IP

The IP module implements layer 3 of the Internet Protocol. It processes IP headers on hehalf of transport srvices (e.g. TCP or UDP). It also contains the Internet Control Message Protocol (ICMP) implementation. This implementation of IP is intended for end-user applications using standard transport layers. The routing portion of the code would need to be considerably improved if this were the the basis of a layer-3 router with multiple interfaces in the core, or edge, of a large network.

Process Information

Prototype Name	ip
Link Order	does not matter
Process Name	“ip”
Prototype Name	icmp
Link Order	does not matter
Process Name	“icmp”

Configuration Commands

The IP process is controlled using three strings passed to it as *COMMAND* messages. It will not attempt to open dataflows to any output devices until it receives the appropriate configuration commands. All configuration data for the process is supplied from sources external to it, making the IP module portable across all implementatins.

config

config *ifno i1.i2.i3.i4 m1.m2.m3.m4 devname protocol*

The *config* command sets the local IP address and netmask for the interface numbered *ifno*. The netmask is used to determine the routing range of the interface. The interface is associated with the lower layer device *devname* and *protocol* selects IP packets within that layer. For the IP module to operate correctly over an ethernet, the downstream device must be capable of supplying the IP-address to MAC-address mapping information to complete the ethernet header. This usually means that interfaces are opened to the ‘earp’ process and not directly to the ethernet driver itself.

Example: “config 0 138.15.103.240 255.255.255.0 earp:0 2048”

route

route *i1.i2.i3.i4 m1.m2.m3.m4 g1.g2.g3.g4 ifno*

The *route* command defines a gateway on an attached interface network and the routing options for that subnet. IP addresses with the same masked subnet prefix as the IP address will be sent to the (local) machine specified by the gateway address on the *ifno* interface.

Example: “route 0.0.0.0 0.0.0.0 138.15.103.254 0”

remove

remove *ifno*

The *remove* command removes all routes associated with the *ifno* interface and closes the IP packer receiver file for that interface. The command is usually generated by an application in response to an event indicating that a removable interface is longer present. Any application-level files currently open to that device will not be able to send data until a new interface or route is defined to replace it.

Example: “remove 0”

Data Definitions

The file *ip.h* contains definitions for the **IP_HEADER**, **ICMP_HEADER**, **TCP_HEADER** and **UDP_HEADER** protocol headers. It also defines the **IP_STATS** and **UDP_STATS** structures used to hold traffic and error statistics. The file also contains a number of definitions for protocols and options for the various IP services.

Process Operation

The operation of the IP module can be divided into the IP layer-3 processing and the ICMP protocol processing parts.

IP Processing

The IP layer handles fragmentation, and insertion of IP options and so it terminates the application dataflow and copies the data to or from the buffers destined for the devices. This also has the usual side-effect of copying the data to or from an uncached buffer.

IP Addressing

In general, an IP packet requires two destination addresses. One is the final destination of the packet, and is placed in the main IP header. The other is the address for the ‘next-hop’. For machines on the local subnet, these addresses are the same. For machines on a different subnet, the next-hop IP address is the address of the gateway on the appropriate local subnet. When IP messages are passed downstream, the IP header contains the final address and the *b_immed* field of the first *mblk* in the chain contains the next-hop address. This is signalled by setting the *b_type* field of the *mblk* to *M_IPDATA.*

IP options

The IP code supports IP option processing, but does not itself take any action in response to options. When options are set, they are sent on the next available data packet, and the options are cleared. Options may be set in one of two ways:

1. as parameters on the URL passed to OPEN in the *urlpath* field.
2. as the data part of an *M_PROTO* mblk sent at the head of a frame (or as a frame by themselves).

In both cases the options are specified as a string in the form:

$$?option[= value,]...?option[= value]$$

where *option* is a two-character hexadecimal number and *value* is a string of byte in hexadecimal. The '?', '=' and ',' characters are literal separators provided to make the syntax more human-readable. In the IP layer the hex numbers are concatenated together into a single sequence of bytes and rounded to a multiple of 4 bytes with no-op codes. This means that options that have variable-length values must have the lengths correctly encoded in the string. Options received on incoming packets are decoded into the string form and passed up to the corresponding transport layer as separate *M_PROTO* messages. They are also included in the data message which follows in the original raw binary.

Other optional fields in the IP header, such as type-of-service and the do-not-fragment bit, are set by the application in the dummy IP header supplied for output buffer.

IP Fragmentation

The IP code assumes that local applications do not generate packets that require fragmentation, since this imposes an extra overhead on the dataflow. However, it does process incoming fragmented packets. Fragments are collected and passed up to the application as a single message block. If a fragmented packet contains options, only the options on the first packet of the fragment are passed to the application. The others are discarded when the fragments are concatenated.

