Chapter 11

Networking

In modern times, the network has emerged as a vital component of operating systems, demanding reliability and easy accessibility. With these key factors in mind, we managed to develop a flexible and scalable module, built upon a theoretically robust foundation.

11.1 The design

The fundamental concept behind our network design was to establish a modular structure akin to the intended architecture of the OSI network layer. As a result, each component possesses minimal knowledge about the layers below it and simply needs to invoke the "intermediate" function, which cascades the request to the network driver. This modular approach enables effortless addition of new protocols, requiring the inclusion of a handler function in the preceding layer [11.7].

In NapoliOS, the network is implemented as a separate service within a distinct domain, adhering to the microkernel structure of the operating system. Although there exists a single instance running on core 0, it is readily accessible from all cores. Notably, the UMP latency remains imperceptible and competes favorably with the basic LMP mechanism, which is limited to 8 bytes per message. The network service is initiated in the `bsp_main()` function and the boot process concludes only after ensuring its readiness.

Consequently, all relevant functionality has been consolidated into a domain named "network", leading to the transformation of the former enet driver into a library.

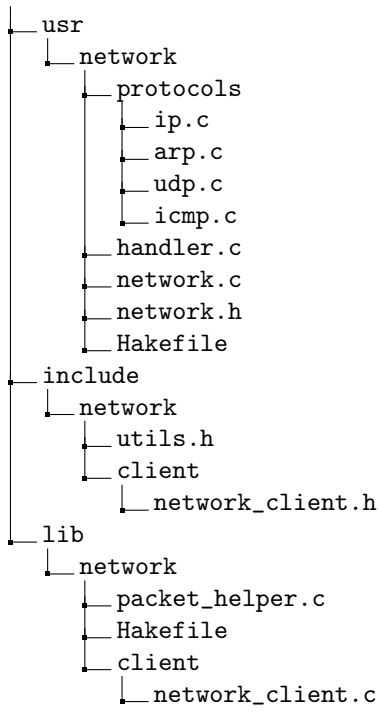
One of the primary challenges encountered during this approach was establishing a reliable communication between the server and the applications. As our system lacked a process-to-process communication mechanism, we designed a solution that could effectively communicate with as many applications as possible. Further exploration of this subject would be discussed in the following sections [11.6].

It is important to note that the current network implementation is solely suitable

for LAN usage, primarily due to the absence of DHCP (Dynamic Host Configuration Protocol) support. Implementing DHCP was not prioritized during our project, as it represented only a fraction of the overall challenge at hand.

11.1.1 The final folder structure

The final folder structure of the networking module follows a well-organized hierarchy, ensuring clear separation of different components and facilitating ease of development and maintenance. Below is an overview of the final folder structure:



The header file `utils.h` contains various utility functions and data structures used throughout the networking module. It also includes the definitions of different packet types, which have been rewritten for convenience and optimized with the `scalar_storage_order("big-endian")` directive. Every other file will be discussed in its related chapter.

11.2 The network layer

With the enet driver implementation already completed, the remaining task involved granting it the necessary dev capability to access the ethernet registers. This dev capability, essential for interacting with the network hardware, is located on core 0, which led us to make the decision to remain on this core (due to the unavailability of a capability passing mechanism between cores). Acquiring the dev capability was straightforward, achieved by invoking

the `lmp_get_devcap()` function and specifying the appropriate base address of the registers. Once the capability was obtained, the driver could be initialized.

This process involves invoking the new `enet_start()` function, which takes the place of the previous `main()` function in the enet driver. As a result, an `enet_driver_state` struct is generated, encapsulating essential information, such as the device queues. These are used to handle the flow of incoming and outgoing network packets, ensuring seamless communication between the operating system and the network interface. The choice of queue sizes was determined after careful consideration of the expected network traffic and the system's capabilities. The receive queue, with its default configuration of 512 slots and a size of 2048 bytes per slot, accommodates the anticipated incoming network packets. This size was chosen to strike a balance between memory utilization and the system's ability to efficiently handle a substantial volume of incoming data. Conversely, the decision to allocate a single slot for the write queue was based on the analysis of the current system state, which indicated that a single slot was adequate for managing outgoing network packets effectively.

