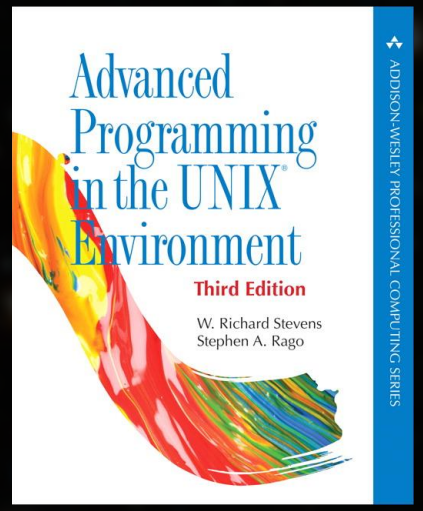
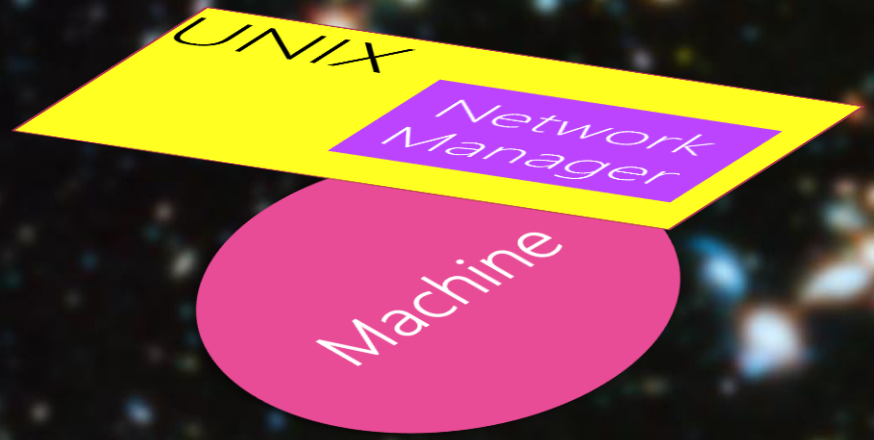
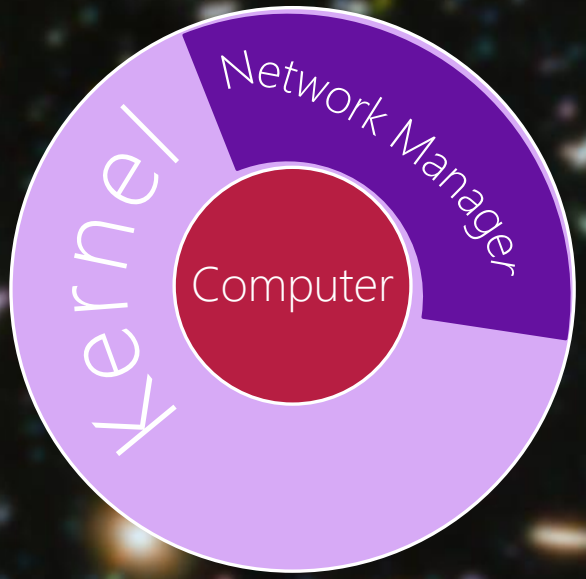


Chapter 16: Network IPC (Sockets)





Computer

Memory

OS Kernel: Device Manager

OS Kernel: Memory Manager

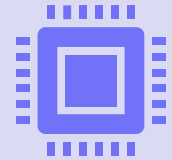
Kernel: File Manager

Kernel: Network Manager

Kernel: Process Manager

Bus

Processor



Multi~~processing~~ Computers

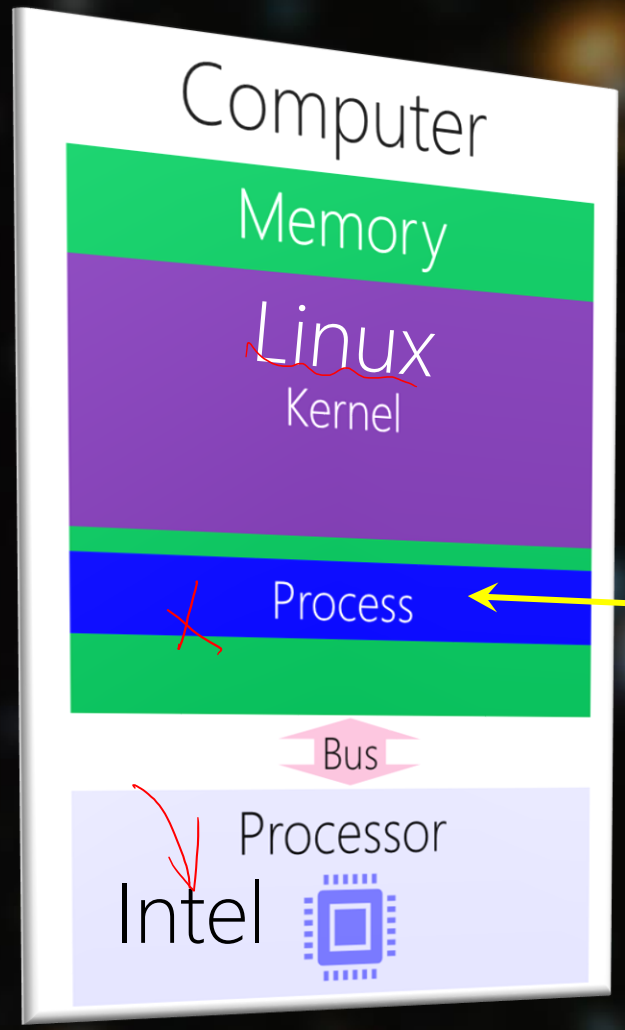
aka Computer Network

Multiple Single Processor ~~Multiprocessor~~

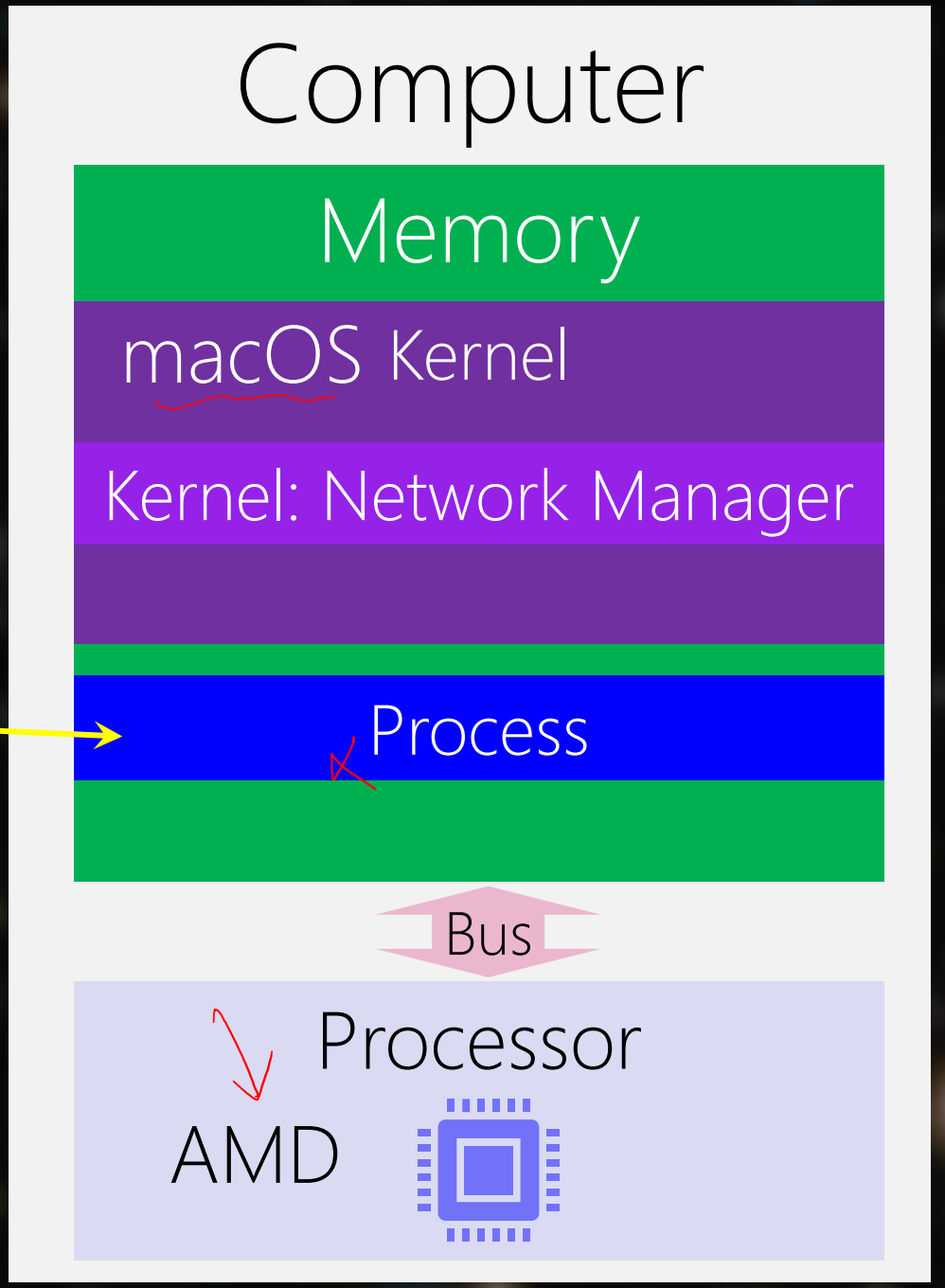
Step outside the computer system. Observe the World!

