**Ping&watchdog - programming in C**

Authors:

Maor Saadon 318532421

Dovi Amiram 305677494

**Contents**

**1 System Characterization 2**

1.1 System Overview

1.1.1 About the System………………………………………………………..…………..3  
1.1.2 How to Install and Run the Program………………………….……..……..7

1.2 System Functionality

1.2.1 Code Description……………………………………………………..…………8-19  
1.2.2 Output…………………………………………………………………………………..21  
1.2.3 Functions………………………………………………………………………….22-31

**2 Research findings 34**

2.1 Wireshark

Part A…………………………………………………………….……………….34-36  
Part B……………………………………………………………………………37-39

2.2 Bibliography………………………………………………………………………..……………..48

**1 System Characterization**

**1.1 System Overview**

**1.1.1 About the System**

Part A:

The ping command is used to check the connection between 2 machines. In part A, we will implement the “ping” command.

we will write a program called “ping.c” which will get an argument indicating which host to ping.

The Internet Control Message Protocol (ICMP) is a supporting protocol in the Internet protocol suite. It is used by network devices, including routers, to send error messages and operational information indicating success or failure when communicating with another IP address, for example, an error is indicated when a requested service is not available or that a host or router could not be reached. ICMP differs from transport protocols such as TCP and UDP in that it is not typically used to exchange data between systems, nor is it regularly employed by end-user network applications (with the exception of some diagnostic tools like ping and traceroute).

ICMP ECHO REQUEST AND ICMP-ECHOREPLY - The ICMP echo request and the ICMP echo reply messages are commonly known as ping messages. Ping is a troubleshooting tool used by system administrators to manually test for connectivity between network devices, and also to test for network delay and packet loss.

The program will send an ICMP ECHO REQUEST to the host, and when receiving ICMP-ECHOREPLY, the program will send the next ICMP ECHO REQUEST (no need to stop).

Part B:

Watchdog is a timer to detect and recover your computer dis-functions or hardware fails. It’s a chip whose sole purpose is to receive a signal every millisecond from the CPU.

It will reboot the system if it hasn’t received any signal for 10 seconds (mostly when hardware fails).

We will modify the ping program and write a watchdog that will hold a timer (TCP connection on port 3000) to ensure that if we don’t receive an ICMP-ECHO-REPLY after sending an ICMP-REQUEST for 10 seconds, it will exit.

We will modify the ping.c program so that it will execute the watchdog.c program as well using fork + exec.

Every time better\_ping.c sends a packet, we will update watchdog.c timer.

**1.1.2 How to Install and Run the Program**

To test the system for yourself, you would need a Linux based operating system.

Instructions:

1. Download the following files:
   1. ping.c
   2. better\_ping.c
   3. watchdog
   4. Makefile
2. Put all of the above files in a single directory.
3. Open said directory in your Linux terminal.
4. Run the following commands:
   1. sudo apt install build-essential
   2. Make
   3. sudo ./parta <IP> - for part A
   4. sudo ./partb <IP> - for part B
5. When you want to close Part A you need to press on Ctrl+c

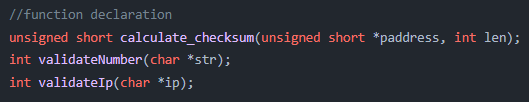
**1.2 System Functionality  
1.2.1 Code Description**

**ping.c:**

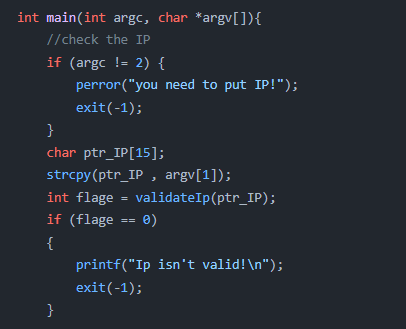
1. The program defines the ICMP header length:



1. Next, the program declares 3 functions implemented below the main code (see detailed explanation below):

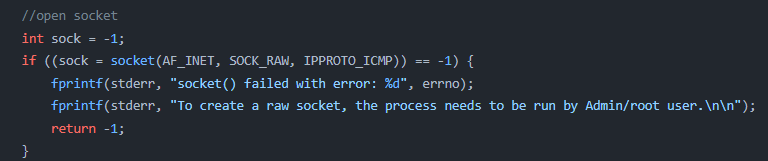
  
The calculate\_checksum() function calculates the checksum of an ICMP header. The validateNumber() function checks if a string is a valid number. The validateIp() function checks if a string is a valid IP address.