Once the driver is launched, the server divides into two threads to efficiently handle the required services. One thread continuously reads the device queue for incoming packets and routes them to the appropriate handler based on the ethertype. Meanwhile, the other thread listens for RPC calls and manages the corresponding responses.

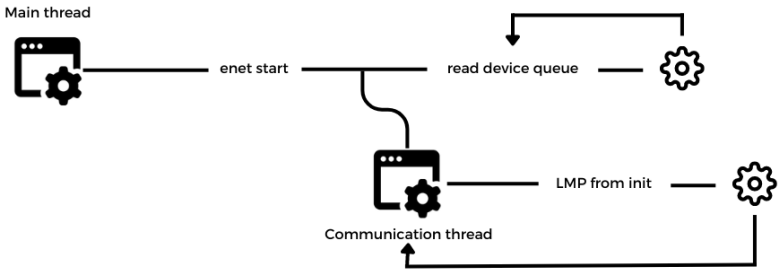


Figure 11.1: Network design

Once this has been completed, an LMP message is sent to the init stating that server is now ready, and the boot process can continue.

11.3 ARP protocol

ARP (Address Resolution Protocol) is a straightforward protocol that requires a few components for its implementation. The most important component is the ARP table, which can be implemented using the Barrelfish collection library. In our case, we decided to use the default implementation with 1013 buckets. The table stores IP addresses as keys and MAC addresses as values. Although this may be excessive for a local network, it reduces the likelihood of collisions.

The `net_handle_arp` function is called whenever a packet with an ethertype of `0x0806` is received by the main thread. The ARP handler first saves the MAC address and IP, even if the packet is not intended for us. Then, one of the following three scenarios may occur:

1. The packet is not intended for us and is dropped.
2. It is an ARP request for us, so we generate an ARP reply by calling `send_arp_reply()`.
3. It is an ARP reply, so we save it in our hash table.

11.3.1 ARP Querying

Every protocol relies on ARP to find the appropriate MAC address for their packets. When a protocol eventually converges to `net_send_packet()` - a function that receives a payload, encapsulates it into an Ethernet packet, and sends it - the MAC address is required. In this function, the `arp_query()` function is called. It returns either the MAC address if it is already stored in the ARP table or sends an ARP request. The thread then sleeps until it receives a response, with a maximum of three attempts. After that, if the address is still not resolved, the packet is dropped.

An important assumption, which is indeed satisfied, is that the thread that needs the MAC address is not the same thread that receives the ARP reply. In fact, only ARP replies and ICMP replies are initiated from the main thread, and they already have the MAC address information.

11.4 IP protocol

The IP protocol serves as the backbone of modern networking, providing the essential addressing and routing mechanisms for data transmission across networks.

While IP itself focuses on addressing and routing, it encapsulates and carries various higher-level protocols, allowing for a diverse range of network services and applications. These higher-level protocols, commonly referred to as IP-based protocols, leverage the underlying IP infrastructure for their data transmission.

Following this structure, our implementation starts from a basic IP module that encapsulates two base function.

- `net_handle_ip(IPv4Packet *pck)`: This function is called when the main thread receives a packet with an ethertype of `0x0800`. It serves as the entry point for processing all IP packets and provides a foundation for registering additional IP-based protocols. The function performs various checks, such as fragmentation detection and checksum validation, to ensure the integrity of received IP packets.
- `send_ip_packet(...)`: This function encapsulates the provided payload into an IP packet and transmits it to the network layer. It allows other protocols or services to utilize IP for transmitting their data packets.

11.4.1 ICMP

ICMP (Internet Control Message Protocol) is a fundamental protocol in the Internet Protocol Suite. It operates at the network layer and is primarily used for network management and diagnostic purposes. ICMP provides a means for network devices to communicate control and error messages to each other, facilitating the exchange of critical information for network troubleshooting, connectivity testing, and error reporting.

