Design of a Decoupled Network Stack With Raspberry Pis and Handmade NICs

Benjamin Modlin
Colorado College

Tyler Chang
Colorado College

Abstract

The internet's overtaking of the world has caused many fundamentals of computer networking to be forgotten. Whether it's certain protocols to follow or design decisions to make, most networking has become standardized to follow the modern internet, leaving little room for analyzing if the actual best choices are being made. Thus, with limited equipment consisting of Raspberry Pis with built-in network technologies disabled, hand-made network interface cards, and optic cables, we attempted to build a functioning network stack with a scalable codebase in 3 weeks.

By utilizing the 5-layer network model and prior research as our only guidance, we aimed to have a chat program working among a network of Pis. Most of the time was spent working on the physical, data link, and network layers of the model. Due to going into the project with limited experience, we made sure that each layer contained smart decoupled interfaces which provided an ease for modification, interaction, and debugging. With physical cabling being wired successfully and a link layer that had working data transmission and reception, the project was ultimately halted after the implementation of network routing revealed unstable conditions in the link layer. With no time left to debug, the project was forced to be paused. However, this experiment has definitely revealed that the modern internet is built on simply a fraction of the methodologies within computer networking. The reinvention process has unfolded both design advantages and flaws of the internet that we are familiar with today, and a reminder that every standardized procedure at one point was simply a decision made by another person.

1 Introduction

In the modern day of computer usage, the ability for communication to occur between computers has become an expectation that the world has taken for granted. From streaming videos on YouTube to playing online multiplayer games, the computer networking that we are familiar with has dominated our perception of the internet as a network built on an ideology containing specific protocols and design standards. On top of software standards, it is apparent that even commercial off-the-shelf hardware contain these standards ingrained within regardless if it's for personal or enterprise use. However, it is easy to overlook the fact that the modern-day internet that we are familiar with is simply only one way of implementing computer network architecture. The use of Ethernet for connection and following the Internet Protocol (IP) along with developing applications on top of the Hypertext Transfer Protocol (HTTP) may seem like an essential standard for networking, yet it is important to realize that these "rules" were still at one point simply decisions made by people that ended up being massively adapted to the rest of networking.

Therefore, with this knowledge in mind, we performed a 3-week experiment where we attempted to reinvent networking without any of the preset standards. Instead, we would discover and come up with the design choices ourselves based on what we see as best fit for the experiment. By using the Raspberry Pi, a small single-board computer running Linux, along with hand-made network interface cards (NIC), the goal was to achieve a working chat program across multiple Pi's using the NICs. With the limited amount of guidance available, it was agreed upon to design the network based on a 5-layered model stack which involves the following layers: physical, link, network, transport, and application. With no such thing as Ethernet or USB connection in our tool set, this meant starting our physical layer from the most basic fundamentals of transmitting on and off voltages from the Pi to the NIC board LED lights through bits and building our network up from here. Although convenient pre-existing technology may not be accessible anymore, the experiment now gave us full control on all decisions for each layer and provided an opportunity for understanding why modern standardized networking procedures may not always be the best design choices depending on the scenario.

2 Background

Before diving into the experimental work, prior research had to be done on both the hardware and software setup in order to obtain a fundamental understanding towards how to approach each step along the way. Since we were modeling the project after the 5-layer network model which contains the physical, data link, network, transport, and application layers [9], it was important to first examine each layer, especially the physical, data link, and network layers, and research the design decisions for each based on our equipment and what others had success with in the past.

2.1 Raspberry Pi

Beginning with the physical layer, it was essential to first analyze what we would use to represent each computer node in our network. As mentioned before, since most commercial off-the-shelf hardware comes with standardized networking architecture such as Wi-Fi or Ethernet built-in, we wanted a computer without these capabilities that had the sole ability to send and receive voltages to and from the NIC. Therefore, the choice to use the single-board computer Raspberry Pi 3B+ in this project was great as networking capabilities could be disabled and in addition to there being 40 general purpose input/output (GPIO) pins along with a 5 V microUSB port on the board [3]. The GPIO pins will allow for connection to external input and output devices through a ribbon cable [17], and the Raspberry Pi Pinout resource provides a visual reference for the functionalities of each pin [11]. Furthermore, the pigpio software library of Raspberry Pi provides interactions with the pins on the Pi and is compatible with Python [10].