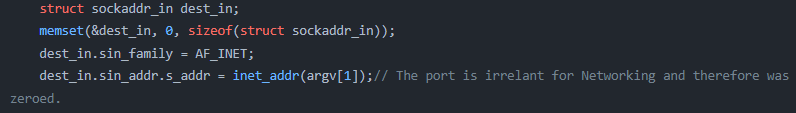
The main function takes two arguments: the number of command-line arguments (argc) and an array of strings containing the arguments (argv). The program first checks if the user provided exactly one command-line argument (the IP address). If not, it prints an error message and exits. It then copies the IP address from argv[1] to the string ptr\_IP, and checks if the IP address is valid using the validateIp function. If the IP address is not valid, it prints an error message and exits.

1. Creating a raw socket:

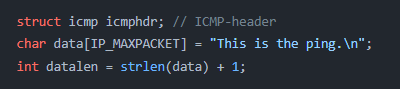


The socket() function creates a new socket and returns a file descriptor for it (for a more detailed explanation – see below).

1. The program then sets up the destination address for the socket:

  
It does this by creating a sockaddr\_in structure and setting the address family to AF\_INET and the IP address to the provided IP address.

1. Declaring an ICMP header structure and creating a data packet containing a message::

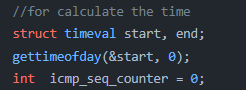


The IP\_MAXPACKET macro is the maximum size of an IP packet. The data packet is a string containing a message, and the datalen variable is the length of the data packet.

1. Printing a message indicating that it is pinging the target IP address:

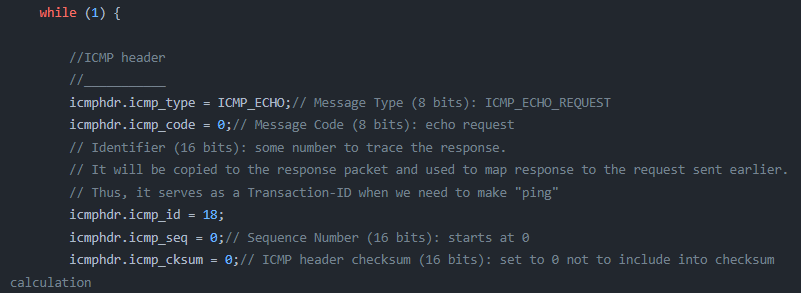


1. Declaring a timeval structure for measuring the round-trip time of each ping:



The gettimeofday function gets the current time and stores it in the start structure.

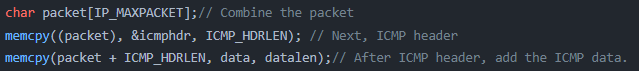
1. The program then enters a loop where it sends a ping, waits for a reply, and prints a message based on whether a reply was received or not. It also calculates the round-trip time for each ping and keeps track of the number of successful and failed pings:



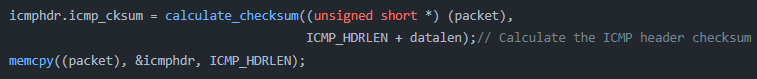
The ICMP header is initialized with the following values:

* 1. icmp\_type: set to ICMP\_ECHO to indicate that this is an echo request packet
  2. icmp\_code: set to 0
  3. icmp\_id: set to 18
  4. icmp\_seq: set to 0
  5. icmp\_cksum: set to 0