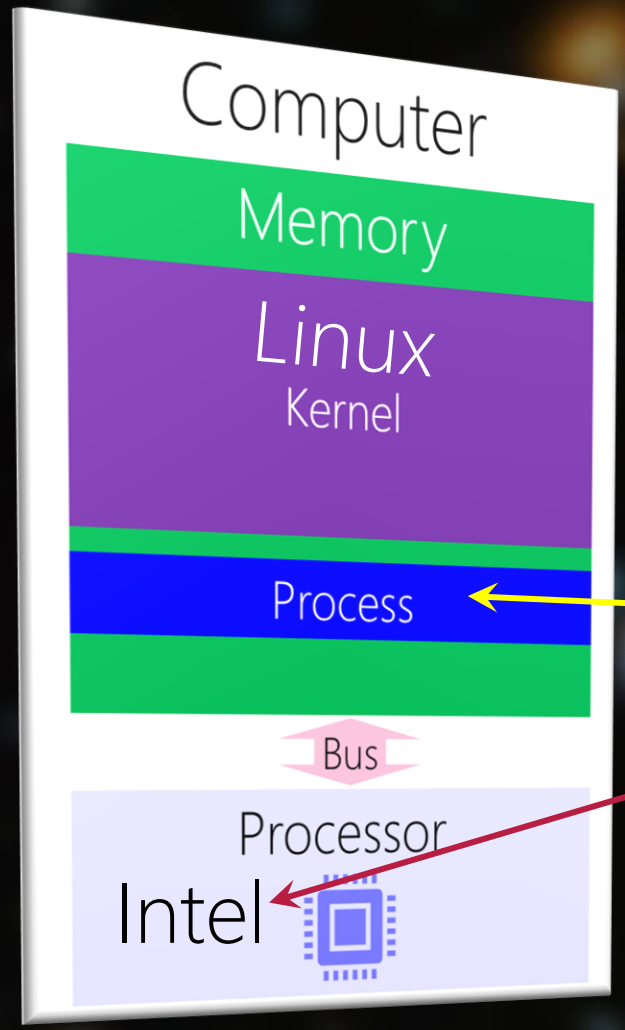
IPC: Family: Parent and Children

Network IPC: Families of a Society

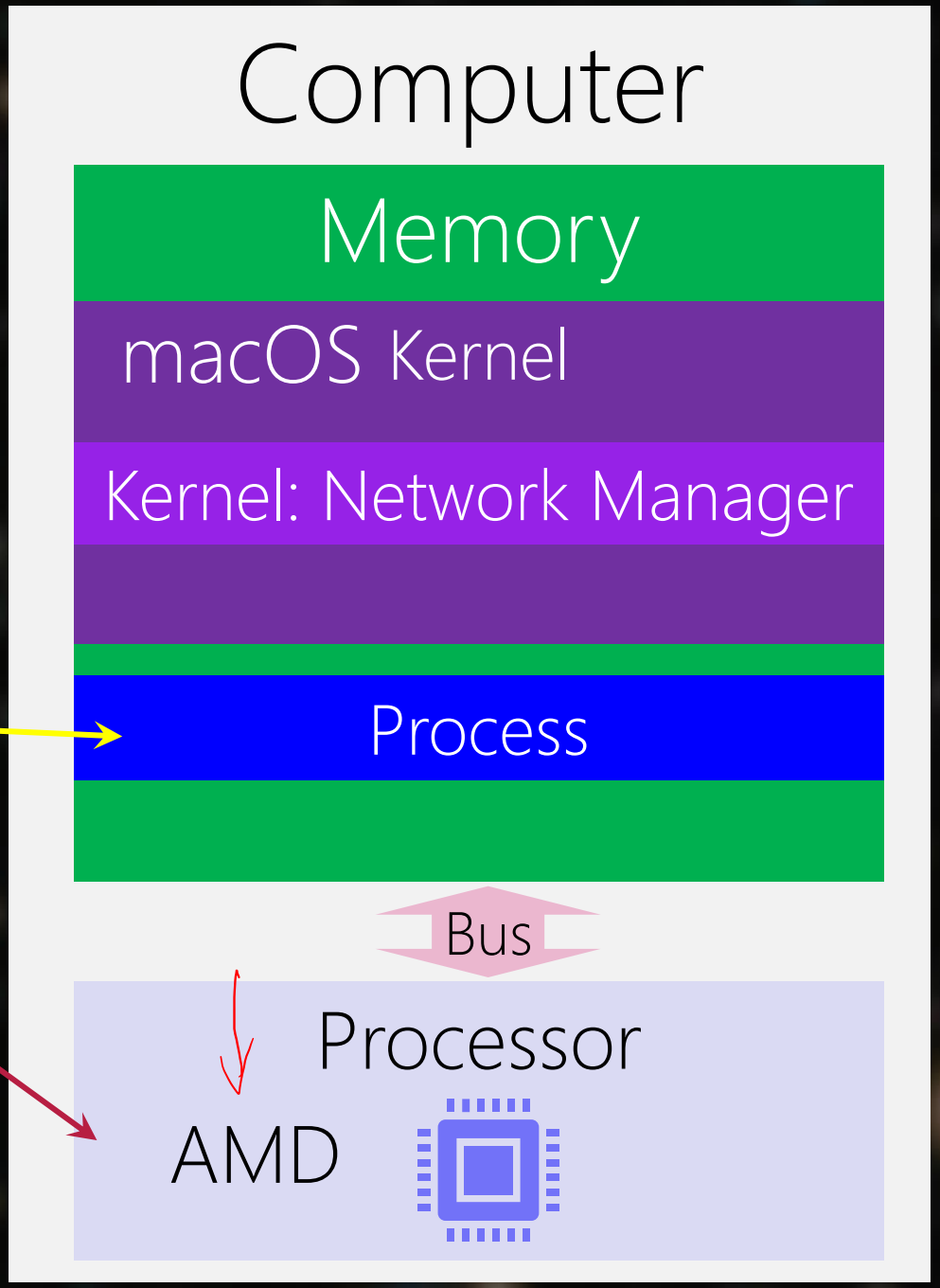

Computer
Memory
Linux
Kernel
Process
Bus
Processor
Intel

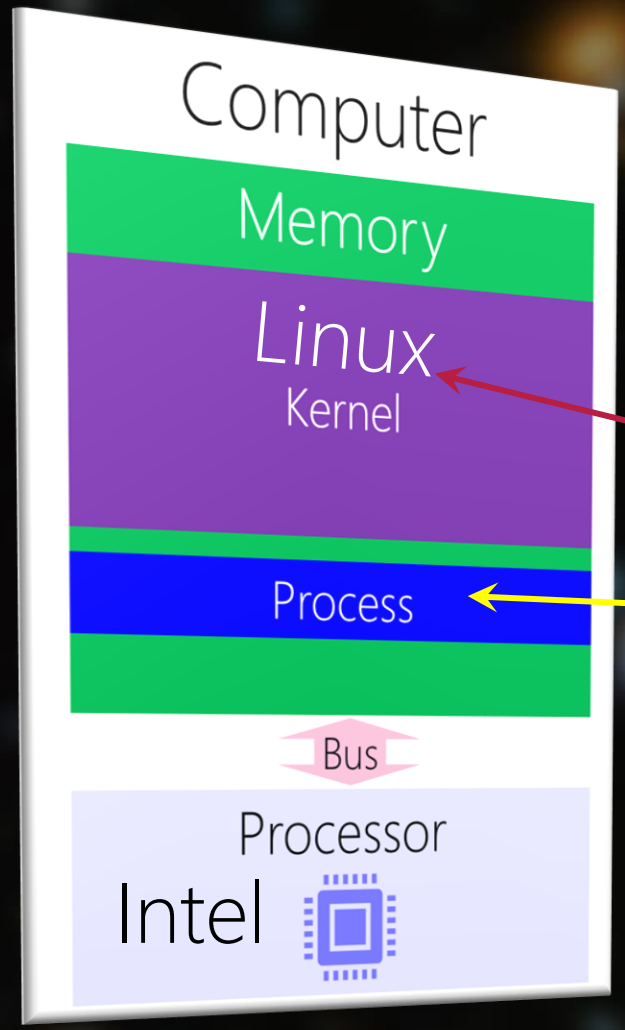
Network IPC


Computer
Memory
macOS Kernel
Kernel: Network Manager
Process
Bus
Processor
AMD

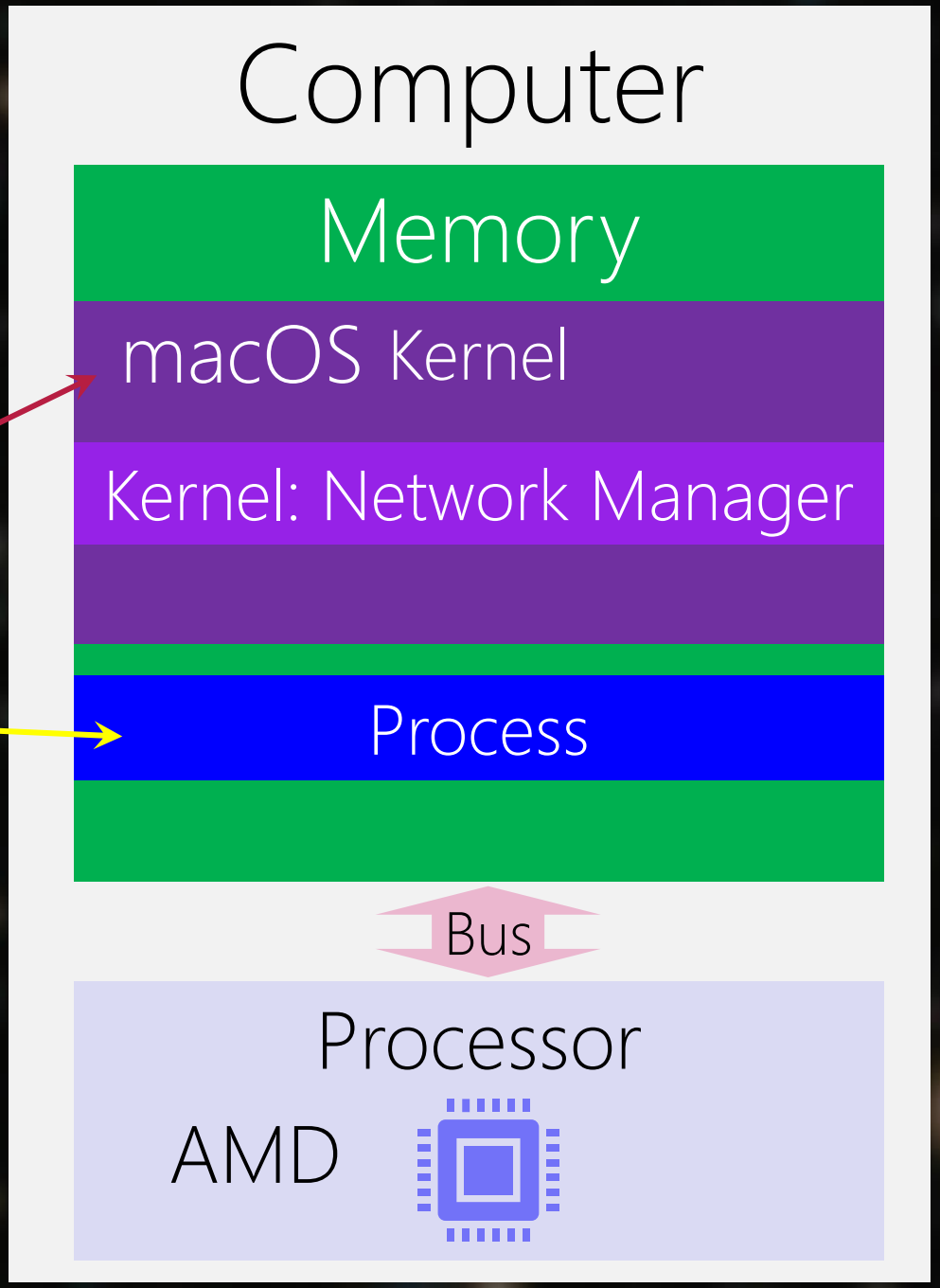


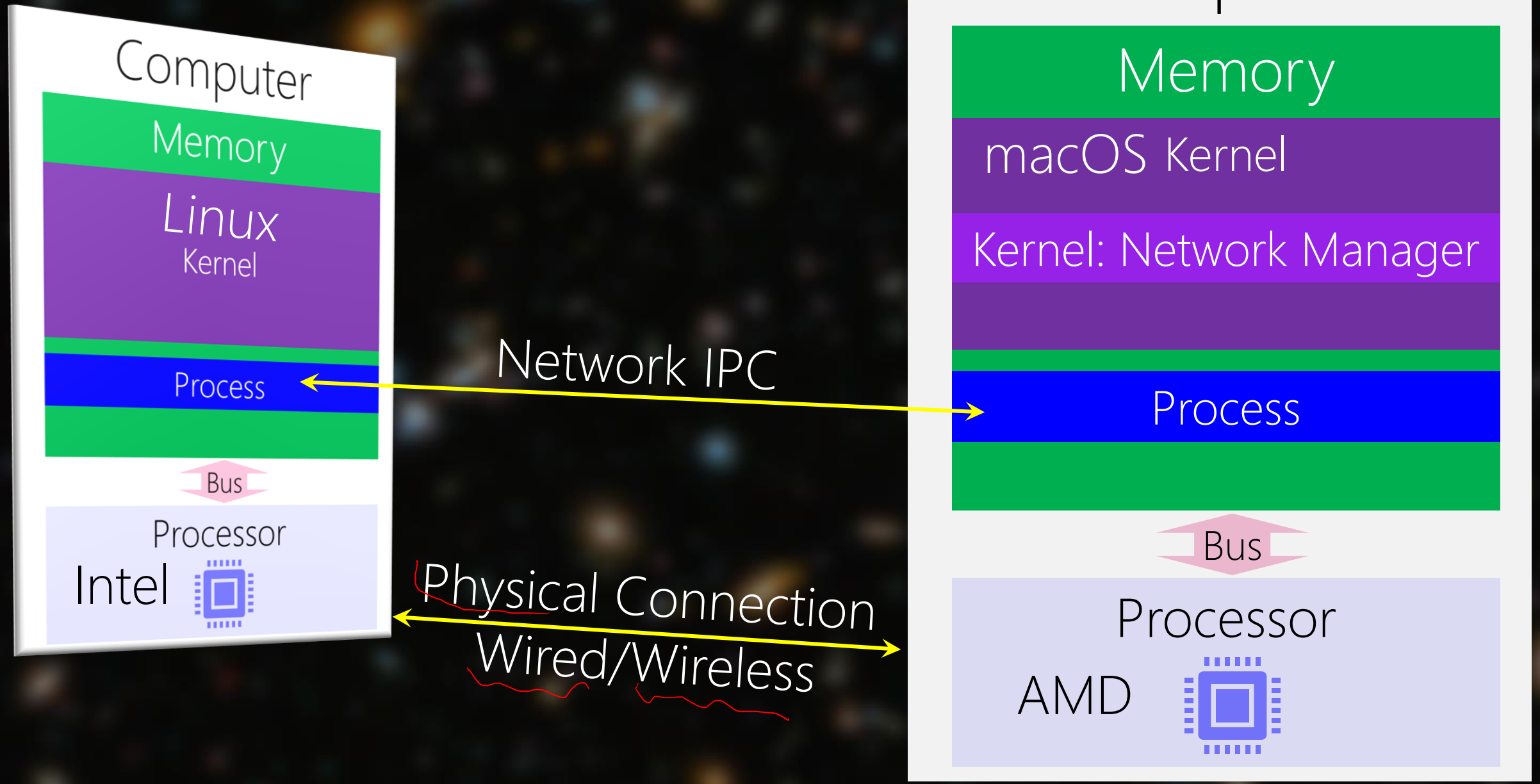
Different Machines

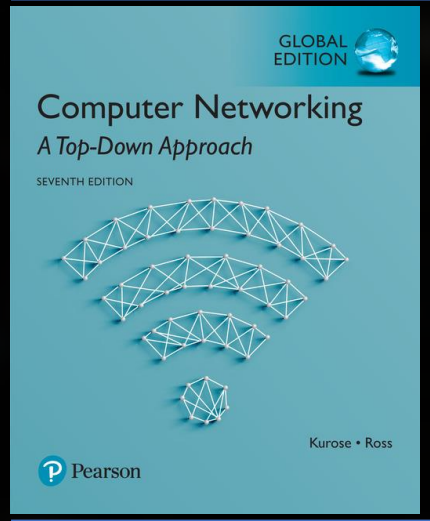




Different OSs



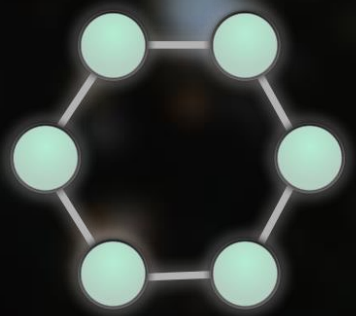

Computer
Memory
Linux
Kernel
Process
Bus
Processor
Intel
Network IPC
Physical Connection
Wired/Wireless
Computer
Memory
macOS Kernel
Kernel: Network Manager
Process
Bus
Processor
AMD



COMP3670: Computer Networks

Network Topology

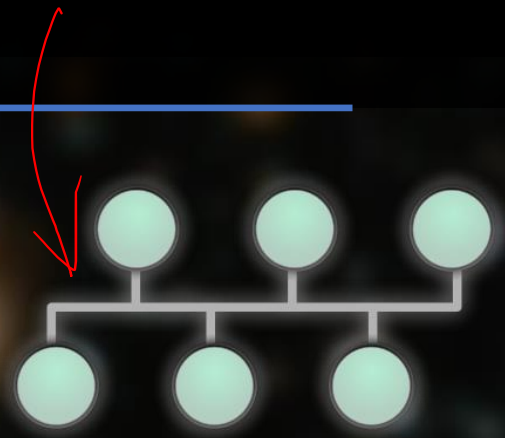
The way computer systems are connected.
By software control, we can convert one to another.



Ring

Line

Star



Bus

Network Protocol

Conversation Protocol between Computers

Language, Order (Who Talks, Who Listens), Addressing (Finding Each Other), …

1975: 2-network communications between Stanford (US) and University College London (UK)

1977: 3-network between sites in the US, the UK, and Norway

Protocol X

There are other network protocols!

X is just a name. It does not represent all this protocol offers!

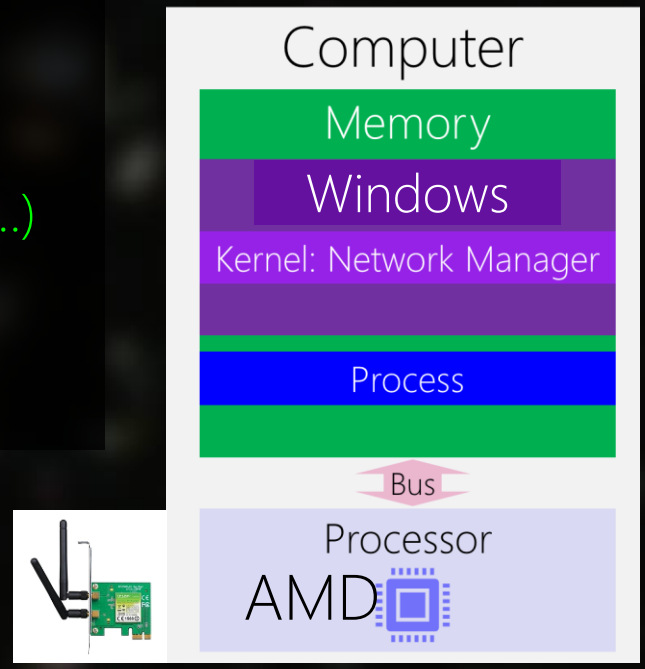
Link Layer

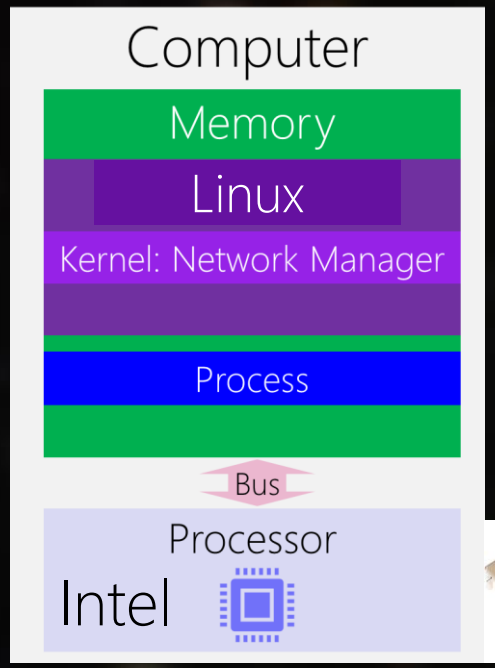
Type of physical connection

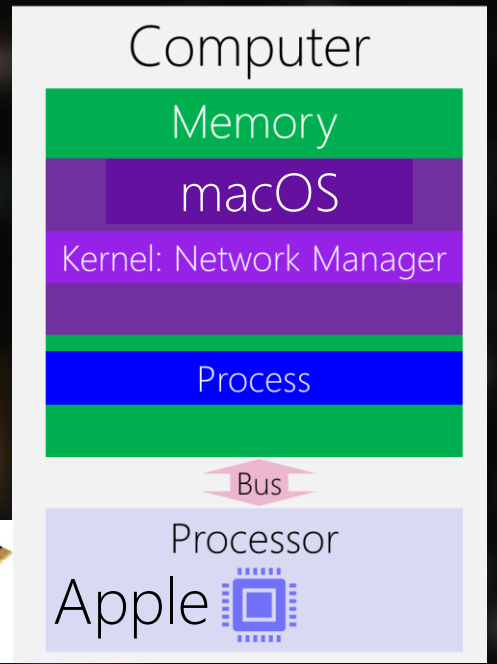
Wired (Ethernet), Wireless (Wifi, Bluetooth, Infrared, ...)