Active and Passive Opens

Because the IP layer does not examine any of the transport-layer headers, it cannot de-multiplex packets for different transport-level applications. Instead it passes all packets for a single IP protocol-type upwards on a single file, created by sending an open request to the IP layer with no destination IP address. This is termed a passive (or listener) open. In order to transmit packets, the IP layer must know the destination address of the packet. This is determined when the file is opened. A file having a destination IP address set in its open URL is an active (sending) file. Packets are never passed upwards on active files, nor may packets be sent on passive files.

Errors

Most errors (for example incomplete fragments) are handled silently within the IP layer. The only case where a response is explicitly generated over the network is when a packet is received for a protocol for which there is no passive file currently open. The IP layer generates an ICMP 'no protocol' message back to the sender.

Messages

CLOSE messages are handled by the main process. All outstanding receive messages are returned to the sender and the local data structure is freed before the reply is generated.

COMMAND messages are handled in the main process according to the 'Configuration' section above.

FETMBLK messages are not supported by the IP process, because it will always deliver full IP packets upwards. The queue handler calls *rome_fatal* if an application sends it a *FETMBLK* request.

GETMBLK messages are added to the receive queue of the file. Replies to *GETMBLK* messages indicate incoming data from the interface. The data blocks are validated and passed upwards to the appropriate application, after option processing and defragmentation. The application receives the block with the pointer at the start of the IP header not, as might be expected, at the start of the transport header, since all current IP transport layers rely on fields provided only in the IP header.

FLUSH messages are replied to immediately from within the queue handler, as the IP layer does not buffer any outgoing data.

NEWMBLK messages are processed within the queue handler. A buffer of a suitable size is allocated and space reserved and cleared at the start for a standard IP header before being returned to the caller.

OPEN messages are handled in the main process. The *ipaddr* field of the URL determines if this is an active or passive open (see above) and the *port* field contains the IP protocol identifier. The contents of the *urlpath* field are used to set the initial IP options. If possible the route for an active open is determined and the interface is fixed. By the time a dataflow reaches IP, the destination machine's IP address should have been resolved (for example using the DNS or NIS services). The IP process does not look at the *host* field in the URL, only the *ipaddr* field.

OUTMBLK/PUTMBLK messages are of zero length are processed in the queue handler, where the buffer is freed. Otherwise the messages are passed to the main process. The message must have been sent on an active file for which there is a current routable interface, otherwise it is returned with an error. A new message is allocated, any IP options in an *M_PROTO* message are handled and the data portion is copied to a new buffer allocated by the driver in the new message. The IP header in the new buffer is completed using the data supplied on the *OPEN* and options set in the prototype header supplied in the message. The gateway field is set in the downstream message and the data are passed to the driver for transmission. replies to *PUTMBLK* messages indicate transmission of the buffer, and the message, which was allocated in the IP layer, is freed in the queue handler.

RETMBLK messages are processed in the queue handler where the data buffer is freed.

ICMP Processing

The ICMP service operates as a separate process, but is co-located with the IP module as it is a mandatory protocol for IP. The implementation is designed for end-user embedded systems, and so offers only minimal functionality.

GETMBLK messages are set up to the IP process when ICMP starts. Replies indicate incoming ICMP messages, of which all are ignored apart from ECHO requests, which generate the corresponding replies, so the machine is 'ping'able.

PUTMBLK messages are used to request ICMP protocol transmissions, usually to indicate non-existant services. In particular, UDP will generate unknown port messages so that 'traceroute' terminates correctly. The messages contain an incoming IP packet in the *b_cont* field and the ICMP *type* and *code* fields in *b_immed* and *b_immed1* respectively. Because the IP process uses the ICMP process to generate errors, the IP header is copied from the incoming message to temporary storage and the reply is sent before any other operations, to prevent IP/ICMP process deadlock. The process then opens a new ICMP-protocol file to the destination, formats and transmits the requested protocol packet and closes the file.

Shared Library Macros and Routines

icmp_report

void *icmp_report*(
ROME_MESSAGE **mptr,*
int *type,*
int *code)*

The *icmp_report* routine requests an ICMP control message be sent to the source of the IP packet passed in the *mptr* argument, with packet type *type* and sub-code *code*. The first part of the IP header may be returned to the sender in the ICMP message, depending on the message type.

ip_route

int *ip_route*(
uint *dest,*
uint **route*)

The *ip_route* routine sets the *route* parameter to the address of the local interface on which a packet to *dest* would be sent. The routine returns 0 if the packet would be routed, non-zero otherwise. The purpose of this routine is to allow transport layers to determine which local IP address will be set in the IP header when the packet is sent, and so compute the correct packet checksum.

ip_stats

IP_STATS **ip_stats*(**void**)

The *ip_stats* routine returns a pointer to the statistics structure maintained by the IP layer.