ICMP messages are encapsulated within IP packets and typically generated by network devices, such as routers and hosts, in response to specific network events or conditions.

Within our IP implementation, whenever a protocol type of 1 is received, the packet is forwarded to the `net_handle_icmp` function. This function primarily focus on two ICMP message types: *echo request* and *echo reply*. These types are commonly known as "ping" messages and are utilized to test network connectivity and measure round-trip times between devices.

The ICMP implementation within our IP protocol includes the following features:

1. Responding to Echo Requests: Upon receiving an echo request (ping), the IP implementation generates an echo reply by reversing the source and destination addresses and changing the ICMP message type from `REQUEST` to `REPLY`.

2. Initiating Echo Requests: To initiate an echo request, the requesting thread sends an ICMP ping message. Since the response to the ping message will be received by the main thread, the implementation employs a linked list data structure to store and track the ping requests.

```
struct ping_list{
    systime_t t_sent;
    systime_t t_received;
    uint16_t tid;
    uint16_t seq_n;
    uint8_t finished;
    struct ping_list *next;
};
```

   Each request is associated with a unique and incremental transaction ID (TID) combined with a sequence number, allowing easy mapping between requests and their corresponding replies.

3. Ping Utility: Additionally, we provide a ping utility function (`icmp_ping`) that measures the time taken to receive a response from a specified destination IP. This function creates an instance of the `ping_list` struct, sets the `t_sent` timestamp to the current time, and initiates a sleep loop with periodic wake-ups to check if the reply has been received. If a reply is received, the time difference is calculated and returned as a string representing the round-trip time. In case of a timeout, an appropriate error message is returned. This has been set arbitrarily to 500ms.

From this function, we purposely return a string for two reasons:

(a) Giving a timeout as a result is easier to handle.

(b) Some problems occur when calculating a float from the timestamp difference. We suppose the ALU may have some issues with the calculation.

```
>>ping 192.168.0.70
0.339 ms
0.289 ms
0.268 ms
0.241 ms
0.259 ms
0.296 ms
0.259 ms
0.570 ms
0.233 ms
0.208 ms
```

```
alessandro$ ping 192.168.0.69
64 bytes from 192.168.0.69: ... time=0.607 ms
64 bytes from 192.168.0.69: ... time=0.261 ms
64 bytes from 192.168.0.69: ... time=0.315 ms
64 bytes from 192.168.0.69: ... time=0.344 ms
64 bytes from 192.168.0.69: ... time=0.281 ms
64 bytes from 192.168.0.69: ... time=0.295 ms
64 bytes from 192.168.0.69: ... time=0.343 ms
64 bytes from 192.168.0.69: ... time=0.314 ms
64 bytes from 192.168.0.69: ... time=0.326 ms
64 bytes from 192.168.0.69: ... time=0.301 ms
```

Figure 11.2: Pinging from inside and outside NapoliOS

11.4.2 UDP

UDP (User Datagram Protocol) is a simple, connectionless transport protocol that operates within the IP protocol. It provides a lightweight and straightforward method for sending data packets between networked devices. Unlike TCP, UDP does not establish a dedicated connection before transmitting data. Instead, it sends independent datagrams from the source to the destination without any guarantee of delivery or reliability. This lack of connection setup and error recovery mechanisms results in reduced overhead and lower latency compared to TCP. Furthermore, UDP is notably easier to implement.

The simplicity of UDP arises from its minimalistic design. The protocol itself only provides basic features, including source and destination port numbers for multiplexing and demultiplexing data streams, as well as a checksum for data integrity verification. Applications utilizing UDP are responsible for managing their own data delivery and error recovery mechanisms, if required.

Our IP implementation incorporates UDP functionality, triggered when the IP handler receives a packet with a protocol type of 17 (UDP). The UDP module provides basic features, including:

1. Checksum Validation and Generation: The UDP implementation automatically checks and generates the UDP checksum for received and transmitted packets. This is extremely important since the UDP checksum calculation involves a special pseudo-header along with the UDP header and data.