2.2 Network Interface Cards (NIC)

In terms of connecting our Pis to the network, we were given multiple hand-made network interface cards (NIC) [14]. Each card has 4 ports with each port containing a light transmitter unit and a fiber optic receiver. The transmission unit of one port could then be connected to the receiver of another unit through an optical cable and vice-versa. The transmitter and receiver each have a green LED attached to it in order to represent status. A lit-up green on the transmitter means data is going out while a lit-up green on the receiver means data is coming in [7]. Each card also contains a set of GPIO pins for connection with a Raspberry Pi through the use of a ribbon cable.

2.3 Data Link Techniques

When it came to the data link layer, it was important to identify first that Raspbery Pis do not have a shared clock when connected to the same network. In other words, this meant that sending and receiving data would be tricky due to the limitations of syncing two Pis together in a given time. Therefore, research was done in asynchronous serial communication which is a communication interface for when the signals being used (our Pis) are not synchronized to each other on a universal clock signal [15]. This is made possible through the use of start and stop bits which wrap the message being transmitted in order to indicate to the receiver when to start and stop processing the incoming packet [15]. Without these bits, the receiver would have no way of knowing when to start processing a new character in the case of a message [8].

Since asynchronous communication requires a universal rate at which bits will be transmitted and received, it was important to account for potential drift issues which led to the research of bit stuffing. Bit stuffing is the act of inserting non-informational bits into data for the purpose of re-syncing bit rates when drift occurs [16]. Asynchronous communication also involves a high idle state meaning that even when nothing meaningful is happening between two Pis, both are still constantly transmitting and receiving and using start and stop bit detection in order to identify a real message. With the sending and receiving processes having to be always running, it was then also important to research multi-threading in Python. Research revealed that the multiprocessing package [4] of Python provides the most realistic threading properties through processes and contains built-in data structures such as the multiprocessing queue [13] which take care of shared data issues between process with locking mechanisms and pipes.

2.4 Network Routing Techniques

Although there are many challenges in the network layer, we were initially the most interested in researching routing algorithms and the protocols associated with them. Starting from the top was to first identify the classifications of routing algorithms which primarily involve adaptive and non-adaptive algorithms. Adaptive algorithms involve changing routing decisions when there is a modification to the network topology whereas non-adaptive algorithms do not [1]. An interest in adaptive algorithms then led to further research on distributed routing methods that involve neighbor information getting passed around [1]. This ultimately led to the discovery of the distance vector routing (DVR) protocol. This routing method involves having each node keep a distance vector table containing distances to its neighbors. Each node would then exchange distance tables with their neighbors and apply the Bellman Ford algorithm [18] in order to compute the shortest path to other nodes [6].

2.5 Transport and Application

The transport layer is in charge of delivering data to the

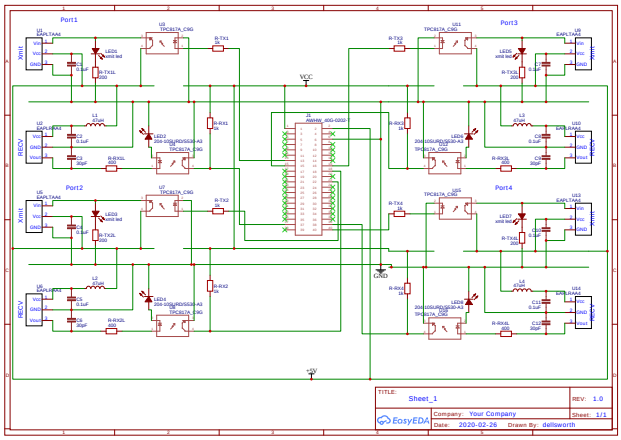

Figure 1: Raspberry Pi to NIC Connections

appropriate computers and their processes. This is typically done through the usages of two popular transport protocols: transmission control protocol (TCP) and user datagram protocol (UDP). TCP involves a three-way handshake between two nodes before any data is sent whereas UDP does not [5]. Since this project is an experiment with no certainty on its possibility, getting a single chat application running on the network would be considered victory and hence, the transport layer was not of the highest priority due to the current goal of only having one application running for now.

