MBone, the **Multicast File Transfer Protocol** (MFTP) is quite descriptive by itself and people "join" to file transmission (previously announced) much in the same way they would join a conference, **Log−Based Receiver−reliable Multicast** (LBRM) is a curious protocol that keeps track of all packets sent in a logging server that tells the sender whether it has to retransmit the data or can drop it safely as all receivers got it. One protocol with a funny name −especially for a multicast protocol− is STORM (**STructure−Oriented Resilient Multicast**). Lots and lots of multicast protocols can be found searching the Web, along with some interesting papers proposing new activities for multicast (for instance, www page distribution using multicast).

A good page providing comparisons between reliable multicast protocols is

http://www.tascnets.com/mist/doc/mcpCompare.html.

A very good and up−to−date site, with lots of interesting links (Internet drafts, RFCs, papers, links to other sites) is:

http://research.ivv.nasa.gov/RMP/links.html.

http://hill.lut.ac.uk/DS−Archive/MTP.html is also a good source of information on the subject.

Katia Obraczka's "*Multicast Transport Protocols: A Survey and Taxonomy*" article gives short descriptions for each protocol and tries to classify them according to different features. You can read it in the IEEE Communications magazine, January 1998, vol. 36, No. 1.

10.References.

10.1 RFCs.

- RFC 1112 "Host Extensions for IP Multicasting". Steve Deering. August 1989.
- RFC 2236 "Internet Group Management Protocol, version 2". W. Fenner. November 1997.
- RFC 1458 "Requirements for Multicast Protocols". Braudes, R and Zabele, S. May 1993.
- RFC 1469 "IP Multicast over Token−Ring Local Area Networks". T. Pusateri. June 1993.
- RFC 1390 "Transmission of IP and ARP over FDDI Networks". D. Katz. January 1993.
- RFC 1583 "OSPF Version 2". John Moy. March 1994.
- RFC 1584 "Multicast Extensions to OSPF". John Moy. March 1994.
- RFC 1585 "MOSPF: Analysis and Experience". John Moy. March 1994.
- RFC 1812 "Requirements for IP version 4 Routers". Fred Baker, Editor. June 1995
- RFC 2117 "Protocol Independent Multicast−Sparse Mode (PIM−SM): Protocol Specification". D. Estrin, D. Farinacci, A. Helmy, D. Thaler; S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. July 1997.
- RFC 2189 "Core Based Trees (CBT version 2) Multicast Routing". A. Ballardie. September 1997.
- RFC 2201 "Core Based Trees (CBT) Multicast Routing Architecture". A. Ballardie. September 1997.

10.2 Internet Drafts.

- "Introduction to IP Multicast Routing". `draft−ietf−mboned−intro− multicast−03.txt`. T. Maufer, C. Semeria. July 1997.
- "Administratively Scoped IP Multicast". `draft−ietf−mboned−admin−ip− space−03.txt`. D. Meyer. June 10, 1997.

10.3 Web pages.

- Linux Multicast Homepage. http://www.cs.virginia.edu/~mke2e/multicast.html
- Linux Multicast FAQ. http://andrew.triumf.ca/pub/linux/multicast−FAQ
- Multicast and MBONE on Linux. http://www.teksouth.com/linux/multicast/
- Christian Daudt's MBONE−Linux Page. http://www.microplex.com/~csd/linux/mbone.html

* Reliable Multicast Links http://research.ivv.nasa.gov/RMP/links.html
* Multicast Transport Protocols http://hill.lut.ac.uk/DS−Archive/MTP.html

10.4 Books.

* "TCP/IP Illustrated: Volume 1 The Protocols". Stevens, W. Richard. Addison Wesley Publishing Company, Reading MA, 1994
* "TCP/IP Illustrated: Volume 2, The Implementation". Wright, Gary and W. Richard Stevens. Addison Wesley Publishing Company, Reading MA, 1995
* "UNIX Network Programming Volume 1. Networking APIs: Sockets and XTI". Stevens, W. Richard. Second Edition, Prentice Hall, Inc. 1998.
* "Internetworking with TCP/IP Volume 1 Principles, Protocols, and Architecture". Comer, Douglas E. Second Edition, Prentice Hall, Inc. Englewood Cliffs, New Jersey, 1991

11.Copyright and Disclaimer.