We don't care in this course!

Layered

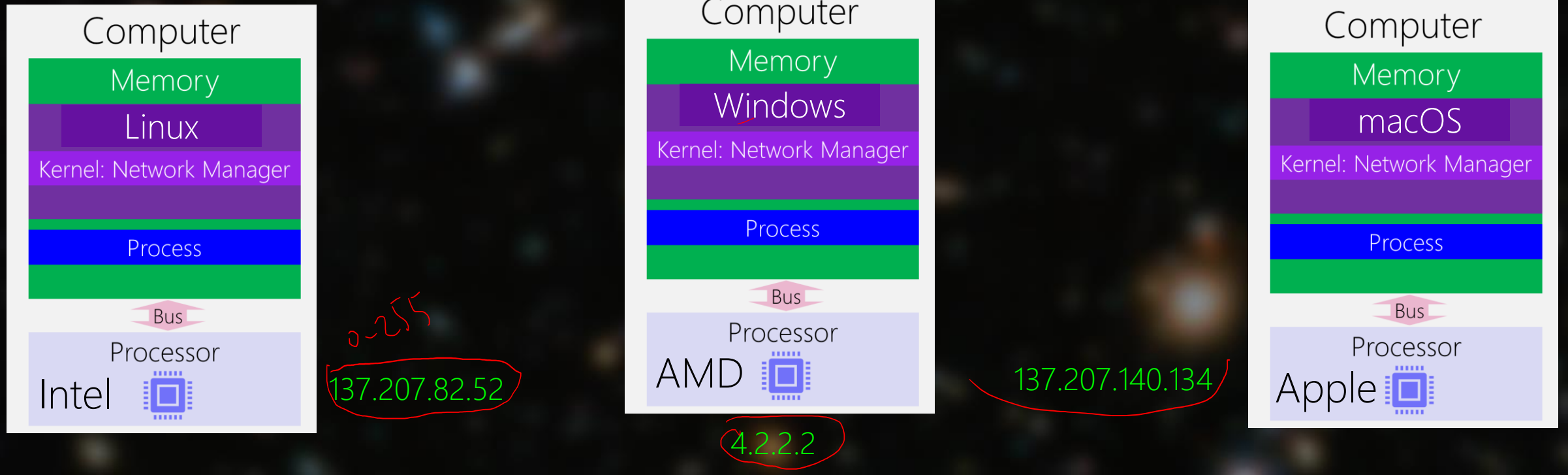






Inter-Network → Internet (Network) Layer → Internet Protocol (IP)

Computers' Address, Names, ...
We use the addresses in this course.
We don't care about the rest.
Why the format is like this?
Who assigns the addresses?
...



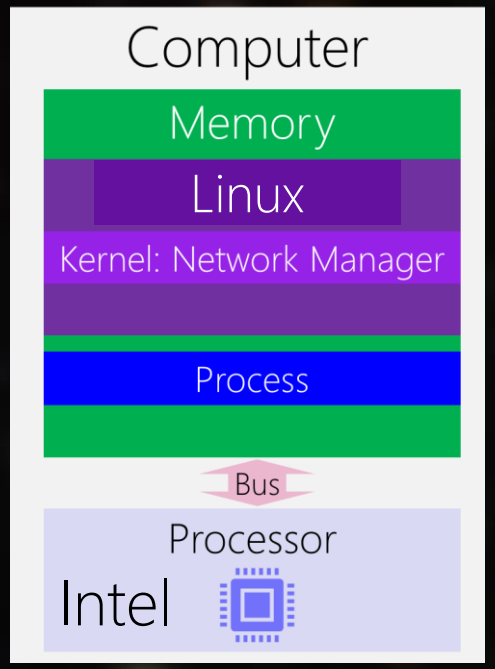
Transport Layer

Agreement on communication protocol

1) Connectionless == Sending a mail

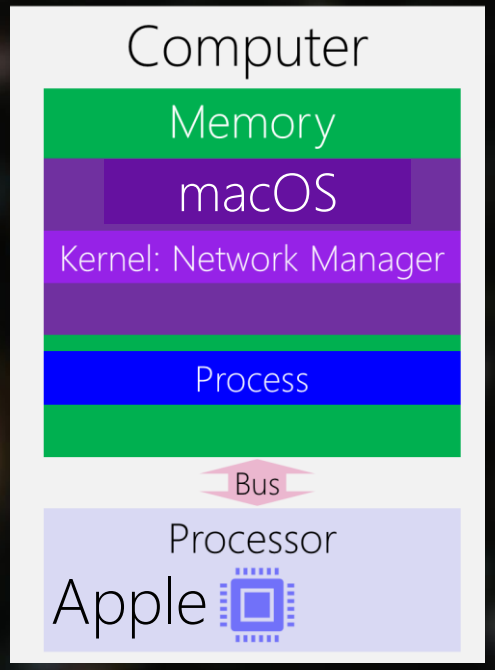
User Datagram Protocol (UDP)

- No order (a mail may be sent sooner, but received later)
- No reliability (non-tracking mail.) Cannot see whether it is received or lost
- Each message is self-contained (Does not depends on previous or next mails)
- Simple and light (no overhead for sender, like PR card by government of Canada)



137.207.82.52

137.207.140.134



Transport Layer

Agreement on communication protocol

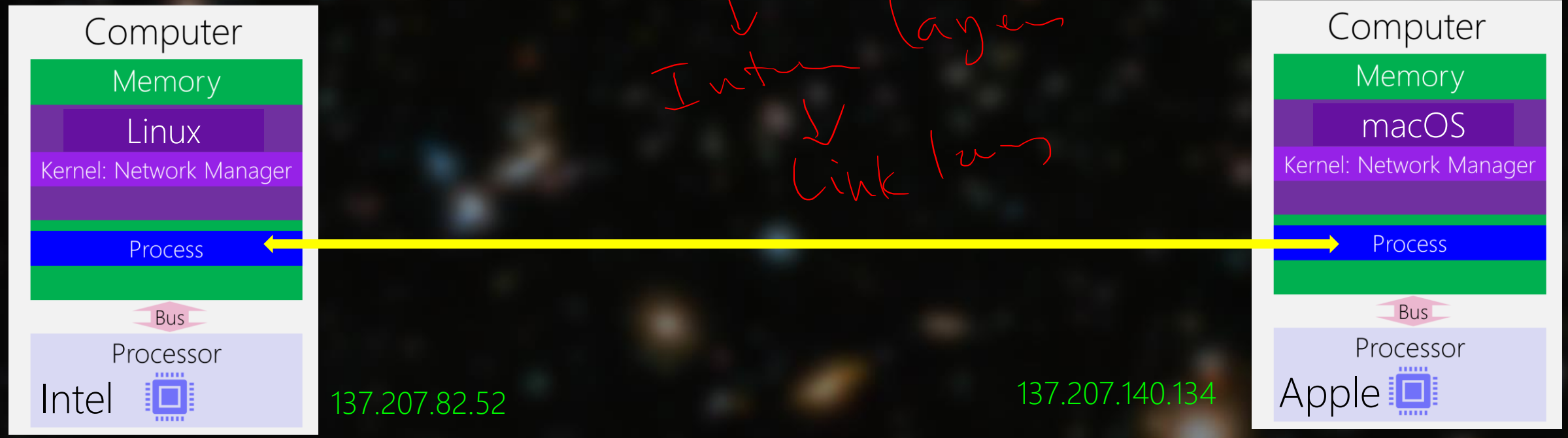
2) Connection-Oriented == Phone Call

Transmission Control Protocol (TCP)

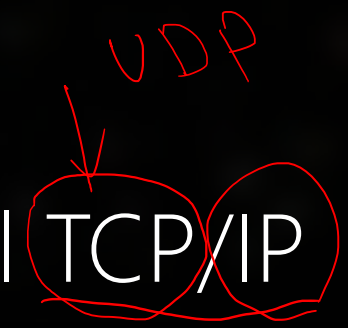
- Foremost setup a connection to make sure there is a receiver ready
- Ordered (when you talk on the phone, the words are transferred in order)
- Reliability (there is an active listener)
- Each *packet* depends on previous or next packets
- Connection overhead for sender

Application Layer

Any process that wants to communicate via the network



Protocol TCP/IP



There are other network protocols!

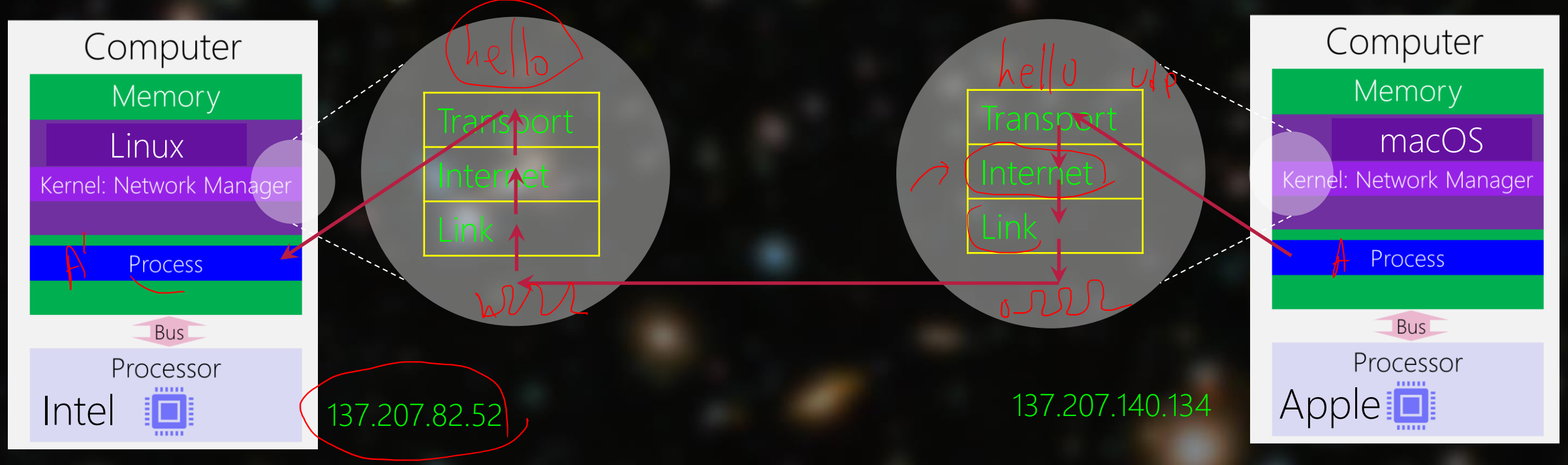
TCP/IP is just a name. It does not represent all this protocol offers!

TCP/IP

Just a name for [Link | Internet | Transport | Application] network protocol

TCP/IP

Just a name for [Link | Internet | Transport | Application] network protocol



TCP/IP

Just a name for [Link | Internet | Transport | Application] network protocol

Like a file descriptor but to read/write to another computer

Socket Descriptor

Socket Programming

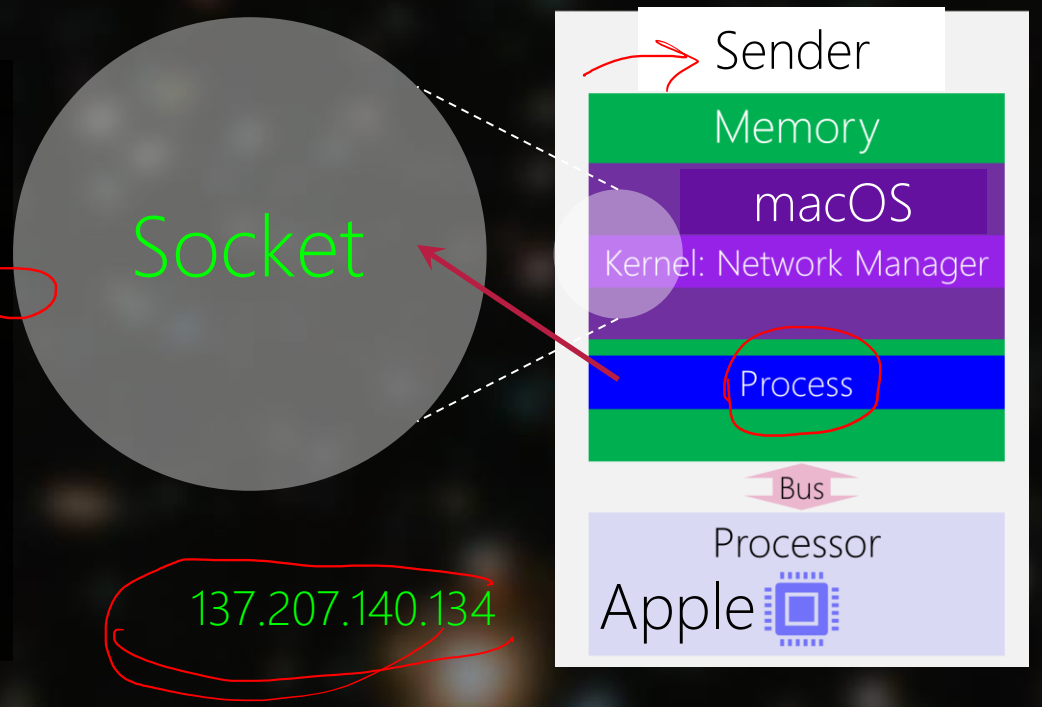
TCP/IP: UDP

TCP/IP: UDP at Sender

Connectionless Communication

Sending a mail

1) Creating Socket
2) Binding a Sender Address (Optional)
3) Find the Receiver's Address
4) Send the Mail to the Receiver