1. Combining the ICMP header and the data packet into a single packet:



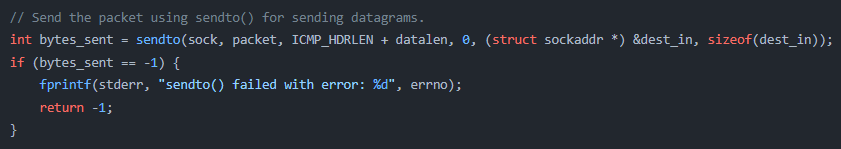
1. Calculating the checksum of the ICMP header and stores it in the header:



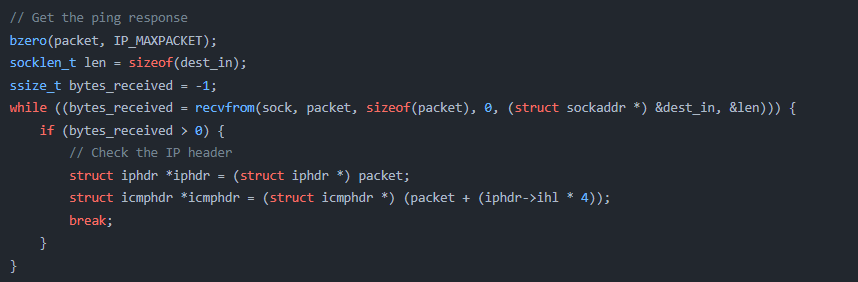
1. Waiting for 1 second before sending the packet:



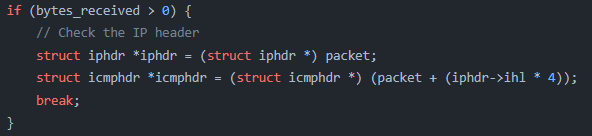
1. Sending the packet using the sendto() function (see detailed explaination below):



1. Clearing 'packet' char array and calling the recvfrom function is to receive a packet from the host.

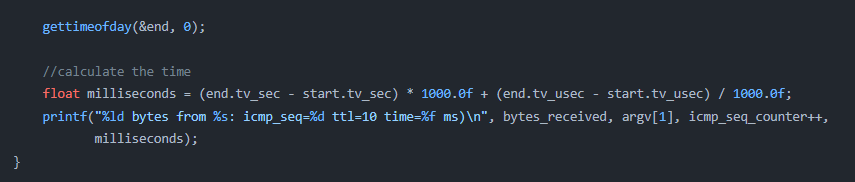
  
The recvfrom function returns the number of bytes received, or -1 if an error occurs. The loop continues until a response is received (i.e., bytes\_received is greater than 0).

1. Once a response is received, the IP header and ICMP header are extracted from the packet:



The IP header is at the beginning of the packet, and the ICMP header follows the IP header. The length of the IP header is specified in the ihl field of the iphdr structure.

1. Finally, the round-trip time is calculated and printed:



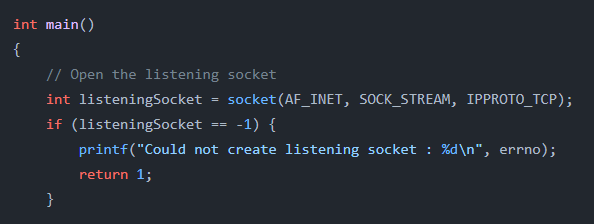
**watchdog.c:**

1. WATCHDOG\_PORT is defined as 3000:



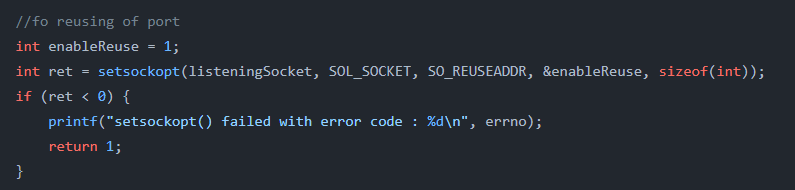
This is the port number that the watchdog program will listen on.