If you publish this document on a CD−ROM or in hardcopy form, a complimentary copy would be appreciated; mail me for my postal address. Also consider making a donation to the Linux Documentation Project or the Free Software Foundation to help support free documentation for GNU/Linux. Contact the Linux HOWTO coordinator, Tim Bynum linux−howto@sunsite.unc.edu, for more information.

12.Acknowledgements.

This is the best opportunity I've ever had to thank so *many* people I feel grateful to. So, I'm afraid this is going to be a large section... It is, in any case, the most important one of this paper (for me, at least...).

First, I want to thank Elena Apolinario Fernández de Sousa (yes, Elena is the first name; the REST is THE surname ;−)). I tried to reflect in this Howto all the knowledge I collected while working with her in connecting our Department to the MBone and debugging problems with locally generated CSCW software across multicast tunnels. She was of invaluable help in finding and correcting network problems, discovering and fixing kernel bugs that puzzled us for days, ... and keeping the sense of humor alive while problems appeared and appeared, but solutions didn't. She also read and corrected the drafts for this document and provided important ideas and suggestions. If this howto is here and is usefull for somebody, it will be, in many aspects, thanks to her. Thanks, Elena!

There is something I have been lucky enough to find all my (still−not−too−long) live, but, despite being repetitive, has never stopped amazing me. I'm talking about people that altruistically employ part of their time and/or resources to help other people learn new things; and, what is better, they enjoy doing it. This is not only (but also, too) explain things they already know, but lend their books, provide access to their sources and facilitate you the way to learn all things they know; sometimes, even more... I know quite a few of that people, and I'd like to thank them for all their help.

Pablo Basterrechea was my "first source of documentation" while I was in my pre−Internet stage. I learned assembly and advanced structured programming entirely from his books (well, the latter also from his programs...). Thanks for all, Pablo.

In my first course at the University that "primary source of documentation" moved to Pepe Mañas. He was teaching then Computer Programming there, and soon I became addict to his bookshelf. He lent me his books lots of times without asking for a minimum sign that could assure that I was going to return them back to him, not even my name! My first approach to TCP/IP was also by his hand: he lent me Comer's "Internetworking with TCP/IP, Volume 1" for the whole summer. He did not even know my name by then, but he lent me the book... That book influenced me a lot, and TCP/IP has become one of my primary fields of interest since that summer.

If there are two persons I must thank most, these are (in alphabetic order ;−)), José Manuel and Paco Moya. Nobody I asked more things more times (C, C++, Linux, security, Web, OSs, signals & systems, electronics, ... anything!) and, despite my persistence, I always got throughly and friendly responses and help. If I'm using GNU/Linux now, this is, again, thanks to them. I feel particularly lucky with friends like them. THANKS.

Iñigo Mascaraque also helped (from him I got my first System Administration book) and encouraged me in my beginnings, but never stopped reminding me that, although this was a fascinating world and an important part of my career, I should not forget the other, less−interesting, parts. (I don't forget, I$!).

As I am on the topic, I'd like to thank my parents, too. They always tried to make the best opportunities available for me. Many thanks for all.

I also feel grateful to Joaquín Seoane, the first who trusted me enough to give me a root password in the time I was learning system administration by myself, and Santiago Pavón, the one who gave me my first opportunity here at DIT.

W. Richard Stevens' books have been a real revelation for me (it's a pity they are so expensive...). If he ever reads this paper, I'd like to thank him for them, and encourage him to keep on writing. Anything that comes out of his hands will −undoubtedly− be good for all of us.

Finally I'd like to thank Richard Stallman, Linus Torvalds, Alan Cox and all contributors to the Linux kernel and the free software in general, for giving us such a great OS.

I'm sure I'm forgetting someone here... Sorry. I'm certain they know I'm grateful to them too, so if they tell me, everybody will know it... :−)