TCP/IP: UDP at Sender

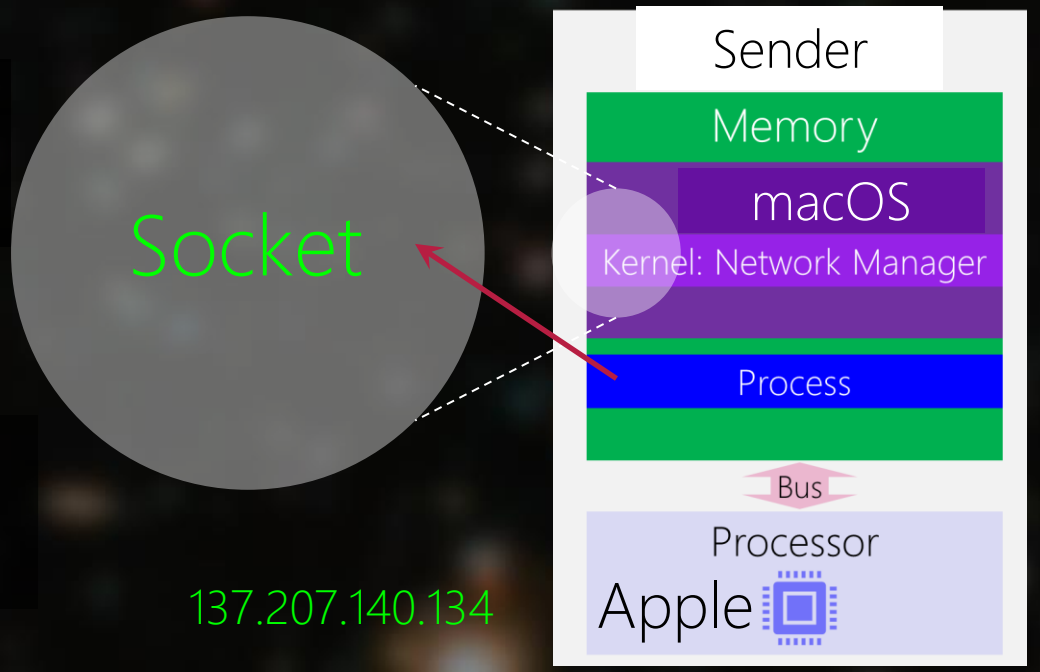
Connectionless Communication

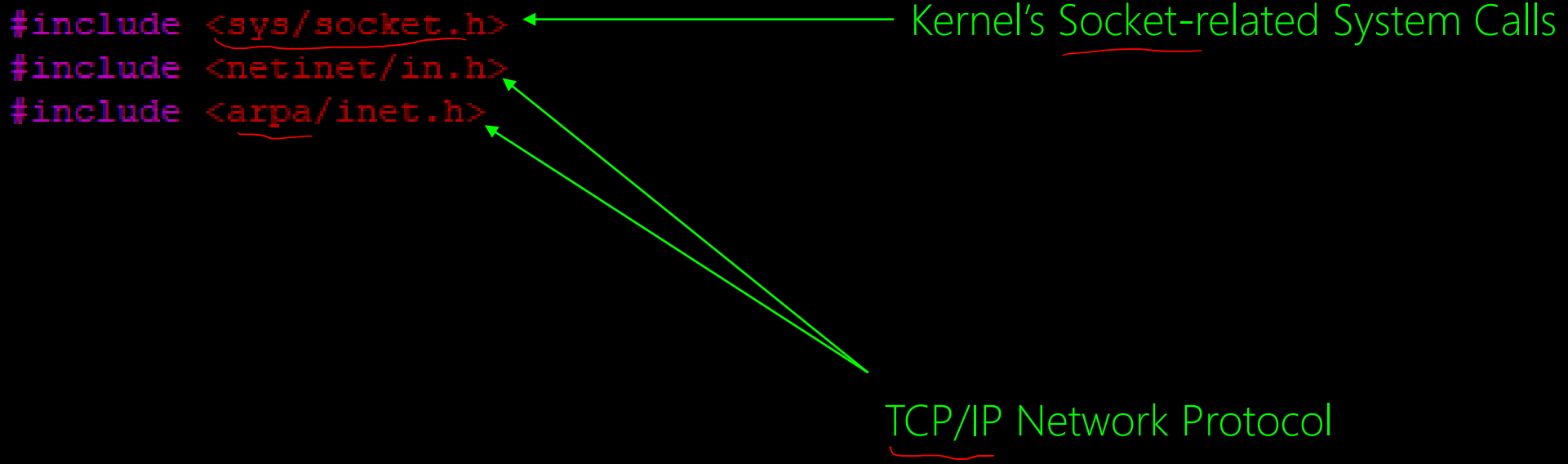
Sending a mail

1) Creating Socket

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Returns socket descriptor (sd) if OK, –1 on error




#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
Kernel's Socket-related System Calls
TCP/IP Network Protocol

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <string.h>
int main(void){
        int domain = AF_INET;//Network Protocol: TCP/IP
        int type = SOCK_DGRAM;//Connectionless
        int protocol = 0;//Default transport: UDP for Internet connectionless
```

Set up the type of network communication

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain (optional in POSIX.1)
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

Figure 16.1 Socket communication domains

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams

Figure 16.2 Socket types

Protocol	Description
IPPROTO_IP	IPv4 Internet Protocol
IPPROTO_IPV6	IPv6 Internet Protocol (optional in POSIX.1)
IPPROTO_ICMP	Internet Control Message Protocol
IPPROTO_RAW	Raw IP packets protocol (optional in POSIX.1)
IPPROTO_TCP	Transmission Control Protocol
IPPROTO_UDP	User Datagram Protocol

Figure 16.3 Protocols defined for Internet domain sockets

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <string.h>
int main(void){
        int domain = AF_INET;//Network Protocol: TCP/IP
        int type = SOCK_DGRAM;//Connectionless
        int protocol = 0;//Default transport: UDP for Internet connectionless

        int sender_sd;//socket descriptor ~= file descriptor
        sender_sd = socket(domain, type, protocol);
        if (sender_sd == -1){
                printf("error in creating socket!\n");
                exit(1);
        }
        else
                printf("socket has created for sender with sd:%d\n", sender_sd);
```

Open a socket and receive a socket descriptor

Very similar to `open()` a file and file descriptor

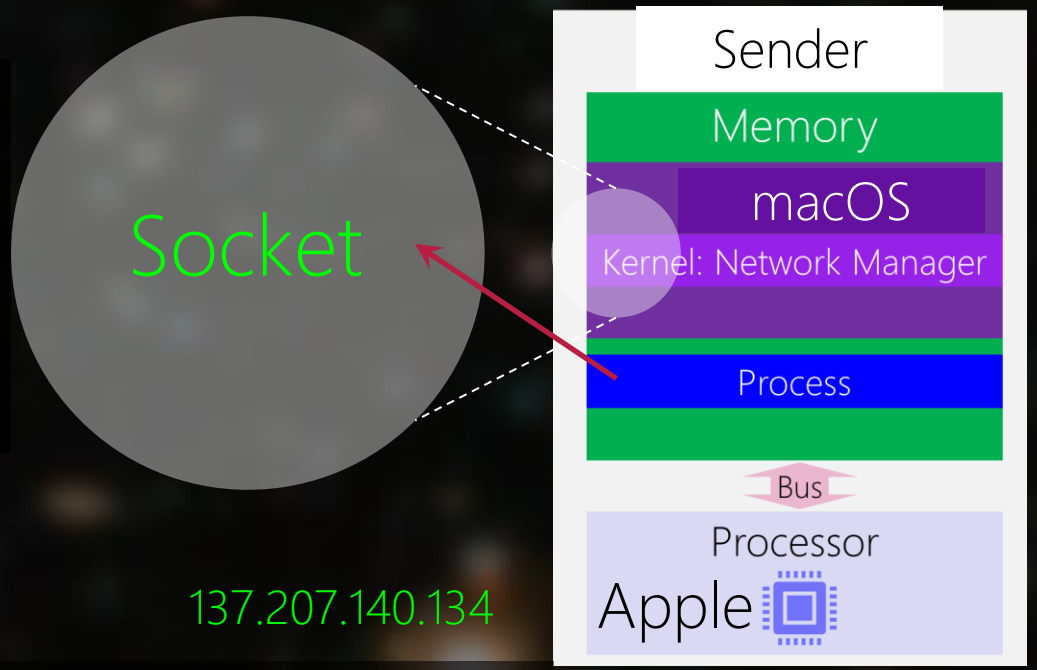
Indeed, behind the scene, there are implemented very similar!

TCP/IP: UDP at Sender

Connectionless Communication

Sending a mail

1) Creating Socket
2) Binding a Sender Address (Optional)




```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t len);
```

Returns 0 if OK, –1 on error

```
//Binding to an address is optional for sender!
struct in_addr sender_sin_address;
sender_sin_address.s_addr = inet_addr("137.207.140.134 ");//nslookup `hostname`
```

v4

IP Address of the computer system
[[0-255][0-255][0-255][0-255]]
4 bytes

There are many ways to know the IP of a computer, e.g.,

```
hfani@bravo:~$ nslookup `hostname`
```

```
//Binding to an address is optional for sender!
struct in_addr sender_sin_address;
sender_sin_address.s_addr = inet_addr("137.207.140.134 ");//nslookup `hostname`
int sender_sin_port = htons(2000);//larger than 1024
```

`htons(0);` → random port by the kernel

In a computer system, there are multiple processes
To distinguish which process is the sender → PORT

End Point (Full Address) → IP:PORT

It is unique all around the world! Why?

```
//Binding to an address is optional for sender!
struct in_addr sender_sin_address;
sender_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname`
int sender_sin_port = htons(2000);//larger than 1024

struct sockaddr_in sender_sin;
sender_sin.sin_family = domain;
sender_sin.sin_addr = sender_sin_address;
sender_sin.sin_port = sender_sin_port;
```

```
//Binding to an address is optional for sender!
struct in_addr sender_sin_address;
sender_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname`
int sender_sin_port = htons(2000);//larger than 1024

struct sockaddr_in sender_sin;
sender_sin.sin_family = domain;
sender_sin.sin_addr = sender_sin_address;
sender_sin.sin_port = sender_sin_port;
int result = bind(sender_sd, (struct sockaddr *) &sender_sin, sizeof(sender_sin));
if (result == -1){
        printf("error in binding sender to the address:port = %d:%d\n", sender_sin.sin_
        exit(1);
}
else
        printf("sender bound to the address:port = %d:%d\n", sender_sin.sin_addr, sende
```

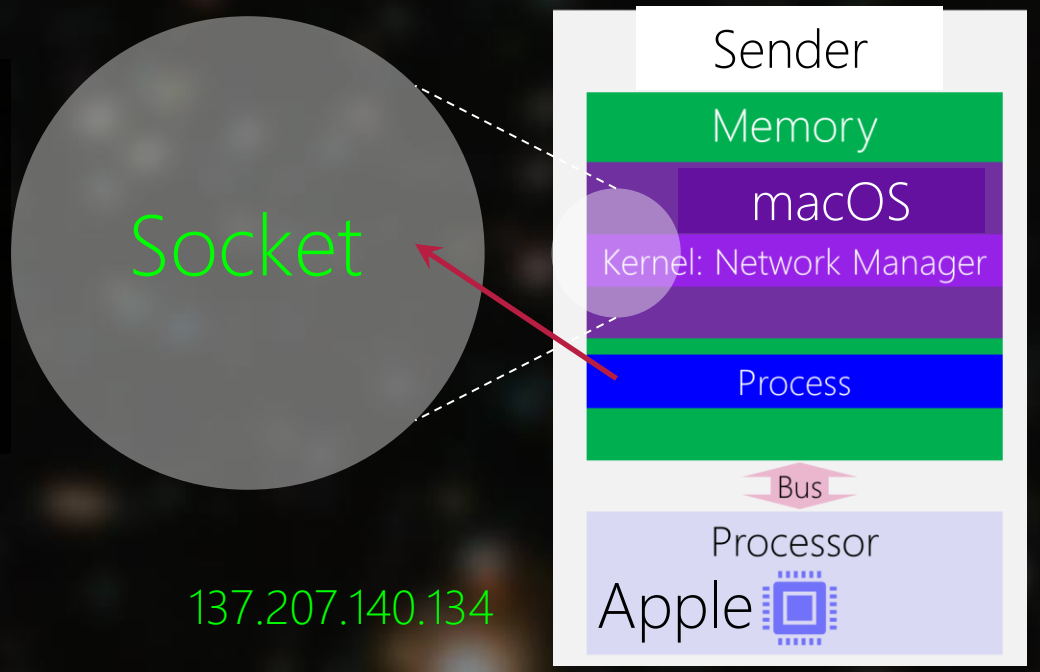
Socket ↔ IP:PORT

TCP/IP: UDP at Sender

Connectionless Communication

Sending a mail

1) Creating Socket
2) Binding to an Address (Optional)
3) Find the Receiver's Address



```
//Sending a message to the receiver at 137.207.82.52:2001
struct in_addr receiver_sin_address;
receiver_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname` at the ta
int receiver_sin_port = htons(2001);//larger than 1024

struct sockaddr_in receiver_sin;
receiver_sin.sin_family = domain;//same network protocol
receiver_sin.sin_addr = receiver_sin_address;
receiver_sin.sin_port = receiver_sin_port;
```

IP:PORT of the Receiver

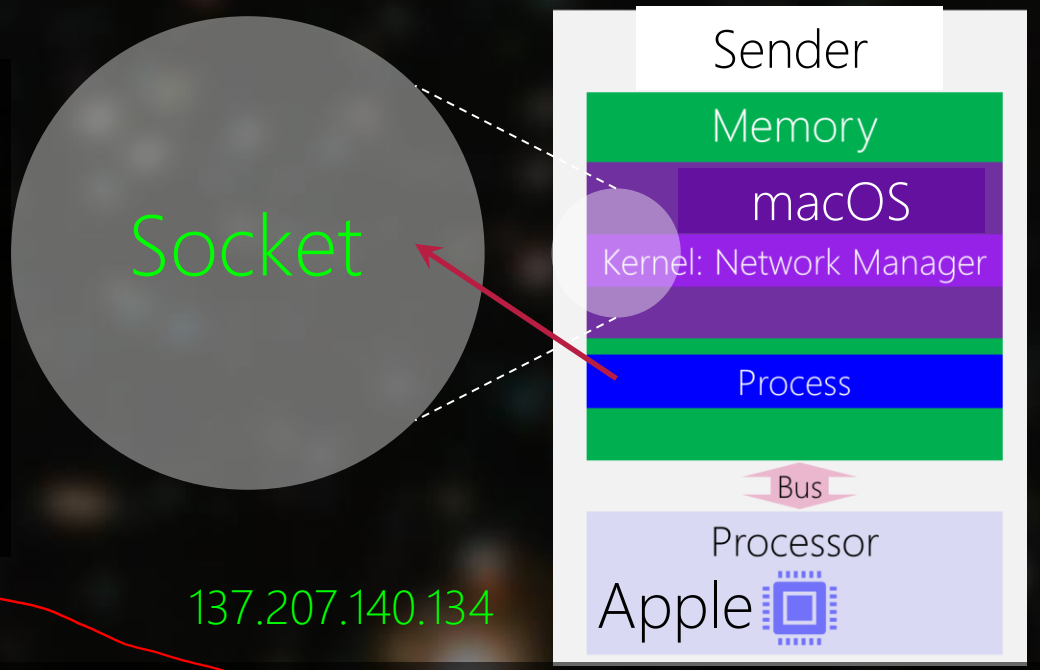
It is unique all around the world! Why?