1. The main() function is then defined, which is the entry point of the program:



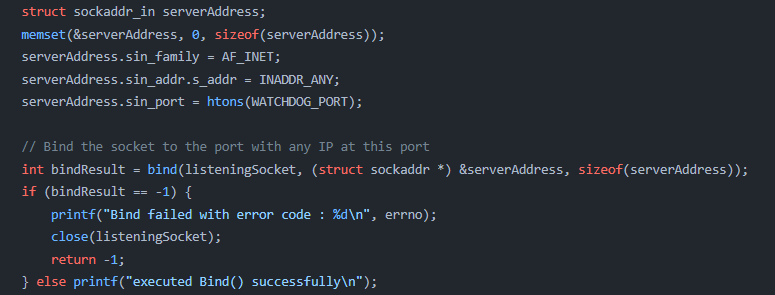
The socket() function is called to create a new socket.

1. The following code enables the reuse of the port on the listening socket:



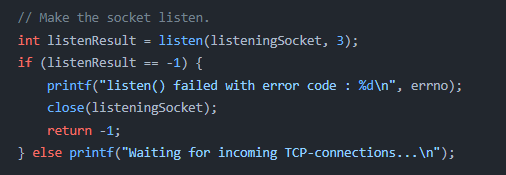
The setsockopt() function is called with the SO\_REUSEADDR option to allow the reuse of the port on the listening socket. This is useful if the program needs to bind to a port that is still in the process of being released by the operating system after the program previously exited. The enableReuse variable is set to 1 to enable the reuse of the port, and the size of this variable is passed as the fifth argument. If the setsockopt() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

1. The following code creates a sockaddr\_in structure to hold the server address and port, and binds the listening socket to this address:



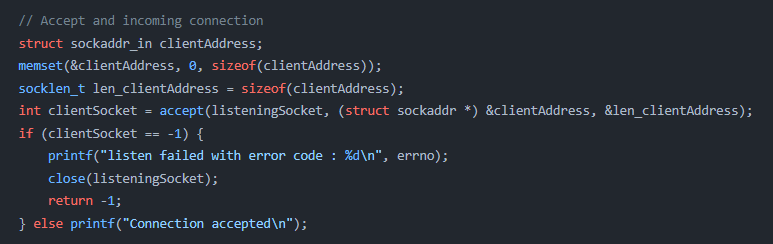
The memset() function is called to clear the serverAddress structure with zeros. The sin\_family field is set to AF\_INET to specify that the address is an IPv4 address, the sin\_addr field is set to INADDR\_ANY to specify that the socket should listen on any available IP address, and the sin\_port field is set to the WATCHDOG\_PORT macro value, converted to network byte order with the htons() function.  
The bind() function is then called to bind the listening socket to the serverAddress structure. If the bind() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

1. The following code starts listening for incoming connections on the listening socket:



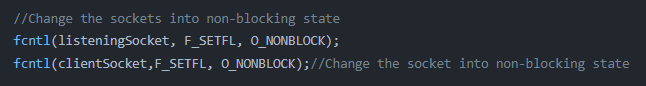
The listen() function is called with a backlog of 3 to specify the maximum number of pending connections that the operating system should allow. If the listen() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

1. The following code accepts an incoming connection and creates a new client socket:



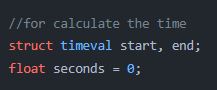
The accept() function is called to accept an incoming connection and create a new client socket. The clientAddress structure is used to hold the client's address and port, and the len\_clientAddress variable is used to store the size of this structure. If the accept() function fails, it returns -1, and an error message is printed to the console indicating the error code (stored in the errno global variable).

