Håvard Wanvik Stenersen

HAAVAWS

02/03-2018

INF3190

Differences between MIP and IPv4

MIP is not fundamentally different from IPv4. Both are network layer protocols, and can be used for the same purpose: facilitating communication between two end-points. While the implementation of MIP in this assignment included only the functionality of a link layer protocol, that is direct host-to-host communication, it can be extended to function as a network layer protocol by implementing routing, in fundamentally the same way IPv4 is used. The differences between IPv4 and MIP comes in the form of three main categories: complexity, performance and scope.

First off, complexity. The complexity of merely implementing IPv4 is higher than implementing MIP. The IPv4 header is larger, while normally 20 bytes, can reach the up to 15 times the size of the 4-byte MIP header, that is 60 bytes. This of course means that, even at its minimum size of 20 bytes, the IPv4 header includes a lot more information than the MIP header, and a lot more needs to be taken into account and implemented when working with IPv4. While the MIP header only includes the source and destination, as well as some type-bits, the payload length and its TTL, the IPv4 header in addition includes the protocol, identification and a field to help with fragmentation of packets, among several others. These all need to be handled appropriately when implementing communication using IPv4, which means the implementation immediately becomes more complex than a MIP implementation which basically only needs to worry about the source and destination of a packet. There is obviously a trade-off here though; While the protocol and implementation may be more complex, it allows for functionality that is simply impossible to implement with MIP, maybe most importantly fragmentation. With MIP, only single packets can reliably be transmitted, because there is no way to guarantee ordering of the packets, and it is virtually impossible to implement re-construction of a fragmented packet in the network layer using MIP.

The second category, performance, is likely the strongest advantage MIP has over IPv4, and it is related to the complexity mentioned earlier. The header is a very small. The advantages and disadvantages of a large or a small header in terms of complexity is one thing, but the performance gain may be significant with a smaller header. Depending on the application, this can improve performance by a large amount, especially if the application sends and receives many very small packets. For example a ping server receiving a large amount of traffic of only bare-bones packets, would likely see a significant boost in the performance of their network. In addition, while possibly negligible, the amount of time it takes to process a MIP packet is smaller than for an IPv4 packet, as there are fewer fields in the header that need to be handled. With a recent CPU, this difference is likely extremely small, but still worth noting. As a bonus, the addresses in MIP are smaller, allowing for more efficient storage of ARP caches.

The last category, scope, is where IPv4 has, in my opinion, the biggest advantage over MIP. Because the scope of MIP is very, very small. With an address space of 2^8, or 256 addresses, compared to IPv4's address space of 2^{32}, or 4 294 967 296, it really isn't any kind of scalable. Even IPv4 is struggling with a lacking address space, so MIP is really only usable for very small operations, which somewhat defeats the purpose of its advantages. When used in small operations, the gains from an increase in performance, and a decrease in complexity really isn't all that valuable. With a small number of machines, a high volume of traffic is less of an issue, and advantage of it being easy to implement doesn't shine as brightly as it could if anyone would be able to more easily set up the protocol on their own and participate in the network. The lack of

reliable framentation, ordering and reconstruction of packets also severely limits the scope of MIP as a network layer protocol. It means that anyone who would wish to use MIP for anything more than simple transfer of small amounts of data would have to employ a transport or application layer protocol implementing the features.

While MIP and IPv4 are fundamentally the same, i.e. they are both network layer protocols, they have very significant differences, and their purposes are clearly different. MIP is meant, as its name suggests, to be used in a small network. With a small address space and very little extra functionality, it really doesn't scale very well, reliably allowing only for small single packet communication, and connecting only a small amount of interfaces, but its bare-bones structure and simple implementation makes it usable for small networks. IPv4 on the other hand scales well, it has a large address space, even if its size has been lacking for some time now, and it implements functions allowing for reliable transfers of large chunks of data, making its scope a lot wider than that of MIP.

How to execute the program

Compilation using Makefile:

1. Make sure all necessary code and header files are in one directory.
    a) Necessary files to compile the MIP daemon:
    mip_daemon.c
    mip_daemon.h
    debug.c
    mip.c
    sockets.c
    b) Necessary files to compile client:
    client.c
    c) Necessary files to compile server:
    server.c
2. To compile all binaries at once, make sure the above files, as well as **Makefile** is in one directory.
3. Open terminal.
4. Navigate to the directory the files are located in.
5. Type **make** and hit enter.