TCP/IP: UDP at Sender

Connectionless Communication

Sending a mail

1) Creating Socket
2) Binding to an Address (Optional)
3) Find the Receiver's Address
4) Send the Mail to the Receiver




```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *buf, size_t nbytes, int flags, const struct sockaddr *destaddr, socklen_t destlen);
```

Returns number of bytes sent if OK, –1 on error

```
char *mail = "a 10 percent promotion for candian tire!";

result = sendto(sender_sd, mail, strlen(mail), 0, (struct sockaddr *)&receiver_sin, size
if (result == -1){
        printf("error in sending message to the receiver!\n");
        exit(1);
}
else
{
        printf("a mail has sent to the receiver at address:port = %d:%d\n", receiver_sin
        printf("the content of the mail is <%s>\n", mail);
}
```

Message

Very similar to `write()` to a file

Receiver's IP:PORT

```c
#include <stdlib.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <string.h>
int main(void){
        int domain = AF_INET;//Network Protocol: TCP/IP
        int type = SOCK_DGRAM;//Connectionless
        int protocol = 0;//Default transport: UDP for Internet connectionless

        int sender_sd;//socket descriptor ~= file descriptor
        sender_sd = socket(domain, type, protocol);
        if (sender_sd == -1){
                printf("error in creating socket!\n");
                exit(1);
        }
        else
                printf("socket has created for sender with sd:%d\n", sender_sd);

        //Binding to an address is optional for sender!
        struct in_addr sender_sin_address;
        sender_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname`
        int sender_sin_port = htons(2000);//larger than 1024

        struct sockaddr_in sender_sin;
        sender_sin.sin_family = domain;
        sender_sin.sin_addr = sender_sin_address;
        sender_sin.sin_port = sender_sin_port;
        int result = bind(sender_sd, (struct sockaddr *) &sender_sin, sizeof(sender_sin));
        if (result == -1){
                printf("error in binding sender to the address:port = %d:%d\n", sender_sin.sin_addr, sender_sin.sin_port);
                exit(1);
        }
        else
                printf("sender bound to the address:port = %d:%d\n", sender_sin.sin_addr, sender_sin.sin_port);

        //Sending a message to the receiver at 137.207.82.52:2001
        struct in_addr receiver_sin_address;
        receiver_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname` at the target machine (here is the same as sender)
        int receiver_sin_port = htons(2001);//larger than 1024

        struct sockaddr_in receiver_sin;
        receiver_sin.sin_family = domain;//same network protocol
        receiver_sin.sin_addr = receiver_sin_address;
        receiver_sin.sin_port = receiver_sin_port;

        char *mail = "a 10 percent promotion for candian tire!";

        result = sendto(sender_sd, mail, strlen(mail), 0, (struct sockaddr *)&receiver_sin, sizeof(receiver_sin));
        if (result == -1){
                printf("error in sending message to the receiver!\n");
                exit(1);
        }
        else
        {
                printf("a mail has sent to the receiver at address:port = %d:%d\n", receiver_sin.sin_addr, receiver_sin.sin_port);
                printf("the content of the mail is <%s>\n", mail);
        }
```

1) Creating Socket

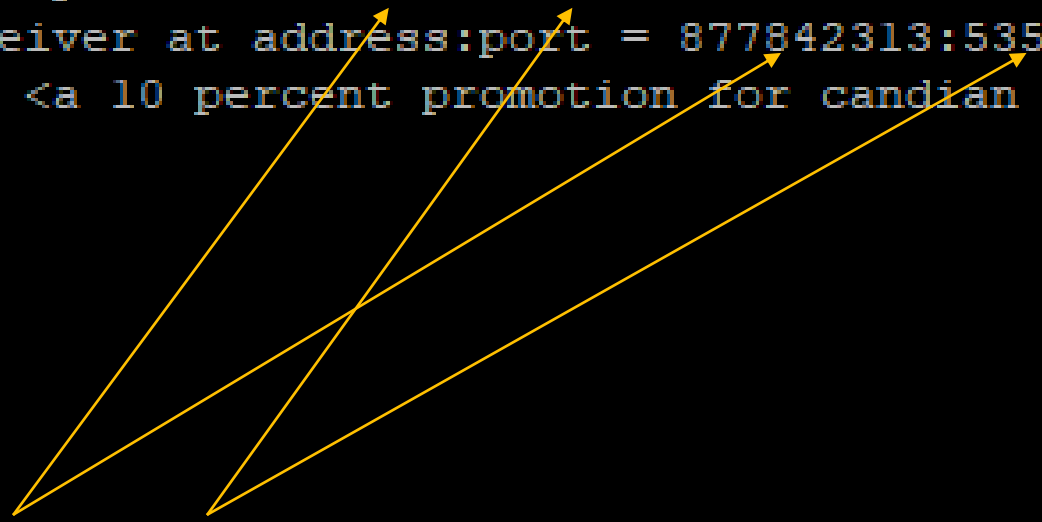
2) Binding a Sender Address (Optional)

3) Find the Receiver's Address

4) Send the Mail to the Receiver

```
hfani@bravo:~$ ./sender
socket has created for sender with sd:3
sender bound to the address:port = -2037592183 :53255
a mail has sent to the receiver at address:port = 877842313:53511
the content of the mail is <a 10 percent promotion for candian tire!>
```

```
hfani@bravo:~$ ./sender
socket has created for sender with sd:3
sender bound to the address:port = -2037592183 :53255
a mail has sent to the receiver at address:port = 877842313:53511
the content of the mail is <a 10 percent promotion for candian tire!>
```



Why IP:PORT does not look like familiar?

Sender's IP:PORT =137.207.140.134:2000

Receiver's IP:PORT=137.207.82.52:2001

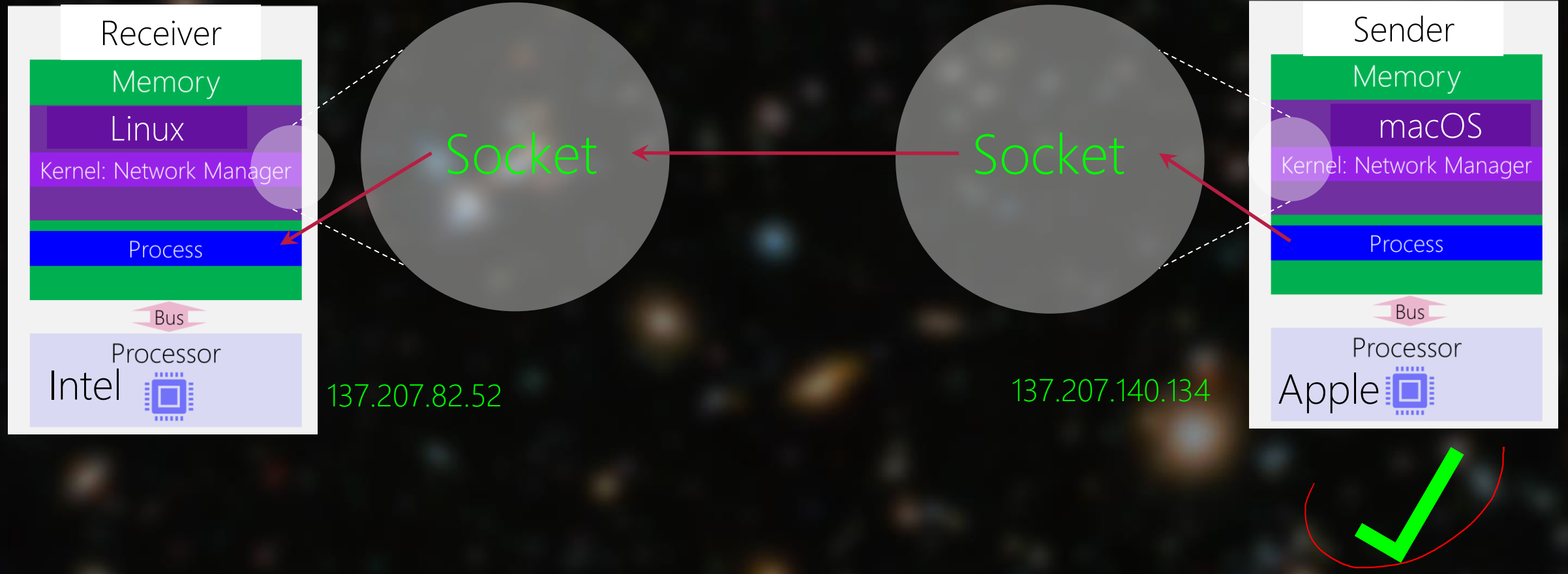
```
hfani@bravo:~$ ./sender
socket has created for sender with sd:3
sender bound to the address:port = -2037592183 :53255
a mail has sent to the receiver at address:port = 877842313:53511
the content of the mail is <a 10 percent promotion for candian tire!>
```

But there no receiver!

What happen to the mail?!

TCP/IP

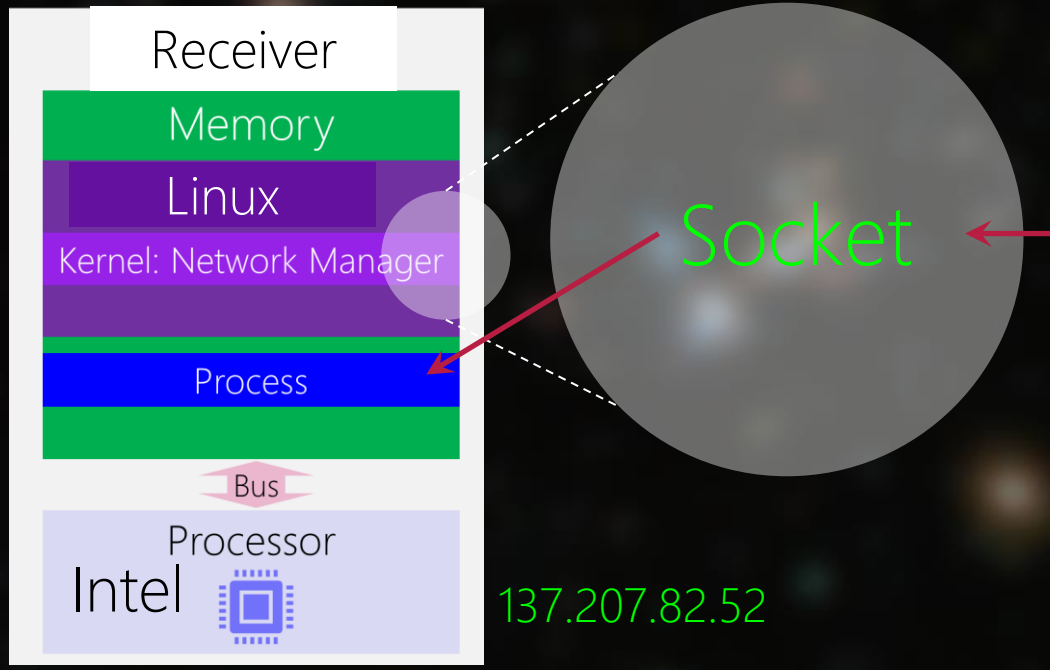
Just a name for [Link | Internet | Transport | Application] network protocol



TCP/IP: UDP at Receiver

Connectionless Communication

Sending a mail

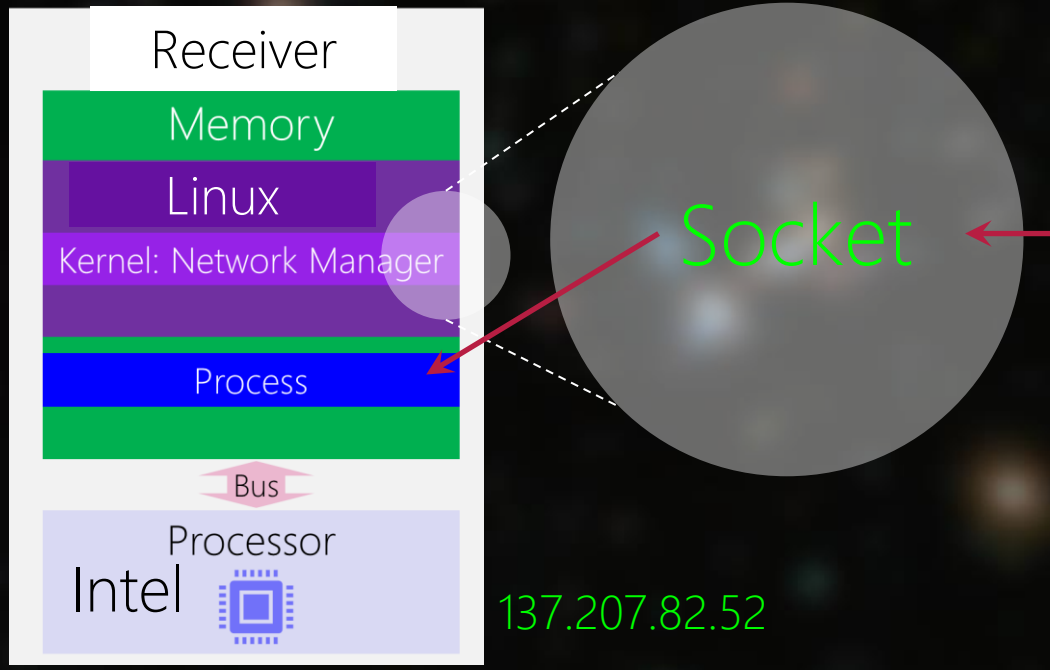


1) Creating Socket
2) Binding to an Address (MUST)
3) Wait to receive a mail
4) Find the Sender's Address (Optional)
5) Read the Mail from the Sender