1. The following code sets both the listening socket and the client socket to non-blocking mode:



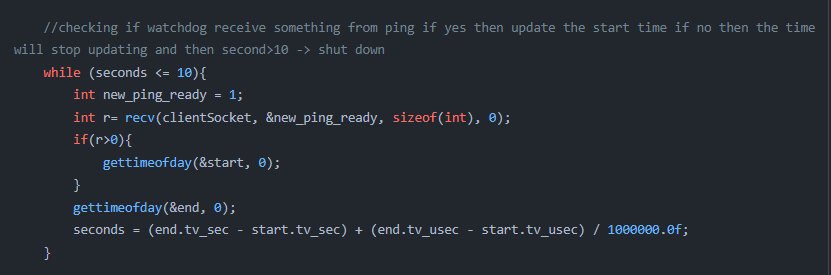
The fcntl() function is called with the F\_SETFL command to set the file descriptor flags for both the listening socket and the client socket. The O\_NONBLOCK flag is passed to set the sockets to non-blocking mode. This means that the sockets will not block when the recv() function is called later in the program, allowing the program to continue execution if there is no data available to be received.

1. The following code initializes the start and end timeval structures, and declares the seconds variable:



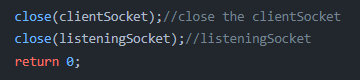
The start and end structures will be used to store the start and end times of the program, and the seconds variable will be used to hold the elapsed time between the start and end times.

1. The following code enters a loop where it waits for a message from the client:



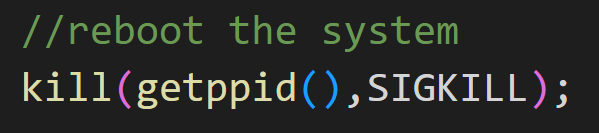
The recv() function is called to receive a message from the client. If the recv() function returns a value greater than zero, it means that a message was received, and the start time is reset using the gettimeofday() function. The end time is then also retrieved using gettimeofday(), and the elapsed time between the start and end times is calculated and stored in the seconds variable. If the elapsed time is greater than 10 seconds, the loop breaks.

1. The following code closes the client socket and the listening socket:



The close() function is called to close both the client socket and the listening socket. This releases the file descriptors associated with the sockets and allows them to be reused.

1. The following code shuts down the system:



In order to finish the program when the timer greater than 10 seconds we use the kill() function sends a signal to a process or process group specified by pid.

**1.2.2 Output**

Part a:

Sending a ping massage until the program us terminated by the user (^c):

A screenshot of a computer

Description automatically generated with medium confidence

Part b:

Timing the time between each ICMP request and response, and announcing when the server is unreachable (over 10 secs to respond)

**Text

Description automatically generated**

**1.2.3 Functions**

ping.c:

**socket():**

**int socket(int domain, int type, int protocol);**

creates a new socket. Takes three arguments: the domain of the socket (e.g., AF\_INET for IPv4, AF\_INET6 for IPv6), the type of the socket (e.g., SOCK\_STREAM for a stream socket, SOCK\_DGRAM for a datagram socket), and the protocol to be used (e.g., 0 for the default protocol). It returns a socket descriptor that can be used to identify the socket in subsequent function calls, or -1 if the socket cannot be created.

**strcpy() :**

**char \*strcpy(char \*dest, const char \*src);**

Copies the contents of a null-terminated string pointed to by src to the memory location pointed to by dest. It returns a pointer to dest.

**memset():**

**void \*memset(void \*s, int c, size\_t n);**

Fills a block of memory with a specified value. It takes three arguments: a pointer to the memory location to be filled, the value to be written, and the number of bytes to be written. It returns a pointer to the memory location.

**inet\_addr():**

**in\_addr\_t inet\_addr(const char \*cp);**

Converts a string representation of an IPv4 address in dot-decimal notation to a 32-bit binary number in network byte order. It returns the binary representation of the address if the conversion is successful, and INADDR\_NONE if the conversion fails.

**memcpy():**

**void \*memcpy(void \*dest, const void \*src, size\_t n);**

Copies a block of memory from one location to another. It takes three arguments: a pointer to the destination memory location, a pointer to the source memory location, and the number of bytes to be copied. It returns a pointer to the destination memory location.