3 Design

With us both coming into this project with little experience in computer networking, we did not know what roadblocks we would expect. Therefore, from the beginning, we tried to make our codebase as resilient as possible to unexpected requirements. Our main goal throughout the project was to decouple each layer from each other and build smart interfaces to avoid a large refactor each step we took. Our repository we used most of the project can be found in this Github repository. Since each layer in our network model abstraction built on all the layers below it, we started with the lowest layers.

3.1 Physical Layer

Since we were provided with Raspberry Pis and prebuilt NICs in the beginning of the class, the physical layer was almost complete. The main challenge with the physical layer was implementing the functions in the Pigpio library [10] to control the transmitting ports on the NIC and read from the receiver ports. Figure 1 outlined the Raspberry Pi GPIO pins we should interact with to control the ports from our code.

3.2 Link Layer

The link layer was by far the most difficult layer to work with for our group. On the surface, it seems easy to setup the clocks to set the optical cables and read them at the same clock-speed. Unfortunately, nodes on our network need to send significant amounts of bits to each other. In order to do that quickly enough, our clock-speeds needed to be set faster and faster. As the clocks got faster, clock drift and sleep time inconsistency became a large problem. Even at around 10-100 bits / second, far slower than modern protocols, the messages would be unintelligible and filled with mistakes with no clock correction.

We knew we needed a way to detect clock drift and adjust the time we were sleeping the computer between each bit. In our first implementation of our link layer, we used the Pigpio callback function to immediately detect the switches on the optical cable in our receiver function and record the time. [10] We could then alter the next sleep time so we read the line as far from the clock ticks as possible. This would drastically lower the chance of clock drift causing a mistake, and worked as a link layer at around 100 bits / second.

We also needed a way to know the beginning and end of each message. Instead of implementing more complicated headers, we used start and stop sequences to relay the beginning and end of each message for our fairly simple network implementation.

It was unlikely that we would encounter long enough runs of zeros or ones without switches that the clock would drift enough to cause a mistake. To be sure, we implemented a bit stuffing/ignoring algorithm in both the receiver and sender to bit stuff the messages to cause switches without changing the filtered message. After n identical bits in a row we would insert the opposite bit. We could set this constant depending on the clock-speed and risk of clock drift.

Next, we needed to be able to send and receive messages at the same time. We attempted to use the built in threads library in python, but then our link layer immediately stopped working. Since the threading library uses concurrency and not parallelism, the multiple threads messes up the timing of the sending and receiving. [12] To fix this, we switched to the analogous library Multiprocessing, which implemented true processes, returning the link layer back to working.

Slightly later in the project, we needed faster than 100 bits / second. Instead of coming up with even more advanced clock correction, we took the suggestion of other groups in our class to use Pipgio's wave serial add function to send faster waves and read them with their serial bit bang functions. This improved our link layer's speed by orders of magnitude.

3.3 Network Layer

The first challenge at the network layer was building receiver and sender queues, to handle sending and receiving

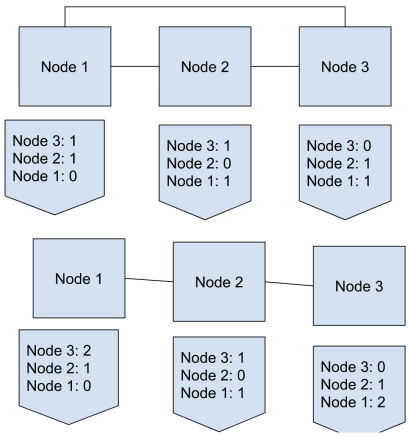

Figure 2: Example Routing Tables for Different Node Shapes

multiple messages at a time from different ports. Since we were now using processes to send and receive at the same time, they did not implicitly have shared memory to easily control the queues. Luckily, the multiprocessing library has an easy to use implementation of a shared memory queue that fulfilled our purposes. [2]

With a functional interface built on top of our link layer, our network layer was far easier to work with. Using the sources outlined in our background section on distance vector routing protocol, we had each network node send out a ping at regular intervals to its neighbors with its routing tables containing the direction (port) and distance on the fastest route to each other node. They were update their tables with new information as new nodes joined the network. One vulnerability in our implementation is that if a connection is severed our network is unable to detect if a node stops sending pings. Since our network is confined to a classroom and we control the connections, we chose not to add this function to save time.