2. Port (De-)Registration: External services or applications can register specific ports through Remote Procedure Calls (RPC). For example, the `NETWORK_REQ` message with type `UDP_REGISTER` is used to associate a service's Process ID (PID) with a desired port. If the port is available, the registration is successful. Otherwise, a `NET_ERR_PORT_BUSY` error is re-

turned.

3. Internal Services: Our implementation supports internal services associated with reserved ports. For example, the echo server (port 7) responds to received packets by sending them back to the source. These internal services are implemented within the network server and offer convenient debugging features.

To handle incoming UDP packets, our implementation employs a hash table for efficient lookup of the destination Process ID (PID) associated with the destination port. This allows for quick and seamless forwarding of packets to the appropriate destination. If the destination is an external service, the packet is added to a write queue, whereas internal services are handled by the `internal_services` handler.

11.4.3 Internal services

Internal services in our IP implementation refer to the reserved ports designated for specific purposes.

The step required to add an internal service are minimal and consists on adding the required port to the global variable `const uint16_t services[]`, and then add your function into the `internal_services` handler. This solution is not the cleanest, but has been adopted to avoid adding more data structures to the code base.

In our minimal version of the OS, we have implemented three internal services:

1. Echo Server (Port 7): The echo server is a basic feature that responds to received packets by sending them back to the source. It serves as a fundamental testing tool for network connectivity. This is easily done by calling the `send_udp_packet` function and inverting the source and sender.

2. Download Speed Measurement (Port 8008): This service provides metrics in Mbps to measure download speeds. It operates by waiting for a packet with the content "start", measuring the total number of received bytes on that port, and calculating the Mbps when a packet with the content "stop" is received. It is important to note that this functionality is primarily intended for the performance section and is not currently designed to be provided as a service for other domains.

3. Upload Speed Test (Port 8009): To complement the download speed measurement service, we have implemented an upload speed test on port 8009. This feature provides metrics for measuring the upload speed from the network to a specified server. The implementation of the upload speed test follows these steps:

   (a) Wait for a packet containing the IP address and port of the server to flood with upload traffic.

   (b) Send the "start" command to initiate the test.

   (c) Start sending maximum-sized UDP packets to the specified server, simulating high upload traffic.

(d) After a specific duration, send the "stop" command to terminate the test.

This upload speed test complements the download speed measurement service, providing comprehensive metrics for evaluating network performance in both upload and download scenarios.

11.5 Network client

The network client serves as an interface for interacting with the network server, providing a convenient way to communicate with the underlying networking infrastructure. It offers a range of functionalities and RPC (Remote Procedure Call) methods that enable higher-level applications to utilize network services, send and receive packets, and manage network configurations.

To facilitate application development, we have included a packet helper library alongside the network client module. The packet helper library offers various useful functions for packet manipulation and conversion. By including the "net_helper" and "net_client" libraries in an application's Hakefile, both the packet helper and the network client can be easily integrated into the application.

11.5.1 Packet helper

The packet helper library provides a collection of functions for generating and printing Ethernet, IP, and ARP packets. It also includes the necessary declarations for network types.

Key functions within the packet helper library that can be utilized by the client program include:

- `print_eth_packet`: Prints an Ethernet packet to the debug console.
- `print_ip_packet`: Prints an IP packet to the debug console.
- `print_arp_packet`: Prints an ARP packet to the debug console.
- `print_mac`: Prints a MAC address represented as a uint8_t array to the debug console.
- `mac_str`: Converts a MAC address represented as a uint8_t array to a string in colon-separated hexadecimal format.
- `ip_to_str`: Converts an IPv4 address represented as a uint32_t to a string in dot-separated decimal format.
- `str_to_ip`: Converts a string in dot-separated decimal format to an IPv4 address represented as a uint32_t.
- `is_mac_null`: Checks if a MAC address is the null address (all zeros).