**gettimeofday():**

**int gettimeofday(struct timeval \*tv, struct timezone \*tz);**

Gets the current time of day. It takes two arguments: a pointer to a struct timeval, which will be filled with the current time, and a pointer to a struct timezone, which will be filled with the current timezone. It returns 0 on success and -1 on failure.

**sleep():**

Causes the calling process to suspend execution for a specified number of seconds. It takes a single argument: the number of seconds to sleep.

**sendto():**

**ssize\_t sendto(int sockfd, const void \*buf, size\_t len, int flags,**

**const struct sockaddr \*dest\_addr, socklen\_t addrlen);**

Sends a message to a socket. It takes six arguments: a socket descriptor, a pointer to the message to be sent, the length of the message, a set of flags, a pointer to a struct sockaddr, and the length of the struct sockaddr. It returns the number of bytes sent on success and -1 on failure.

**bzero():**

**void bzero(void \*s, size\_t n);**

Sets all the bytes in a block of memory to zero. It takes two arguments: a pointer to the memory location to be filled with zeros, and the number of bytes to be set to zero. It returns no value.

**recvfrom():**

**ssize\_t recvfrom(int sockfd, void \*buf, size\_t len, int flags,**

**struct sockaddr \*src\_addr, socklen\_t \*addrlen);**

recvfrom() places the received message into the buffer buf. The caller must specify the size of the buffer in len.  
If src\_addr is not NULL, and the underlying protocol provides the source address of the message, that source address is placed in the buffer pointed to by src\_addr. In this  
case, addrlen is a value-result argument. Before the call, it should be initialized to the size of the buffer associated with src\_addr. Upon return, addrlen is updated to contain the actual size of the source address. The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.  
If the caller is not interested in the source address, src\_addr and addrlen should be specified as NULL.

**calculate\_checksum(unsigned short \*paddress, int len):**

This function calculates the checksum of a given input buffer: paddress, of length: len. It does this by performing a series of bitwise operations on the data in the buffer.

**validateNumber(char \*str):**

This function checks if the input string: str, consists of only digits. If it does, it returns 1, otherwise it returns 0.

**validateIp(char \*ip):**

This function checks if the input string: ip, is a valid IP address. It does this by splitting the string by the '.' delimiter and checking if each part consists of only digits and is between 0 and 255. If all these conditions are met, it returns 1, otherwise it returns 0.

**close():**

**int close(int fd);**

The close function is used to close a file descriptor, such as a socket descriptor. It takes a single argument, which is the file descriptor to be closed. This releases any resources associated with the socket and prevents any further communication through the socket.

better\_ping.c:

1. All the functions in ping.c (see above)
2. Additional functions:

**fork():**

**pid\_t fork(void);**

creates a new process by duplicating the calling process. The new process is called the child process, and the calling process is called the parent process. The child process is an exact copy of the parent process, except for the return value of the fork function. The fork function returns the process ID of the child process to the parent process, and 0 to the child process.

**execvp():**

**int execvp(const char \*file, char \*const argv[]);**

replaces the current process image with a new process image. It takes two arguments: the name of the file to be executed and an array of pointers to null-terminated strings that represent the arguments to the new process. The last element of the array must be a null pointer. It returns no value, but if the execution of the new process image is successful, the calling process is replaced by the new process and does not return. If the execution fails, execvp returns -1 and sets the global variable errno to indicate the error.