The tables that were pinged by each node were serialized and deserialized by JSON functions. The continuously updating table was accessible from all necessary processes by using shared memory multiprocessing dictionaries. [12] These dictionaries, like Multiprocessing queues, automatically were locked by Multiprocessing to prevent race conditions.

With the tables accessible to each process, when we received a message, we included an address stop bit to know the final node address was between it and the start bit. The process reading the receiver queue could then either process the message or send it in the right direction. Figure 2 shows how the tables change when a direct connection is added between nodes 1 and 3.

3.4 Application Layer

By the time we got our routing working, we were almost out of time in our class. We started building the application layer with a user interface reading commands from the user to know what "application" to start. We did not have time to make real applications, we got direct messages working partially.

Unfortunately, by the time we were testing routing direct messages between nodes, our Pigpio wave link layer stopped functioning altogether. The messages would come in incorrect and with non-bit symbols. We did not have enough time to debug this issue that may have been hardware limitations or problems with the Pigpio library. We suspect having so many processes running with the application layer being added may have triggered the issue, because our old slower clock correcting link layer was also no longer functioning.

4 Evaluation

While we did run into problems at the end of the class, we greatly improved our intuition with some aspects of underlying networking protocol. Setting aside some issues with the nondeterminism that comes with working with hardware, our codebase was written in a way where building new layers did not require a large amount of refactoring in the layer below.

The interfaces built to run link layer and network layer processes allowed the more abstracted layer to be independent on how the layers below functioned. We could then make necessary edits to lower levels as we built up, an introduced more complexity. With a modern link layer setup, we could most likely build many abstractions without issue.

With the newer link layer with Pigpio waves, our network layer worked quickly for the hardware we were using. We could send short messages rapidly, and longer "text" messages took about a couple of seconds. Obviously our network is not fast enough to run graphics or anything a modern computer would need to do, but the routing tables were sent fast enough to generate updated tables in small networks easily.

5 Future Work

If we had more time to finish the application layer and build a simple transport layer, we could realistically run simple text based applications at each node.

5.1 Reliable Link Layer

In order to make progress on our network, we need our link layer to be reliable again. It is possible we could reduce the processes we are using, switch link layers altogether to another protocol, or use different hardware. It is also possible we made a mistake in the way we implemented our multiprocessing. With some work on our implementation, our Pigpio wave

method could work once again. Another issue that could be causing our link layer to be unreliable is port crosstalk. Some of the other groups noticed port crosstalk with our handmade NICs. Since our link layer stopped working while testing a partial application layer, the increase in traffic in the ports of all the NICs could be introducing a port crosstalk problem we did not have before.

5.2 Transport Layer

If we got to the point where we needed to receive data for multiple applications at the same time, we could add a "port" sequence in our messages. The process reading the receiver queue could then know which application was assigned to each port and send the data to that application. This could be done either with some form of shared memory or data pipes, which are also built-in to the Multiprocessing library. [12] This would be our simple implementation of a transport layer.

6 Conclusion

Although the initial goal of having a chat program running among a network of Pis was not quite reached, it was relieving to know that we were only a reliable link layer away from creating a fully functional network capable of running multiple applications. More importantly, the learning outcomes on computer networking from this experiment has been incredibly substantial. Starting off with NICs that had never been used before, sending and receiving voltages from the Pis to them and seeing the correct lights flick on was a successful start of the physical layer. Looking back, it was incredible to see that the simple task of sending and receiving 1s and 0s served as a reliable foundation for our network and is found in most of our modern technologies. Next came the link layer which was the most difficult to implement. Whether it was understanding asynchronous communication, fixing clock drift, or getting shared memory to work with Python's multiprocessing, the trial and error of this layer has contributed to a significant expansion in knowledge towards making the most optimal decisions for linking two computers, especially thinking outside of common modern methodologies. The networking layer allowed for lots of opportunities to be creative when it came to designing routing. Although we researched and settled on distance vector routing, it was pleasant to learn about all the possibilities that we could've gone with, and furthermore, understanding that different topologies will require different routing procedures. The choice of having decoupled interfaces for each layer also ended up being an excellent decision as we now know exactly what to do going forward.