Manual compilation using gcc:

1. Make sure the files necessary to compile the binary you want to compile are in one directory.
2. Open terminal
3. Navigate to the directory the files required for the binary you want to compile are in.

a) To compile the MIP daemon, use the following command:
gcc -std=gnu99 mip.c sockets.c debug.c mip_daemon.c -o mip_daemon

b) To compile the client, use the following command:
gcc -std=gnu99 client.c -o client

c) To compile the server, use the following
gcc -std=gnu99 server.c -o server

Executing the binaries:

1. To execute the **mip_daemon**:

   a) Open terminal.

   b) Navigate to the directory the **mip_daemon** binary is located in.

   c) Execute the binary by using a command on the following form:
   ./mip_daemon [-h][-d] <Socket_application> [MIP addresses ...]
   Here, **-h** and **-d** are optional arguments for printing a help message and activating logging during execution respectively. Note that the **-h** option will make the **mip_daemon** stop execution after printing the help message, no matter what the other arguments were.
   <Socket_application> is the path that you want IPC with the connected application to be performed over. Note that the same argument needs to be supplied to the application you want to connect to the **mip_daemon** when executing it.
   [MIP addresses ...] is exactly one unique MIP address per raw network interface available through the terminal you are executing the **mip_daemon** in.

   d) Example:
   Typing **ip a** in the terminal prints the following:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: B-eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc htb
state UP group default qlen 1000
    link/ether 8e:9c:64:c1:f1:b0 brd ff:ff:ff:ff:ff:ff
    inet 10.0.0.2/8 brd 10.255.255.255 scope global B-eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::8c9c:64ff:fec1:f1b0/64 scope link
       valid_lft forever preferred_lft forever
6: B-eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc htb
state UP group default qlen 1000
    link/ether ba:44:37:cb:c1:fa brd ff:ff:ff:ff:ff:ff
```

inet6 fe80::b844:37ff:fecb:c1fa/64 scope link
valid_lft forever preferred_lft forever

e) Here there are two available raw network interfaces, B-eth0, and B-eth1.

f) To execute the **mip_daemon**, one could use the following command:
./mip_daemon -d IPC 1 2

g) This will run the **mip_daemon** in debug mode, perform IPC on the path **IPC** and will associate its two raw network interfaces with the MIP address **1** and **2** respectively

2. To execute the **server**:

a) Run the **mip_daemon** by following the steps starting at 1.

b) Make sure no other **server** is already connected to the **mip_daemon**.

c) Open terminal.

d) Navigate to the directory the **mip_daemon** was executed from.

e) Make sure the **server** binary is in the directory.

f) Execute the server by using a command on the form:
./server [-h] <Socket_application>
Here, **-h** is an optional argument for printing a help message when executing the program. Note that the **-h** option will make the **server** stop executing after printing the help message, no matter what the argument is.
<Socket_application> is the path that you want IPC with the **mip_daemon** to be performed over. Note that this argument needs to be the same one used when executing the **mip_daemon**.

g) Example:
The mip_daemon was executed using the following command:
./mip_daemon -d IPC 1 2

h) To execute the server, use the following command:
./server IPC

i) This will run the server and connect it to the **mip_daemon**, and they will communicate via IPC over the path **IPC**.

3. To execute the **client**:

a) Run the **mip_daemon** by following the steps starting at 1.

b) Make sure no other **server** is already connected to the **mip_daemon**.

c) Open terminal

d) Navigate to the directory the **mip_daemon** was executed from.

e) Make sure the **client** binary is in the directory.

f) Execute the server by using a command on the form:

./client [-h] <destination_host> <message> <Socket_application>
Here, **-h** is an optional argument for printing a help message when executing the program. Note that the **-h** option will make the **client** stop executing after printing the help message, no matter what the argument is.