**wait():**

**pid\_t wait(int \*wstatus);**

waits for a child process to change its state. It takes two arguments: a pointer to a pid\_t variable that will be filled with the process ID of the terminated child process, and an optional pointer to a status variable that will be filled with the exit status of the terminated child process. It returns the process ID of the terminated child process, or -1 if there are no child processes or if an error occurs and sets the global variable errno to indicate the error.

watchdog.c:

1. socket(), memset(), fcntl(), sleep(), gettimeofday() close() (see detailed explanation above).
2. Additional functions:

**Bind():**

**int bind(int sockfd, const struct sockaddr \*addr,**

**socklen\_t addrlen);**

When a socket is created with socket(), it exists in a name space (address family) but has no address assigned to it. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called “assigning a name to a socket”.  
It is normally necessary to assign a local address using bind() before a SOCK\_STREAM socket may receive connections.  
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

**Listen():**

**int listen(int sockfd, int backlog);**  
listen() marks the socket referred to by sockfd as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept().  
The sockfd argument is a file descriptor that refers to a socket of type SOCK\_STREAM or SOCK\_SEQPACKET.  
The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow.

If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.  
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

**Accept():**

**int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen);**  
The accept() system call is used with connection-based socket types (SOCK\_STREAM, SOCK\_SEQPACKET). It extracts the first connection request on the queue of pending connections for the listening socket, sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket. The newly created socket is not in the listening state. The original socket sockfd is unaffected by this call.  
The argument sockfd is a socket that has been created with socket(), bound to a local address with bind(), and is listening for connections after a listen().  
The argument addr is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer. The exact format of the address returned addr is determined by the socket’s address family. When addr is NULL, nothing is filled in; in this case, addrlen is not used, and should also be NULL.  
The addrlen argument is a value-result argument: the caller must initialize it to contain the size (in bytes) of the structure pointed to by addr; on return it will contain the actual size of the peer address.  
The returned address is truncated if the buffer provided is too small; in this case, addrlen will return a value greater than was supplied to the call.   
If no pending connections are present on the queue, and the socket is not marked as nonblocking, accept() blocks the caller until a connection is present. If the socket is marked nonblocking and no pending connections arenpresent on the queue, accept() fails with the error EAGAIN or EWOULDBLOCK.

**setsockopt():**

**int setsockopt(int sockfd, int level, int optname,**

**const void \*optval, socklen\_t optlen);**

Used to set options on a socket. It takes three arguments: the socket descriptor, the level at which the option is defined, and the option name. It can be used to modify a variety of options, including socket timeouts, buffer sizes, and the type of service provided.

**fcntl():**

**int fcntl(int fd, int cmd, ... /\* arg \*/ );**

Performs file control operations on a file descriptor. It takes three arguments: the file descriptor on which to perform the operation, the command to be executed, and an optional argument that depends on the command. It returns a result that depends on the command, or -1 if the command fails and sets the global variable errno to indicate the error.

**kill():**

**kill [options] <pid> [...]**

The kill() function sends a signal to a process or process group specified by pid. The signal to be sent is specified by sig and is either 0 or one of the signals from the list in the <sys/signal. h> header file. The process sending the signal must have appropriate authority to the receiving process or processes.

**2 Research findings**

**2.1 Wireshark**

When we put <IP> = 1.1.1.1 on part A

**Table

Description automatically generated with low confidence**

When we put <IP> = 1.1.1.1 on part B

**Text

Description automatically generated**

**2.5 Bibliography**

Ping

<https://www.geeksforgeeks.org/ping-in-c/>

For checking the IP

<https://www.tutorialspoint.com/c-program-to-validate-an-ip-address>

Watchdog

https://www.youtube.com/watch?v=xJb-btYIYYA&ab\_channel=udemyjobzzsolutions

Fork

https://www.geeksforgeeks.org/fork-system-call/

About the functions  
‘man’ command on vs terminal

For many things

[ChatGPT: Optimizing Language Models for Dialogue (openai.com)](https://openai.com/blog/chatgpt/)

<https://www.google.com/search?gs_ssp=eJzj4tTP1TcwMU02T1JgNGB0YPBiS8_PT89JBQBASQXT&q=google&oq=googlr&aqs=chrome.1.69i57j46i10i131i199i433i465i512j0i10i131i433i512l4j69i60j69i65.3825j0j4&sourceid=chrome&ie=UTF-8>