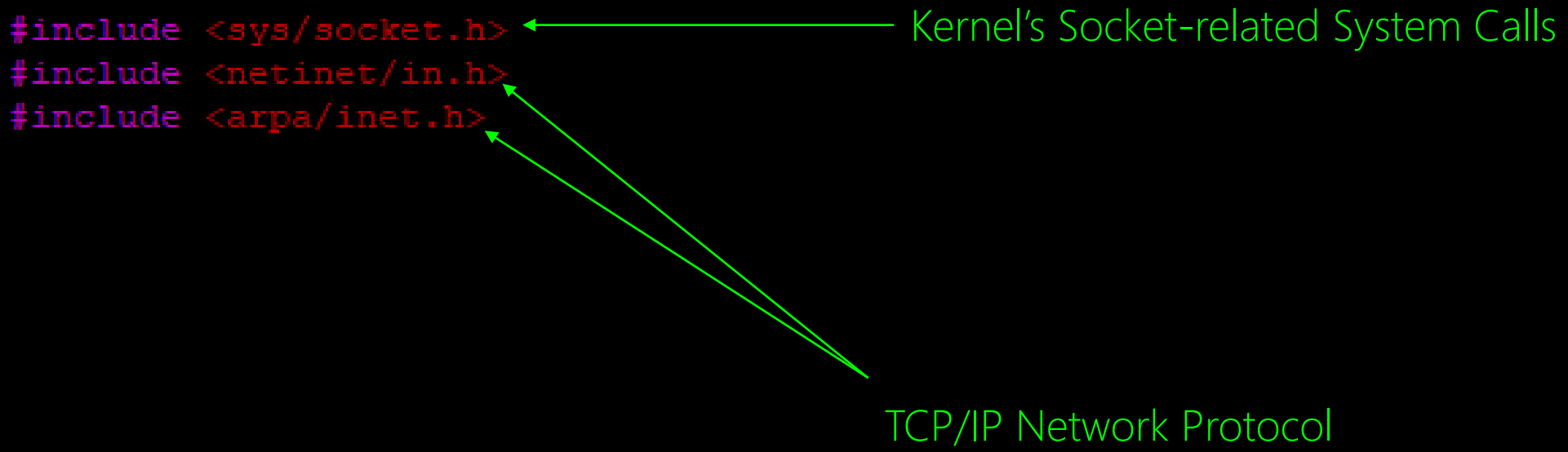
TCP/IP: UDP at Receiver

Connectionless Communication

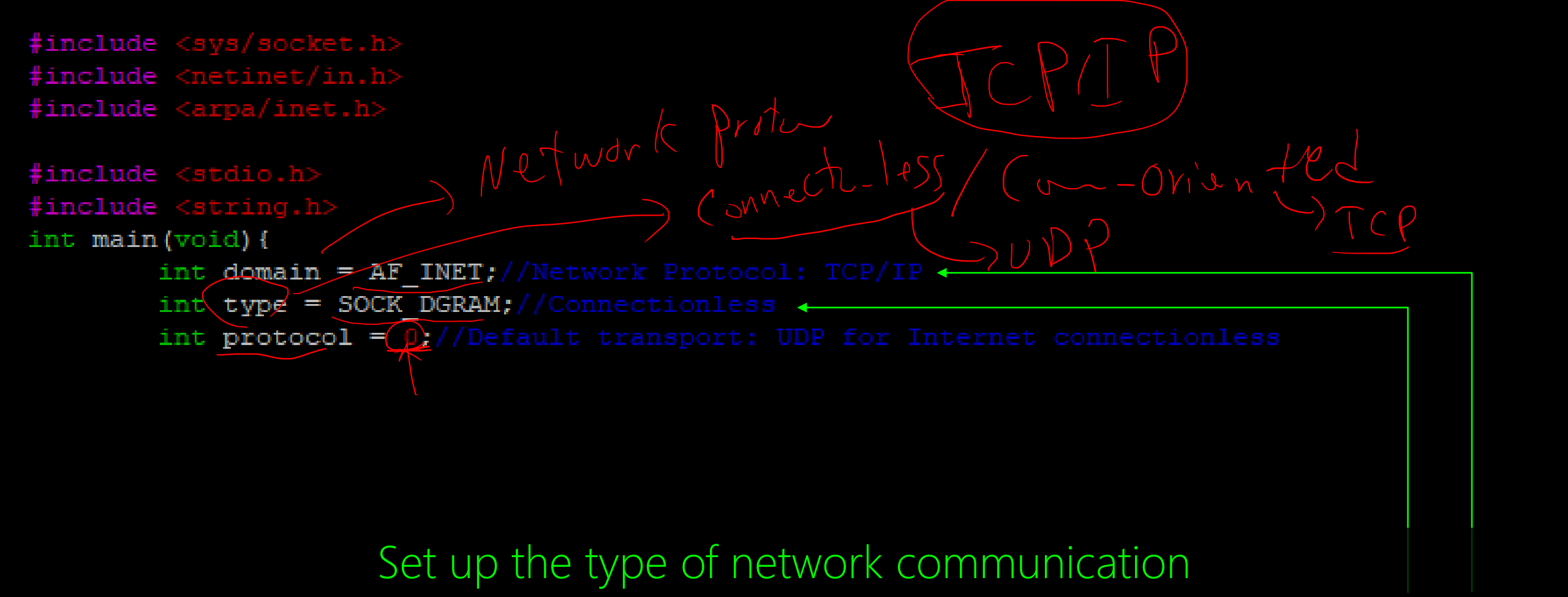
Sending a mail



1) Creating Socket


#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
Kernel's Socket-related System Calls
TCP/IP Network Protocol

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <string.h>
int main(void){
        int domain = AF_INET;//Network Protocol: TCP/IP
        int type = SOCK_DGRAM;//Connectionless
        int protocol = 0;//Default transport: UDP for Internet connectionless
```



Set up the type of network communication

```c
int receiver_sd;//socket descriptor ~= file descriptor
receiver_sd = socket(domain, type, protocol);
if (receiver_sd == -1){
        printf("error in creating socket!\n");
        exit(1);
}
else
        printf("socket has created for receiver with sd:%d\n", receiver_sd);
```

Open a socket and receive a socket descriptor

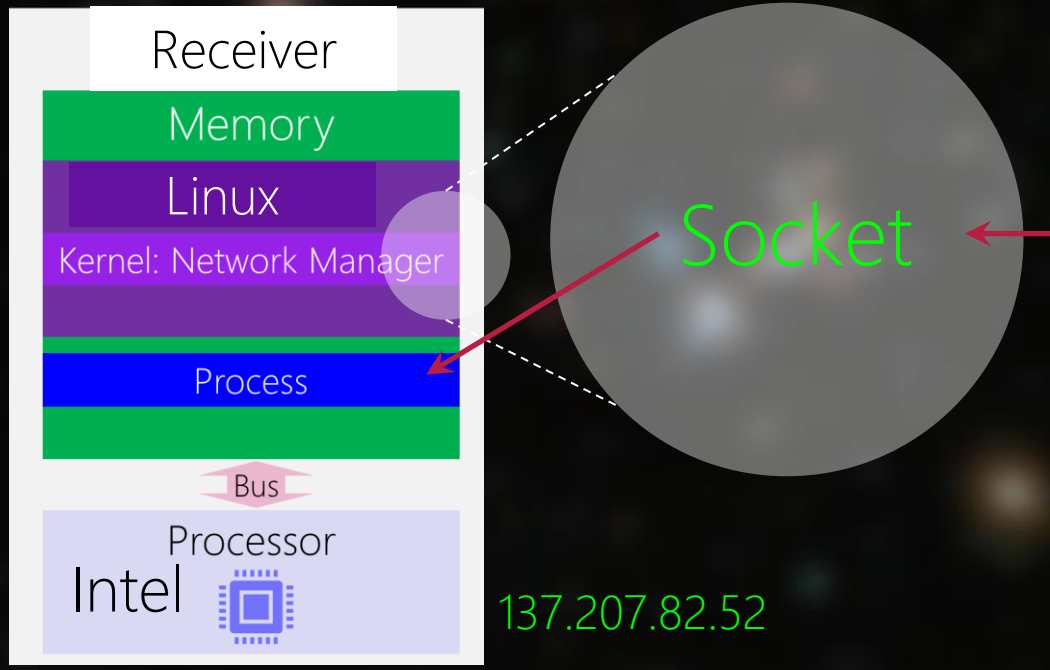
Very similar to `open()` a file and file descriptor

Indeed, behind the scene, there are implemented very similar!

TCP/IP: UDP at Receiver

Connectionless Communication

Sending a mail



1) Creating Socket
2) Binding to an Address (MUST)

```c
//Binding to an address is a must for receiver!
struct in_addr receiver_sin_address;
receiver_sin_address.s_addr = inet_addr("137.207.82.52");//nslookup `hostname`
int receiver_sin_port = htons(2001);//larger than 1024

struct sockaddr_in receiver_sin;
receiver_sin.sin_family = domain;
receiver_sin.sin_addr = receiver_sin_address;
receiver_sin.sin_port = receiver_sin_port;
int result = bind(receiver_sd, (struct sockaddr *) &receiver_sin, sizeof(receiver_sin));
if (result == -1){
        printf("error in binding receiver to the address:port = %d:%d\n", receiver_sin.sin_addr, rece
        exit(1);
}
else
        printf("receiver bound to the address:port = %d:%d\n", receiver_sin.sin_addr, receiver_sin.si
```

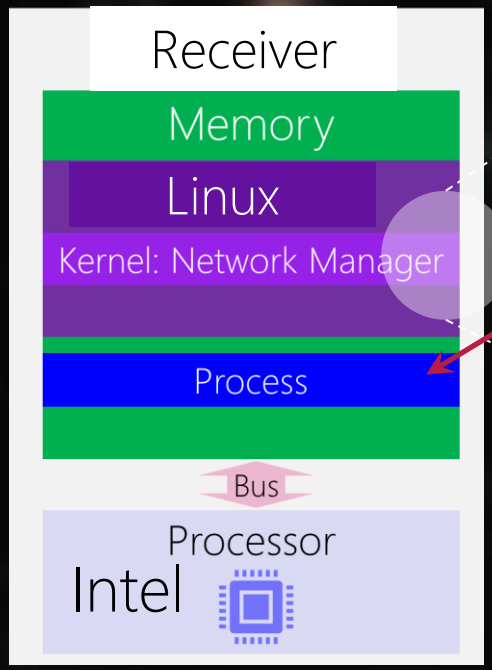
Very similar to the sender's binding of socket to IP:PORT

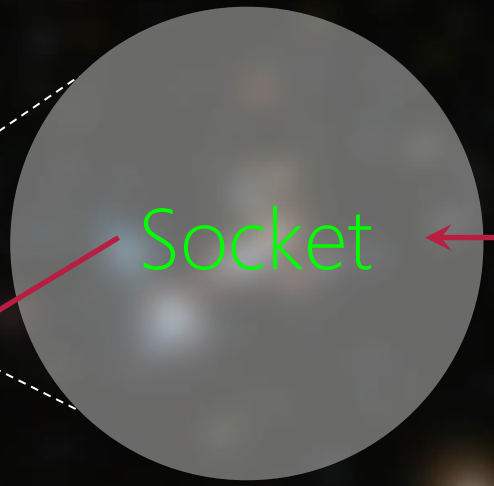
But for receiver it is a MUST. Why?

TCP/IP: UDP at Receiver

Connectionless Communication

Sending a mail





137.207.82.52

1) Creating Socket
2) Binding to an Address (MUST)
3) Wait to receive a mail

Pause

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *restrict buf, size_t len, int flags, struct sockaddr *restrict addr, socklen_t *restrict addrlen);
```

Returns length of message in bytes, −1 on error

```c
//wait for a message ...
struct sockaddr_in sender_sin;//I want to know who send the message
char mailbox[100];
int sender_sin_len;
setbuf(stdout, NULL);
while(1)
{
        result = recvfrom(receiver_sd, mailbox, sizeof(mailbox), 0, (struct sockaddr *) &sender_sin,
        if (result == -1){
                printf("error in opening mail from sender!\n");
                exit(1);
        }
        else
                printf("the content of mail is: %s", mailbox);
}

}
```

Pause

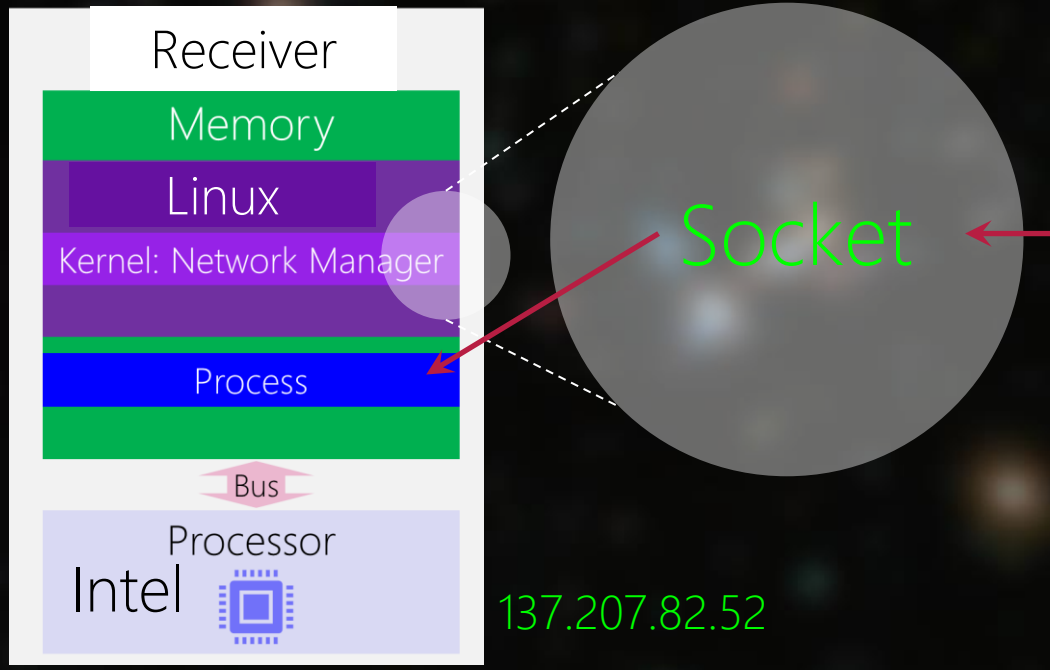
Wait to receive a mail at socket

Very similar to `read()` from a file

It is a blocking call! It pauses until a new mail

TCP/IP: UDP at Receiver

Connectionless Communication
Sending a mail



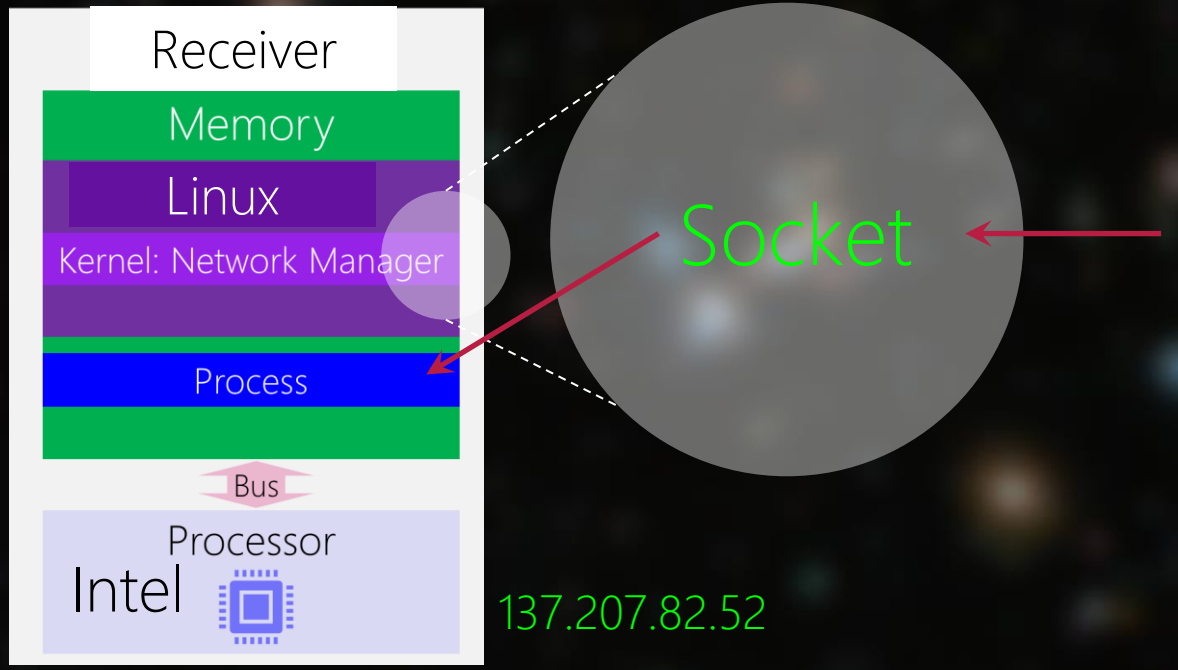
1) Creating Socket
2) Binding to an Address (MUST)
3) Wait to receive a mail
4) Find the Sender's Address (Optional)

```
        //wait for a message ...
        struct sockaddr_in sender_sin;//I want to know who send the message
        char mailbox[100];
        int sender_sin_len;
        setbuf(stdout, NULL);
        while(1)
        {
                result = recvfrom(receiver_sd, mailbox, sizeof(mailbox), 0, (struct sockaddr *) &sender_sin,
                if (result == -1){
                        printf("error in opening mail from sender!\n");
                        exit(1);
                }
                else
                        printf("the content of mail is: %s", mailbox);
        }

}
```