11.5.2 Network client

The network client module aims to minimize the application code required for network interactions. It offers a comprehensive set of client-accessible features, including:

1. `client_icmp_ping`: Sends an ICMP ping request to a specified destination IP and returns a string with the ping result.
2. `client_udp_service`: Registers a UDP service on the specified port. If the port is already in use, an error is returned.
3. `client_udp_listener`: Polls the network server until a UDP packet is available, which is then returned. This function acts as a blocking operation.
4. `client_send_udp_packet`: Sends a UDP packet with the specified payload to the specified destination IP and port.
5. `client_echo_udp`: Echoes back the payload of a received UDP packet.

With these functions, building a client application becomes incredibly straightforward. For example, an echo server can be implemented in just a few lines of code:

```
client_udp_service(69);
while(1){
    IPv4Packet *pck;
    client_udp_listener(&pck);
    client_echo_udp(pck);
}
```

11.5.3 (un)SSH

Using the network client and packet helper libraries, we have successfully implemented a remote shell without any additional overhead.

The UDP-Shell implementation follows the same approach as the previously listed client, but with added complexity in parsing the payload and redirecting the output.

Initially, our idea was to spawn a remote shell whenever the UDP handler received a packet containing `ssh` on an available port. However, due to issues with the process manager's stability, we opted for an alternative approach. For a more detailed explanation, please refer to the Shell chapter (9.5.2).

11.6 Handling the requests

Building the network itself was mostly about making the right design choices and considering future code reuse. However, the most challenging aspect of the network was handling requests from processes, particularly due to the absence of a solid foundation for RPC (Remote Procedure Call).

As process-to-process communication was not a target functionality, we initially postponed its implementation for a later stage, which unfortunately did not happen. This decision proved to be highly debatable and eventually had significant repercussions. In hindsight, implementing a direct RPC approach would have been simpler and more effective.

The core idea behind our request handling is that every request must pass through the init process, leading to two possible scenarios:

- If the request originates from core zero, it is forwarded directly from the init to the network server.
- If the request originates from a core other than zero, it first goes from the init of that core to core zero, and then gets forwarded to the network server.

Although this solution undoubtedly affects latency, it also impacts the overall design plans.

Figure 11.3: LMP vs UMP request

11.6.1 First try

In our first design attempt, we proposed an asynchronous server that would send the packet through RPC as soon as it arrives. The mechanism for inter-process communication is similar to how we handled the process manager. However, instead of handling the request in the init process, they are forwarded to the network server.

When the init process receives a request of type `NETWORK_REQ` (via UMP or LMP), it is sent to the `network_handler(lmp_msg_t *msg, char *buffer)` function. If the buffer is empty, it indicates that the request is via LMP, and the remaining data needs to be obtained (due to the 8-byte packet limit). Otherwise, the request can be processed.

Every request follows a specific structure, including the Process ID (PID) that is used by the init process to find the appropriate RPC channel. If the request is sent by the network, the PID corresponds to the receiver. Conversely, if the request is sent by any other domain, the PID would be its domain ID.

The `net_type` is used to determine the type of the request. For example if we wanted to register an UDP service we would use `UDP_REGISTER`. At this point the packet is received by the dedicated thread in the network server, and it can be processed.

As an asynchronous system, we also needed to provide a means of sending packets proactively. This required the receiving packets thread to talk with the

communication thread through a mutex-safe writing queue.

When receiving a packet destined for an external domain, the function `net_add_lmp_req` is called. The communication thread then checks this queue before each read attempt, using a snippet of code similar to the following:

```
while(true){
    // Check for send requests (if the queue contains something then send it)
    net_send_message();
    // Check for incoming requests from other domains
    event_dispatch_non_block(ws);
}
```

This approach of handling clients offers low latencies and low overhead since communication occurs only when necessary. However, issues can arise when multiple clients attempt to access the system simultaneously, leading to race conditions.