The idea of there not being a one-size-fits-all solution started to become apparent everywhere regardless of the layer that we were working in. Deeper reflection reveals that this idea symbolizes the whole purpose of the project. Although our invention of networking may contain similarities to the modern internet, a majority of design choices differed vastly and yet we were still on the route to success. The modern web has made choices become standards and the ability to rationalize is often lost, further emphasizing the importance in holistic analysis of the technologies that we have grown to be so comfortable with.

References

[1] Ankit87. Classification of routing algorithms. https://www.geeksforgeeks.org/classification-of-routing-algorithms/, 2021. Online; accessed 10-Oct-2022.

[2] Jason Brownlee. Multiprocessing Queue in Python. https://superfastpython.com/multiprocessing-queue-in-python/. [Online; accessed 0-5Oct-2022].

[3] eTechnophiles. Raspberry pi 3 b+ pinout with gpio functions, schematic and specs in detail. https://tinyurl.com/46hh723a/. [Online; accessed 26-Sep-2022].

[4] Python Software Foundation. Multiprocessing in python. https://docs.python.org/3/library/multiprocessing.html, 2022. Online; accessed 05-Oct-2022.

[5] Pamela Fox. Transmission Control Protocol (TCP). https://www.khanacademy.org/computing/computers-and-internet/xcae6f4a7ff015e7d:the-internet/xcae6f4a7ff015e7d:transporting-packets/a/transmission-control-protocol--tcp. [Online; accessed 28-Sep-2022].

[6] Geeks For Geeks. Distance Vector Routing (DVR) Protocol. https://www.geeksforgeeks.org/distance-vector-routing-dvr-protocol/. [Online; accessed 10-Oct-2022].

[7] Ultimate Tech Hub. What network port lights mean. https://www.youtube.com/watch?v=0haGpWNdGJk. [Online; accessed 27-Sep-2022].

[8] IBM. Start, stop, and mark bits. https://www.ibm.com/docs/en/aix/7.2?topic=parameters-start-stop-mark-bits. Online; accessed 18-Oct-2022.

[9] Karthikayan Mailsamy. 5-layer network model made simplified! https://medium.com/@karthikayanmailsamy/5-layer-network-model-made-simplified-e813da0913ba/, 2020. Online; accessed 18-Oct-2022.

[10] PIGPIO. The pigpio library. https://abyz.me.uk/rpi/pigpio/index.html, 2021. [Online; accessed 27-Sep-2022].

[11] Pinout. Raspberry Pi Pinout. https://pinout.xyz/#/. [Online; accessed 27-Sep-2022].

[12] Python. multiprocessing - process-based parallelism. https://docs.python.org/3/library/multiprocessing.html. [Online; accessed 05-Oct-2022].

[13] Super Fast Python. Multiprocessing queues in python. https://superfastpython.com/multiprocessing-queue-in-python/, 2022. Online; accessed 05-Oct-2022.

[14] Tech Target. What is a network interface card. https://www.techtarget.com/searchnetworking/definition/network-interface-card. [Online; accessed 26-Sep-2022].

[15] VectorNav. Asynchronous Serial Communication. https://www.vectornav.com/resources/inertial-navigation-primer/hardware/asynccomm. [Online; accessed 28-Sep-2022].

[16] Wikipedia. Bit stuffing. https://en.wikipedia.org/wiki/Bit_stuffing. Online; accessed 29-Sep-2022.

[17] Wikipedia. Ribbon Cables. https://en.wikipedia.org/wiki/Ribbon_cable#Color-coding. [Online; accessed 26-Sep-2022].

[18] Wikipedia. Bellman–ford algorithm. https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm, 2022. Online; accessed 10-Oct-2022.