You can ignore but you can know who is the sender and decide Sender's IP:PORT

TCP/IP: UDP at Receiver

Connectionless Communication
Sending a mail



1) Creating Socket
2) Binding to an Address (MUST)
3) Wait to receive a mail
4) Find the Sender's Address (Optional)
5) Read the Mail from the Sender

```
        //wait for a message ...
        struct sockaddr_in sender_sin;//I want to know who send the message
        char mailbox[100];
        int sender_sin_len;
        setbuf(stdout, NULL);
        while(1)
        {
                result = recvfrom(receiver_sd, mailbox, sizeof(mailbox), 0, (struct sockaddr *) &sender_sin,
                if (result == -1){
                        printf("error in opening mail from sender!\n");
                        exit(1);
                }
                else
                        printf("the content of mail is: %s", mailbox);
        }

}
```

Like the `read()` buffer for file

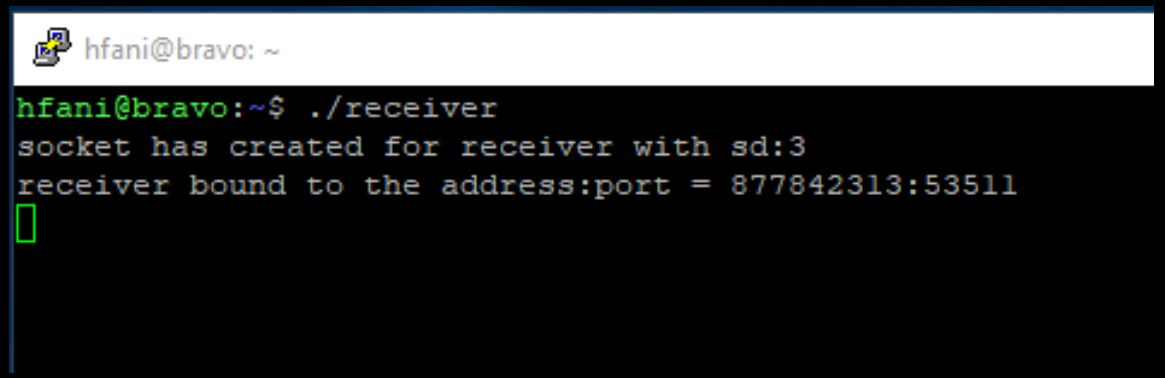
```
//wait for a message ...
struct sockaddr_in sender_sin;//I want to know who send the message
char mailbox[100];
int sender_sin_len;
setbuf(stdout, NULL);
while(1)
{
        result = recvfrom(receiver_sd, mailbox, sizeof(mailbox), 0, (struct sockaddr *) &sender_sin,
        if (result == -1){
                printf("error in opening mail from sender!\n");
                exit(1);
        }
        else
                printf("the content of mail is: %s", mailbox);
}

}
```

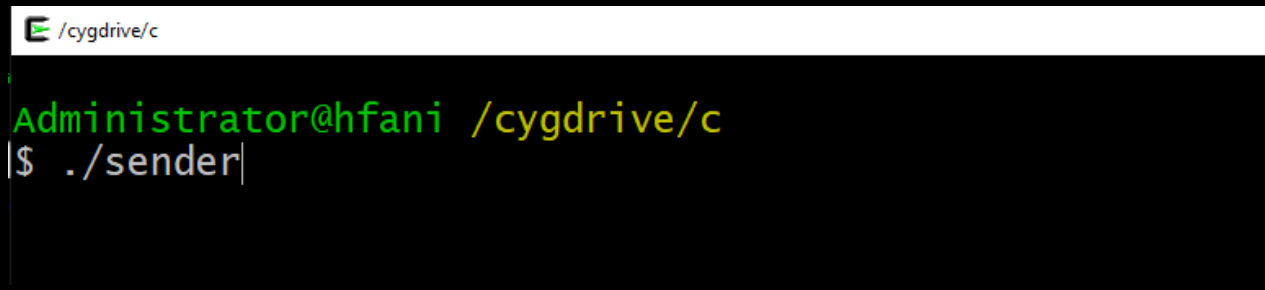
Usually, receiver never dies. It is waiting to receive a new mail forever.

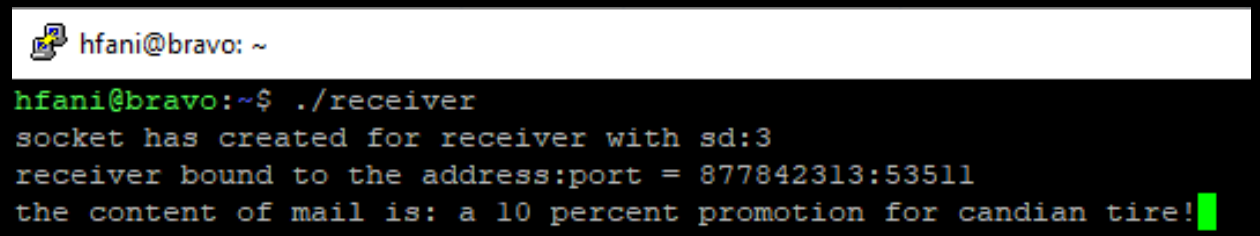
A better way to avoid receiver process waste time on waiting would be?

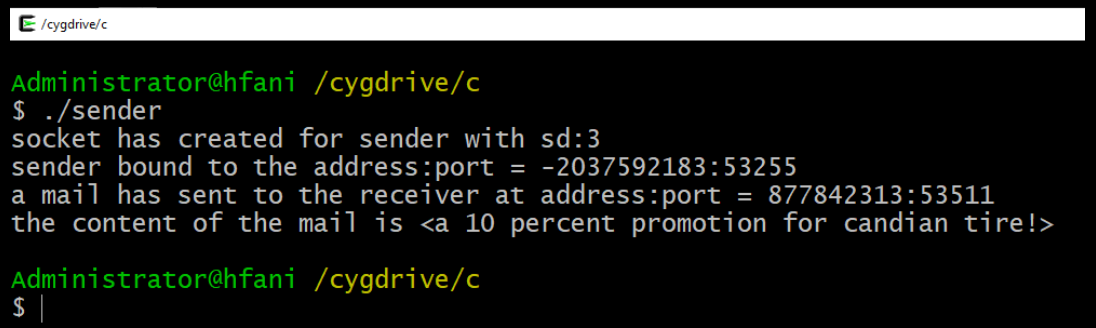
```
//wait for a message ...
struct sockaddr_in sender_sin;//I want to know who send the message
char mailbox[100];
int sender_sin_len;
setbuf(stdout, NULL);
while(1)
{
        result = recvfrom(receiver_sd, mailbox, sizeof(mailbox), 0, (struct sockaddr *) &sender_sin,
        if (result == -1){
                printf("error in opening mail from sender!\n");
                exit(1);
        }
        else
                printf("the content of mail is: %s", mailbox);
}

}
```

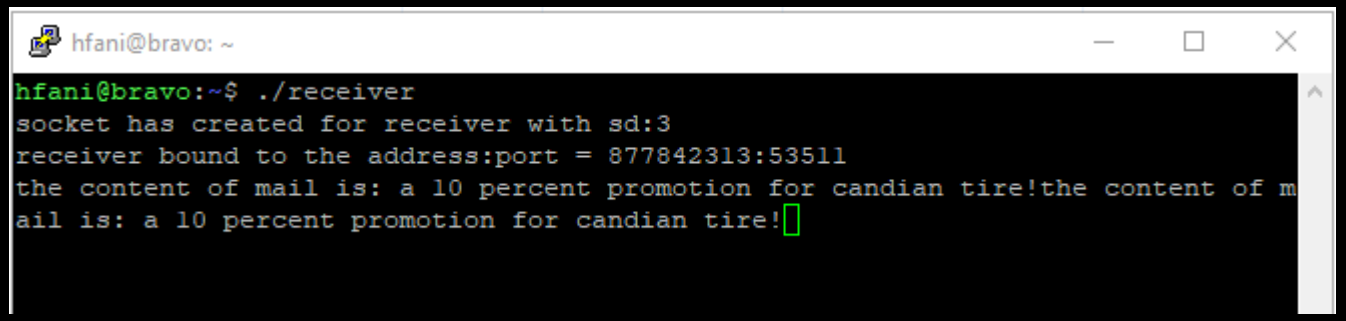
`fork()` and give the task of reading mails to the child!
The receiver then does other important tasks.

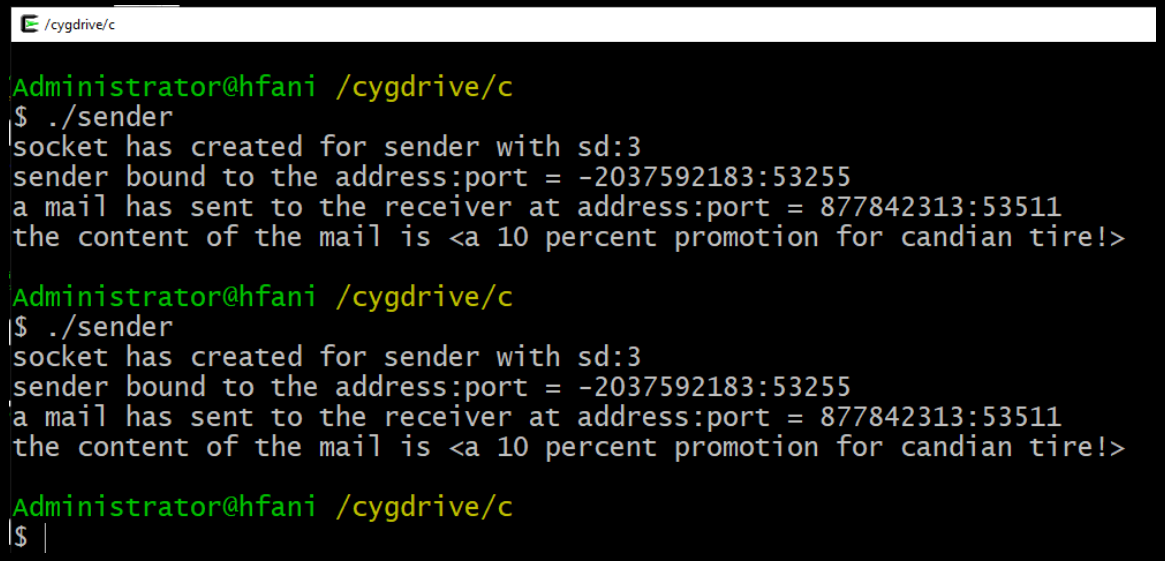

hfani@bravo: ~
hfani@bravo:~$ ./receiver
socket has created for receiver with sd:3
receiver bound to the address:port = 877842313:53511


/cygdrive/c
Administrator@hfani /cygdrive/c
$ ./sender


hfani@bravo: ~
hfani@bravo:~$ ./receiver
socket has created for receiver with sd:3
receiver bound to the address:port = 877842313:53511
the content of mail is: a 10 percent promotion for candian tire!


/cygdrive/c
Administrator@hfani /cygdrive/c
$ ./sender
socket has created for sender with sd:3
sender bound to the address:port = -2037592183:53255
a mail has sent to the receiver at address:port = 877842313:53511
the content of the mail is <a 10 percent promotion for candian tire!>
Administrator@hfani /cygdrive/c
$


hfani@bravo: ~
hfani@bravo:~$ ./receiver
socket has created for receiver with sd:3
receiver bound to the address:port = 877842313:53511
the content of mail is: a 10 percent promotion for candian tire!the content of m
ail is: a 10 percent promotion for candian tire!

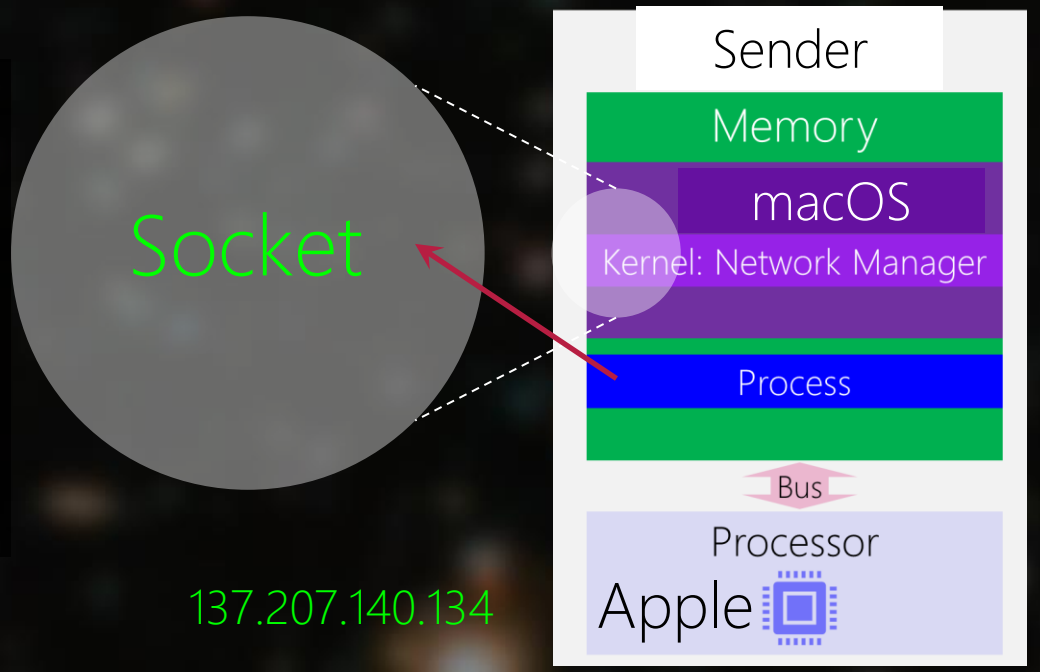

/cygdrive/c
Administrator@hfani /cygdrive/c
$ ./sender
socket has created for sender with sd:3
sender bound to the address:port = -2037592183:53255
a mail has sent to the receiver at address:port = 877842313:53511
the content of the mail is <a 10 percent promotion for candian tire!>
Administrator@hfani /cygdrive/c
$ ./sender
socket has created for sender with sd:3
sender bound to the address:port = -2037592183:53255
a mail has sent to the receiver at address:port = 877842313:53511
the content of the mail is <a 10 percent promotion for candian tire!>
Administrator@hfani /cygdrive/c
$