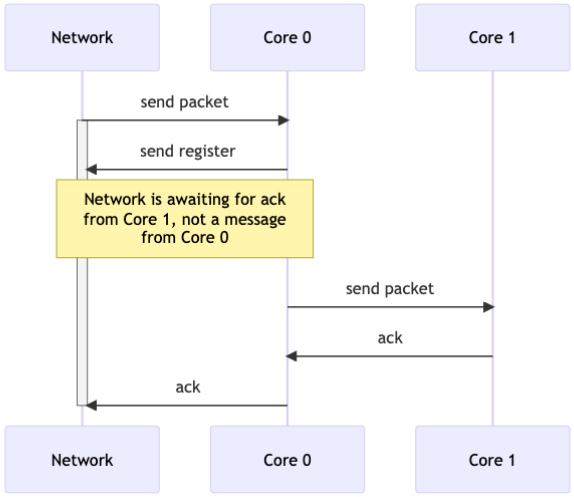


Figure 11.4: Network race condition

From this attempt, we realized that the server could not initiate RPC messages but could only respond to them.

11.6.2 Take two

For the second attempt, we aimed to reuse most of the previous code and make minor modifications. We introduced polling instead of sending packets directly to the write queue. In this approach, the client sends a `POLL_REQ` to inquire if there are any packets with its PID. The server responds with an empty packet or the first packet received in order in the queue. Implementing this solution required adding a single case switch in the logic of the request handler and disabling message sending in the previous `while(true)` loop.

This solution works, but comes with certain disadvantages since polling must

be performed every n seconds (in our client implementation, it is set to 0.1 µs). Consequently, the RPC channel with the init process would be used for every poll request.

11.6.3 The final approach

Due to the constant load on the RPC, unpredictable deadlocks occurred, leading to unreliable behavior. Moreover, the latency was moderate and not acceptable in a network scenario.

To address these challenges, we decided to revert to the first design and tackle the concurrency issue by implementing a small RPC call that creates another LMP channel with the init process (`aos_rpc_create_lmp_chan`). Since the main thread rarely communicates with init and is the one that receives the packets, it made sense to send the packets through that channel. Furthermore, this implementation allowed us to receive responses from the init process regarding the status of individual processes. For example, if a process was terminated, we could deregister it accordingly.

This change resulted in a highly reliable and fast communication, as demonstrated in the performance section.

11.6.4 Future

Given more time, the optimal implementation would combine the first attempt with direct process-to-process communication. This solution would halve the processing time, which is highly valuable in networking scenarios.

11.7 Scalability

A key feature of this network implementation is its flexibility and modularity, which allows for easy integration of new components and protocols. The design follows a layered approach, adhering to the principles of the OSI (Open Systems Interconnection) model, enabling the addition of various internet protocols.

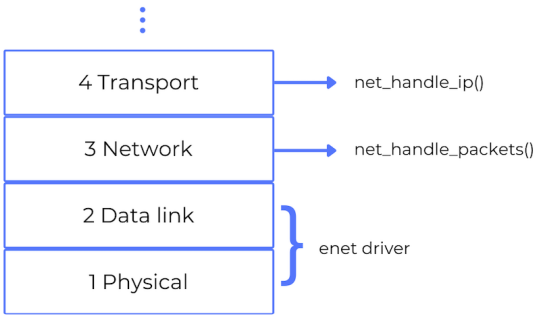


Figure 11.5: OSI stack with handlers

Each protocol within the network stack has a handler that performs common operations. These operations typically involve checking the packet's checksum, as reliability is not guaranteed at this level, and then determining the appropriate upper-level protocol to follow based on the protocol identifier.

Within the network layer, a switch statement is employed to handle upper-level protocols such as ARP (Address Resolution Protocol) and IPv4 (Internet Protocol version 4). Likewise, the IP layer includes a switch statement that manages protocols like UDP (User Datagram Protocol) and ICMP (Internet Control Message Protocol).

To demonstrate the ease of adding new protocols, consider the scenario where we want to incorporate the TCP protocol in a future release. In the code snippet provided below, we can simply add a case for TCP in the switch statement to handle TCP packets:

```
void net_handle_ip(IPv4Packet *pck){
    // Various checks

    switch(pck->header.protocol){
        case 1:
            // Handle ICMP packet
            net_handle_icmp(pck);
            break;
        case 6:
            // Handle TCP packet
            // net_handle_tcp(pck);
            break;
        case 17:
            // Handle UDP packet
            net_handle_udp(pck);
            break;
        default:
            NET_DEBUG("Not implemented protocol, dropping packet\n");
            break;
    }
}
```

By organizing the network stack in this modular and extensible manner, the implementation demonstrates scalability and the potential to accommodate a wide range of protocols and networking requirements.