<destination_host> is the MIP address of the host you want to ping
<message> is the message you want to send along with the ping to the host specified in **<destination_host>**.
<Socket_application> is the path that you want IPC with the **mip_daemon** to be performed over. Note that this argument needs to be the same one used when executing the **mip_daemon**.

g) Example:
The **mip_daemon** was executed using the following command:
./mip_daemon -d IPC 1 2

h) To execute the client, you can use the following command:
./client 3 test IPC

i) This will execute the **client**, and send a ping message with the content test to the **mip_daemon**, which will attempt to forward it to a host directly connected to the host running this **mip_daemon**, and wait for a PONG response. The **client** will time out after 500 ms if no PONG response has been received by then.

Note that it is possible to execute a **client** while a **server** is connected to the **mip_daemon**. In this case, the **mip_daemon** will wait until the **server** has disconnected before attempting to ping the specified host.

Files included in the assignment delivery

1. Tar-ball haavaws.tgz, containing the following files:
   a) client.c
   b) debug.h
   c) Makefile
   d) mip.h
   e) mip_daemon.c
   f) mip_daemon.h
   g) server.c
   h) sockets.c
   i) readme.pdf

Graph of function calls at both the client and server side found on the next page. Graph for client side on the left, graph for server side on the right. Highres.

Client: msg s_mip
- socket(): c_sock
- setsockopt(timeout)
- connect(mip_daemon)
  - C1
- gettimeofday(): Start
- sendmsg(msg, s_mip)
  - C2
- recvmsg(c_sock): pong
  - c_sock! C3
- gettimeofday(): End
- End - Start: latency
- print: latency
- exit(SUCCESS)

Mip daemon: c_mip, c_mac
- setup_unix_socket()
  - socket(): un_sock
  - bind()
  - listen()
- setup_eth_sockets()
  - getifaddrs(): ifaddrs
  - for: ifaddrs ? AF_PACKET
    - socket(): eth_socks
    - bind()
- create_epoll_instance()
  - epoll_create(): epfd
  - epoll_ctl(ADD) → epfd: un_sock
  - for: eth_socks
    - epoll_ctl(ADD) → epfd
- signalfd(): sig_fd
- epoll_ctl(ADD) → epfd: sig_fd
- epoll_wait(epfd)
  - un_sock! C1
- accept(un_sock): conn_sock
- epoll_ctl(ADD) → epfd: conn_sock

Mip daemon cont.:
- epoll_wait(epfd)
  - conn_sock! C2
- recvmsg(conn_sock): msg, s_mip
- send_mip_packet(msg, s_mip)
  - c_mip ? NO
- send_mip_broadcast(s_mip): bcast
  - for: eth_socks
    - send_mip_packet(bcast, s_mip, b_pack)
      - construct_mip_packet(bcast, s_mip): b_pack
      - send(b_pack)
        - M1
- epoll_wait(epfd)
  - eth_sock! M2
- recv_mip_packet(eth_sock)
  - recv(eth_sock): s_mip, s_mac
  - update_mip_arp(s_mip, s_mac)
- send_mip_packet(msg, s_mip)
  - construct_mip_packet(msg, s_mip): pack
  - send(pack)
    - M3
- epoll_wait(epfd)
  - eth_sock! M4
- recv_mip_packet(eth_sock)
  - recv(eth_sock): pong
  - sendmsg(pong)
    - C3

Mip daemon cont.
- epoll_wait(epfd)
  - sig_fd! Ctrl-c
- exit(SUCCESS)

Server: pong
- socket(): s_sock
- connect(s_sock)
  - S1
- recvmsg(s_sock): msg
  - s_sock! S2
- sendmsg(s_sock, pong)
  - S3
- recvmsg(s_sock)
  - Ctrl-C: EINTR
- exit(SUCCESS)

Mip daemon: s_mip, s_mac
- setup_unix_socket()
  - socket(): un_sock
  - bind()
  - listen()
- setup_eth_sockets()
  - getifaddrs(): ifaddrs
  - for: ifaddrs ? AF_PACKET
    - socket(): eth_socks
    - bind()
- create_epoll_instance
  - epoll_create(): epfd
  - epoll_ctl(ADD) → epfd: un_sock
  - for: eth_socks
    - epoll_ctl(ADD) → epfd
- signalfd(): sig_fd
- epoll_ctl(ADD) → epfd: sig_fd
- epoll_wait(epfd)
  - un_sock! S1
- accept(un_sock): conn_sock
- epoll_ctl(ADD) → epfd: conn_sock
- epoll_wait(epfd)
  - eth_sock! M1
- recv_mip_packet(eth_sock)
  - recv(eth_sock): c_mip, c_mac
  - update_mip_arp(c_mip, c_mac)
  - send_mip_packet(s_mac)
    - construct_mip_packet(s_mac)
    - send(s_mac)
      - M2

Mip daemon cont.
- epoll_wait(epfd)
  - eth_sock! M3
- recv_mip_packet(eth_sock)
  - recv(eth_sock): c_mip, msg
- sendmsg(sock_conn, msg)
  - S2
- recvmsg(sock_conn): pong
  - sock_conn! S3
- send_mip_packet(c_mip, pong)
  - construct_mip_packet(c_mip, pong)
  - send(pong)
    - M4
- epoll_wait(epfd)
  - sig_fd! Ctrl-c
- exit(SUCCESS)