TCP/IP: UDP at Sender

Connectionless Communication

Sending a mail

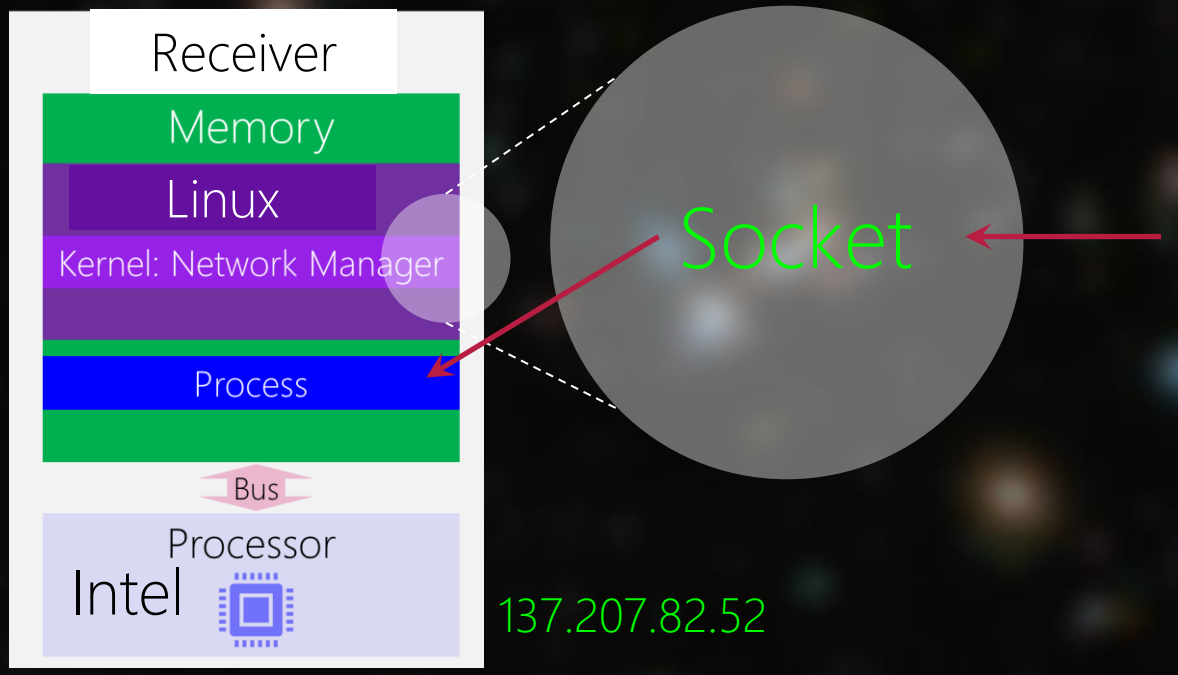
1) `socket()`
2) `bind()`
3) Receiver's Address
4) `sendto()`



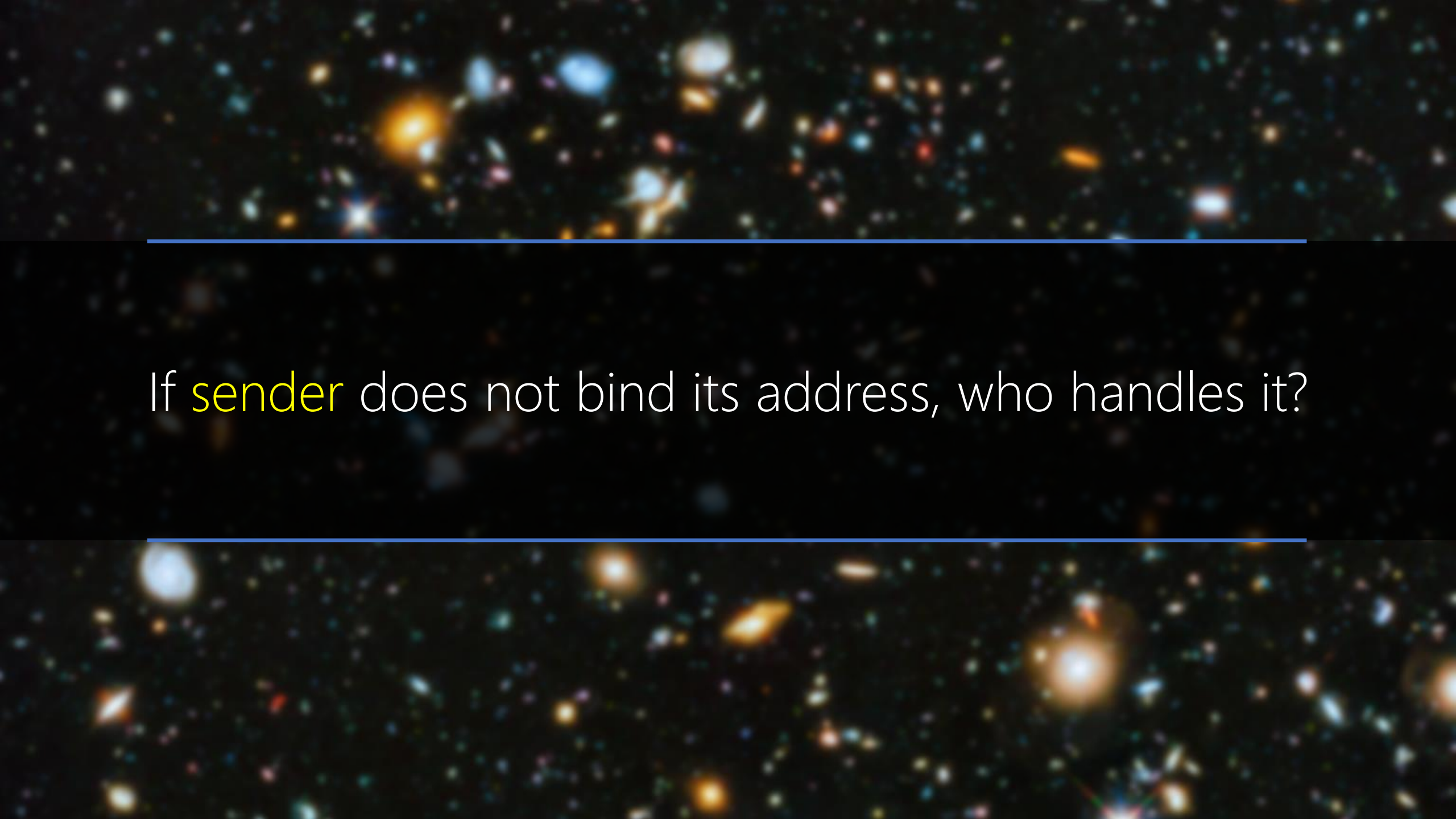
TCP/IP: UDP at Receiver

Connectionless Communication

Sending a mail

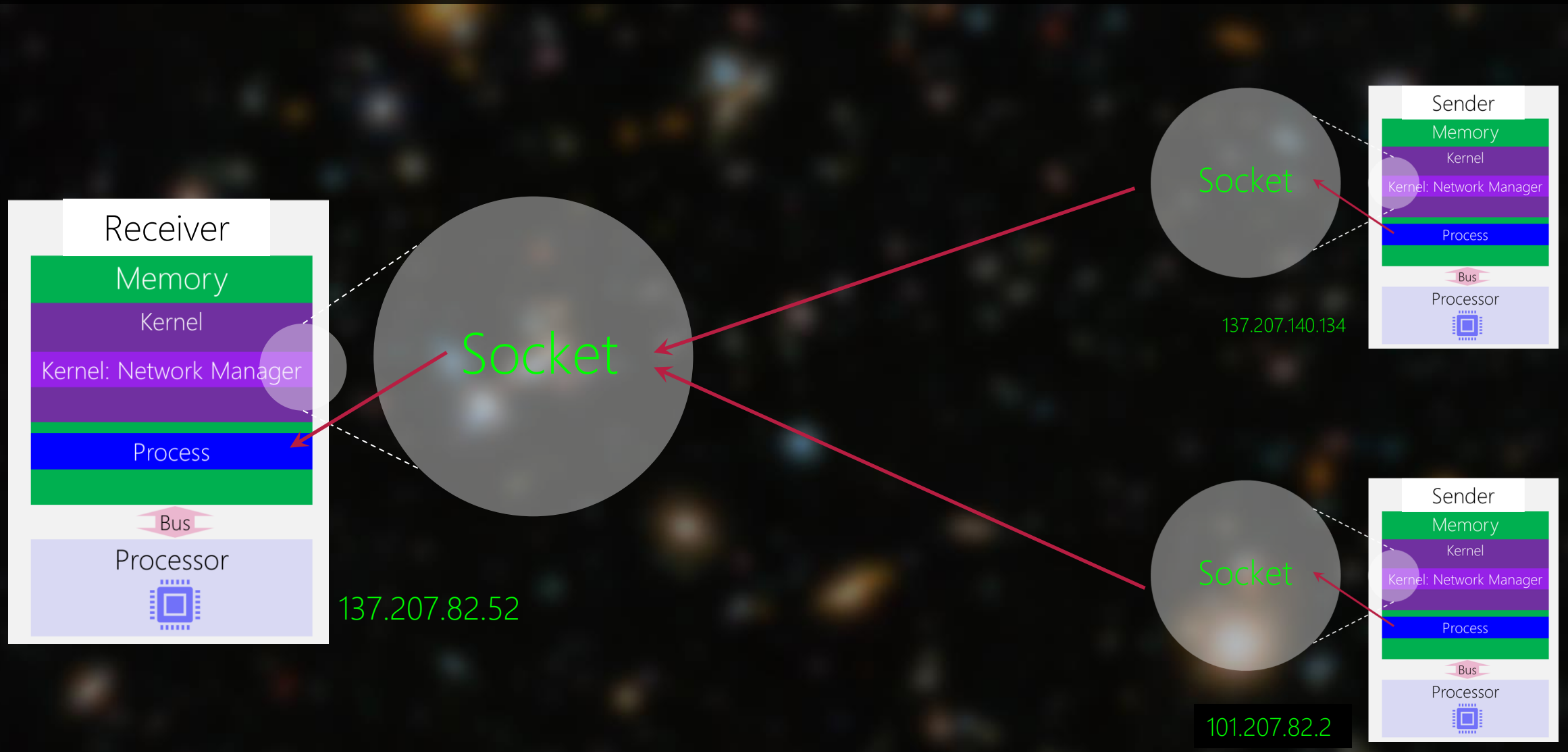


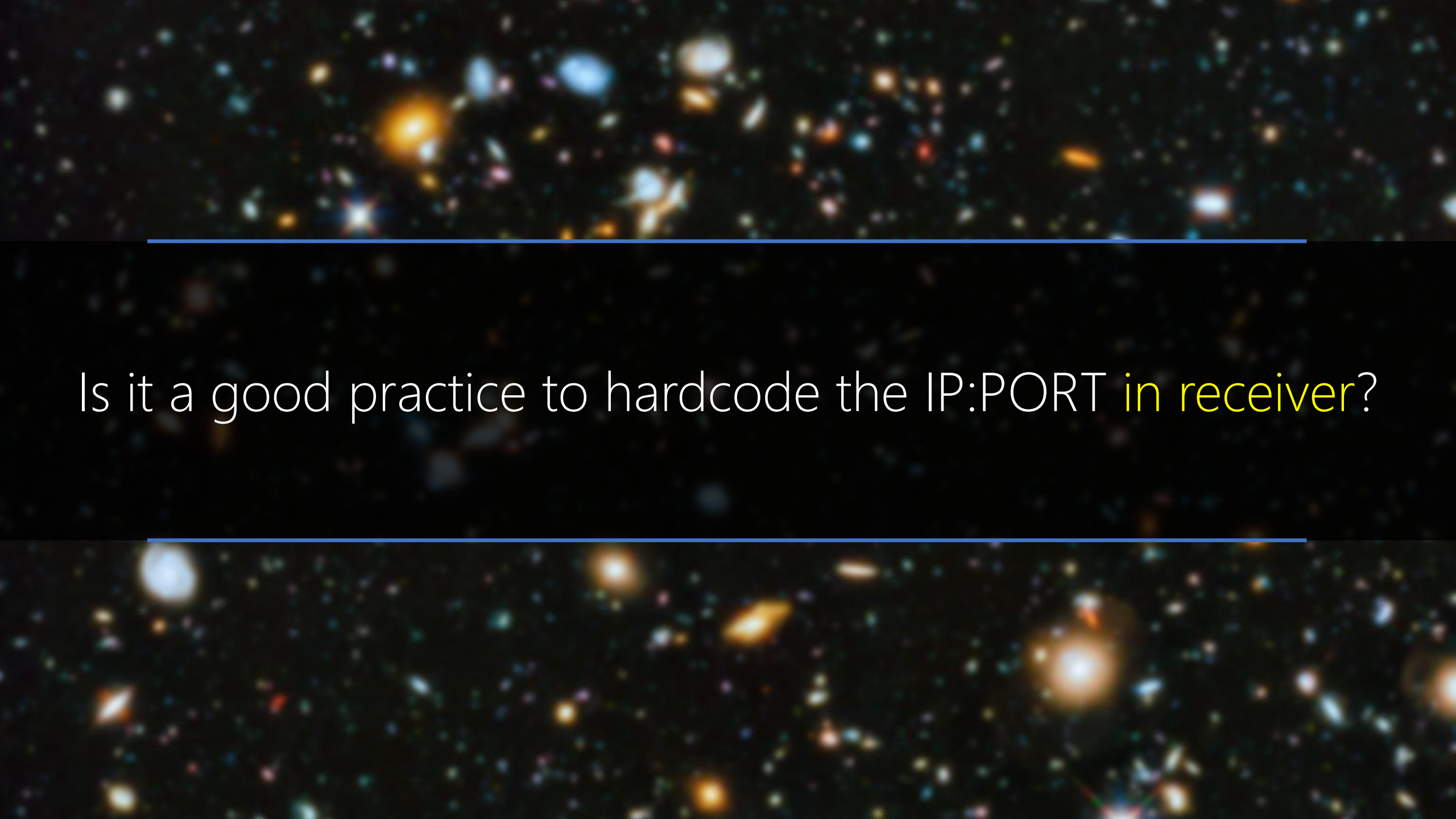
1) `socket()`
2) `bind()`
3) `recvfrom()`
4) Find the Sender's Address (Optional)
5) Read the Mail from the Sender


If sender does not bind its address, who handles it?

TCP/IP : UDP

Many senders, single receiver




Is it a good practice to hardcode the IP:PORT in receiver?

Socket Programming

TCP/IP: TCP

Connection-Oriented, Reliable, Ordered

Lab 11