11.8 Limitation

Due to the limited amount of time to complete the project, obviously there are some features that are still missing or are incomplete.

- **Connectivity**: The network is currently limited to functioning within a local network due to the absence of the DHCP protocol. Without DHCP, the network lacks information on where to route packets if the destination IP is unreachable. Consequently, the subnet functionality remains unimplemented as it relies on DHCP for proper operation. Incorporating the DHCP protocol would not only enable the network to identify the router's location but also provide a more efficient mechanism for IP management, eliminating the need for manual configuration.

- **Latency**: Not having a direct RPC connection with each process using the network, the latency is obviously higher than it should. While the impact of this latency is currently negligible, it can become a critical issue in latency-sensitive services. In such scenarios, the increased latency may significantly impact the overall performance and responsiveness of the system, potentially rendering it unsuitable for real-time or time-critical applications.
- **TCP**: Due to the substantial amount of time invested in debugging and rewriting the communication system, the implementation of TCP functionality has been temporarily set aside.
- **Fragmentation**: As fragmentation is not considered a core functionality of the system, its implementation has been deferred to a future version.

Despite the existing limitations, the system has been purposefully designed with scalability in mind, providing a strong foundation for future development. This inherent scalability allows the system to readily accommodate the implementation of missing features and simplifies their integration into potential future iterations.

11.9 How to run the tests

To evaluate some of the network functionalities, we have included a straightforward test that involves spawning multiple network clients. To execute the test, simply run the following command:

```
1 python3 tools/autograder/autograder.py -t m7_tests_net
```

The content of the client can be found in `./usr/test/net/net_test.c`. This file contains the implementation of a simple client that performs a port registration and echoes back any received packet.

The IP is fixed and currently set to 192.168.0.69. To change it, the definition can be found in the `network.c` file.

Additionally, for the download and upload tests, you can use the provided script in the appendix [A.4]. To run the download test, simply execute the script. For the upload test, send a packet using the following format:

```
alessandro$ nc -u 192.168.0.69 8009
YOUR_IP UPLOAD_SERVER_PORT
```

11.10 Performance

Performance is a critical aspect in networking, and measuring various metrics can provide valuable insights into the network's efficiency and responsiveness.

To evaluate the performance of our network implementation, we utilized a set of tools that allowed us to measure several key metrics. Each metric was measured 100 times, and the median value is reported.

- **Download speed**: This metric quantifies the rate at which data can be sent from external sources to our system. The reported values depend on the type of yield used during packet reception. With the `thread_yield` method, we achieved a download speed of 93 Mbps, which nearly saturates the available bandwidth. However, when using `thread_yield_dispatcher`, the download speed decreases to 76 Mbps.
- **Upload speed**: This metric measures the speed at which data can be sent from our system to external destinations. We obtained an upload speed of 1 Mbps, which is significantly lower than the download speed. This limitation is expected due to having only one slot in the device queue.
- **Ping from OS to external**: This metric assesses the round-trip time (RTT) for sending an ICMP echo request packet from our operating system to an external target, providing an indication of the network's latency. We achieved an acceptable RTT of 0.25 ms. It is worth noting that the overhead of the guest OS may contribute to this latency, as pinging between two instances of NapoliOS yields results comparable to a loopback.
- **Ping from external to OS**: This metric measures the RTT for sending an ICMP echo request packet from an external source to our operating system, evaluating the network's responsiveness from external sources. Similar to the previous metric, we obtained an RTT of 0.25 ms.
- **Loopback ping**: This metric examines the RTT for sending an ICMP echo request packet within the loopback interface of our system, demonstrating the efficiency of communication between different processes within the same machine. The loopback ping yielded an impressive RTT of 0.01 ms, indicating the high efficiency of the network stack for local communication.

By analyzing these metrics, we can gain valuable insights into the performance characteristics of our network