fakeip - IP Protocol and Network Emulator

This software simulates direct access to the IP layer. It has been tested on rabbit, a BSD unix system, but only uses standard functions so it should also work on other unixes. Programs using this software to communicate may be running on the same computer or on separate computers with real network access. It allows the programmer to control the reliability of the simulated network, but uses UDP for communications, so there will be some unreliability in connections between separate computers regardless of the programmer's settings.

An FIP address is 6 bytes long. The first four bytes are the same as the computer's real IP4 address. The last two bytes are from the UDP port number used to simulate the network adapter. On startup, the system chooses a port number based on the user's UID. If that port is not available it progresses through alternative port numbers in a uniform way.

This means that the first FIP-using process run by any user will nearly always have the same FIP address, as will the second, third, etc.

The function

```
string ip_address_to_string(byte * addr);
```

converts a 6 byte FIP address to a C++ string, e.g. "129.171.33.6.220.206"

An FIP packet consists of a 16 byte header followed by up to 65535 bytes of data. Transmission of very large packets will be extremely unreliable, anything above about 1500 bytes is not recommended. This is the data structure used to represent an FIP packet

```
struct ip_packet
{ byte source_addr[6];
  byte dest_addr[6];
  short int protocol;
  short int length;
  byte content[]; };
```

Protocol may be any value the programmer chooses, and length must be the number of bytes of content, not including the header.

To send data, the programmer must first create a packet using this function

```
ip_packet * create_ip_packet(int length);
```

which automatically allocates 12+length bytes. The programmer is responsible for setting the source_addr, dest_addr, and protocol fields. The system will delete packets once they have been transmitted. The programmer must never delete packets that have been given to the system for transmission, or attempt to access them ever again.

When a packet is received from the system, it is the programmer's responsibility to delete it after processing it. This is the function for deleting a received packet

```
void delete_ip_packet(ip_packet * p);
```

Do not use the C++ delete operator on FIP packets.

The FIP system is accessed through a C++ object of the class ip_system. It has no public data members, only methods. After creating an ip_system object, the programmer must use its start method to prepare it for use and connect to the network. Once started, the get_address method returns the system's own 6 byte FIP address (or NULL if a network connection could not be made)

Sample initialisation code:

```
void main()
{ ip_system net;
  net.start();
  byte * my_fip_addr = net.get_address();
  cout << "my FIP address is " <<
          ip_address_to_string(my_fip_addr) << "\n";
```

The array of bytes returned by get_address gives the correct values to be used as the source_addr when creating an FIP packet to transmit, e.g.

```
ip_packet * p = create_ip_packet(len);
for (int i=0; i<6; i+=1)
  p->source_addr[i] = my_fip_addr[i];
```

Once an FIP packet has been created and the header and content have been set, the transmit method is used to turn it over to the fip system:

```
net.transmit(p);
```

After this, do not attempt to access p again, and do not delete it. Transmission is usually instantaneous, as may be verified by the fact that the transmit_queue_length method always returns 0:

```
cout << net.transmit_queue_length() <<
        " packets still waiting for transmission\n";
```

When the FIP system receives a packet it is kept in a queue until the program requests it, so it is not necessary to be ready immediately, but the queue has a maximum capacity of 100 packets. The system does nothing to verify packets, everything that is received goes into the queue.

To request a packet from the received queue, use either the receive or the wait_to_receive method, they both return a pointer to an ip_packet object or NULL if none is available.

```
ip_packet * p = net.receive();
if (p!=NULL)
{ cout << "received something!\n";
```

The receive method does not wait. If no packets are already in the queue, it immediately returns NULL. Once a packet has been taken from the queue, it is the programmer's responsibility to delete it with delete_ip_packet.

The wait_to_receive method when used with no parameters behaves exactly like receive, except that it will wait an unlimited time until a packet is available (just like using cin to get user input), and will never return NULL:

```
ip_packet * p = net.wait_to_receive();
```

The third alternative is to use wait_to_receive with an int parameter. This is a time limit expressed in milliseconds (0.001 sec).

```
        ip_packet * p = net.wait_to_receive(100);
```

If there is already a packet in the queue, this returns immediately. Otherwise it will wait for up to one tenth of a second. If a packet arrives during this period, again it returns immediately. When the waiting period elapses, wait_to_receive returns NULL.

For testing and debugging, the set_trace_level method is useful.

```
        net.set_trace_level(0);
```

is the normal setting, it does nothing. After

```
        net.set_trace_level(1);
```

the system will automatically report every packet that is transmitted or received, printing the packet's header on standard output, like this:

```
    [sent 129.171.33.6.87.198 -> 129.171.33.6.220.206 proto 2 len 37]
    [rcvd 129.171.33.6.220.206 -> 129.171.33.6.87.198 proto 2 len 256]
```

After

```
        net.set_trace_level(2);
```

the system will display headers and contents of all packets. Bytes in the data content of the packet are printed as characters if they are in the normal range of ASCII printable characters (32 to 126), otherwise they are printed in decimal inside square brackets, so for example a `\n` character appears as [10]:

```
   [rcvd 129.171.33.6.220.206 -> 129.171.33.6.87.198 proto 2 len 4: yes[10]]
```

For realistic testing, the set_reliability method may be used to simulate normal IP packet loss. It takes a floating point parameter between 0.0 and 1.0.

```
        net.set_reliability(1.0);
```

is the default setting, and no packets are deliberately lost (but remember that communications between programs on different computers are still subject to genuine IP losses).

A setting of 0.5 means that half of all transmitted packets will not make it, and half of all received packets will be totally ignored. So if two communicating processes both set their reliability to 0.5, only about one quarter of packets will survive the whole journey. A setting of 0.0 means total loss.

The packet loss is random. If the setting is 0.9, then every packet transmitted or received is independently given a 0.1 probability of being thrown away.

The FIP system runs as a separate thread, started by the start method, and semaphores are in place where needed. If your own program also uses threads, you must make sure that main does not exit until everything is finished. To simplify this, the wait_until_dead method

```
        net.wait_until_dead();
```

simply waits until the FIP system exits, which never happens in normal use. So it is useful to have as the last statement in main.