UNIVERSITY OF CALIFORNIA
SANTA CRUZ

BUILDING A NETWORK STACK FOR THE TWIZZLER OPERATING SYSTEM

A thesis submitted in partial satisfaction
of the requirements of the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Barbara Inez Moretto Dama

September 2020

The Thesis of Barbara Moretto Dama
is approved:

Professor Katia Obraczka, chair

Professor Peter Alvaro

Professor Ethan Miller

Quentin Williams
Acting Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

BUILDING A NETWORK STACK FOR THE TWIZZLER OPERATING SYSTEM

by Barbara Inez Moretto Dama

Abstract

This thesis describes the design and implementation of the first version of the Twizzler network stack. In this implementation, we focused on providing Twizzler developers with the network functionality that was immediately needed, while having flexibility that promotes growth and expansion. To encourage research in creating a data-centric network stack, we implemented the FLIP protocol as Layer 3 of the network stack, moving away from the traditional TCP/IP stack. FLIP allows for custom headers and its adaptability gives application developers control over which functions the network provides for their specific application. We also describe the implementation of ARP, the first network application using the Twizzler stack, which allows for network discovery. We expect that the future directions of this work will propel the development of a full-fledged network stack for the Twizzler operating system.

Acknowledgements

I would like to thank Professor Katia Obraczka for not only helping me with this research, but for being a mentor since the beginning of my career at UCSC as an undergraduate student. I would also like to thank Peter Alvaro and Ethan Miller for giving me the opportunity to work on this project and for introducing me to the Twizzler Operating System. Lastly, I would like to thank Daniel Bittman for working closely with me during this project, as we collaborated together in implementing the beginnings of the Twizzler's network stack.

Chapter 1

Introduction

The Twizzler[1] operating system was designed to change the assumptions of the underlying memory format from those of traditional operating systems in order to adapt to newly-available technology, specifically byte-addressable non-volatile memory (NVM). To take the best advantage of what NVM has to offer, an operating system must change its persistent data path allowing user programs to directly access NVM, as we will describe in more detail in Chapter 2. By redesigning the persistent data path, Twizzler provides us with a true global object-ID space for NVM. This global object-ID space allows for persistent data to reside in any location, and as we start the design of the Twizzler network stack we must take this into consideration. As Twizzler becomes a distributed operating system, we envision that the network stack itself will assist in the management and movement of persistent data objects. We foresee the development of a new distributed network protocol which will run on both end-hosts and network devices, that will provide services for discovering the location of objects and will help manage the transfer of object data. The Twizzler network stack, which will run in user space, will be able to assist user programs as they attempt to access persistent data that is not local.

An important research question is what the role of the network should be in relation to concerns such as coherency, replication and consistency. These

requirements create the need for lightweight messaging, where the network protocols do not add unnecessary overhead to the network communication. It is also important to note that the Twizzler network stack is still in the early stages of development; and thus, there is a great possibility that Twizzler will eventually need to support many different types of applications, some of which we have not even thought of yet. To prepare for these possibilities, our goal is to create a flexible network stack that will support current applications as well as new ones, supporting the growth and expansion of Twizzler.

In order to carry out our vision we decided against implementing a traditional TCP/IP stack as our first version of the Twizzler network stack. Although we will eventually need a full TCP/IP stack for Twizzler to be compatible with other operating systems, our immediate focus is to create a stack for Twizzler that will support the needs of our envisioned user cases. For this reason, we have chosen to use the Flexible Interconnection Protocol (FLIP)[2] for our Twizzler network stack. FLIP allows for custom headers and will sit above the data-link layer. FLIP gives application developers control over which functions the network provides for their specific application. The flexibility of FLIP also allows us to meet different communication needs for future applications.

In this thesis, we describe the design processes that we used and the work that was done to build the first two layers of the Twizzler network stack using the FLIP protocol architecture. When considering how the network will assist in the movement and management of objects, it became evident that we would need a mechanism to

discover the location of these objects. The Address Resolution Protocol (ARP) is a protocol that provides a mechanism for the discovery of addresses in a network by associating them to an IP address. For this reason, we decided to implement ARP with FLIP; this allows us to easily modify ARP to perform resolution on object-IDs, allowing for an object's location to be discovered. Although we envision the network assisting in the resolution of object-IDs in the future, by implementing ARP as our first network application, we allow for early stages of testing to begin in the distribution of the object space.

When designing the Twizzler network stack, we made all of our decisions while keeping in mind the necessity to have adaptability and extensibility that could foster future growth. This first version of the Twizzler network stack can now support a subset of FLIP functionalities. It can also be easily expanded to support a full FLIP implementation and to also support a TCP/IP stack on top of the data-link layer. We hope to see our work expand into a complete and distributed network stack, as we revolutionize the way FLIP is used to create a new data-centric network stack.

Chapter 2

Background

2.1 Twizzler, the Data-Centric Operating System

Traditional operating systems were designed for a memory hierarchy with two regions; RAM, which is fast, volatile, and byte-addressable memory, and disk, which is slow, persistent and block-addressable. In contrast, NVM gives operating systems a main persistent memory that is low-latency and byte-addressable. NVM technology allows user processes to directly access persistent memory because of its uniqueness of being byte-addressable. In order to use NVM technology to its fullest potential, operating systems running on a machine that utilizes NVM technology should not interact with persistent storage as a traditional operating system would. This means that the operating system's persistent data path must be modified, so that access to persistent memory is fast and direct. The Twizzler operating system does just that, as it provides a modified persistent data path that is meant to utilize the advantages provided by NVM technology.

To truly understand why the persistent data path needs to be modified when using NVM, it is important to first understand how a traditional operating system manages persistent data. In a traditional operating system, any data that resides in persistent storage cannot be directly accessed or modified. Any user program that needs to write to or read from a file that is in persistent memory needs assistance from

the operating system. The file system (the set of functions in the operating system code that is responsible for moving data to and from persistent storage) is accountable for any read and write operations and for managing the data structures needed to perform these operations. These system calls to the file system tend to hog resources and hinder the performance of the CPU.

When a user process requests access to a file, the file system needs to first open the file, thus requiring its information to be loaded into memory before it can be read. To open a file, the file system must first copy the hard disk's directory, which will require another load operation. This operation allows the file system to find the directory entry of the file it is trying to access in the hard disk. The file directory is stored in a known location in the hard disk so that the file system knows where it is located and thus can be copied to memory and read from. The file directory contains information for each file stored in the hard disk, such as the file's name, location in the hard disk, ownership, access rights, and so on. Once the file system has access to the directory and the entry for the specific file it needs to open, it must copy that entry into the System Wide Open File Table (SWOFT), which holds a copy of the directory entry for each file that is currently open by any user process. The SWOFT, which resides in memory, allows the file system to easily remember where the file is stored in the hard disk. This allows the file system to skip the step of referencing the directory during any consecutive read and write operations to a file that is open. The

file system also needs to create a per-process Open File Table (OFT), so that the user process can keep track of the logical byte number of any file it is currently trying to access.

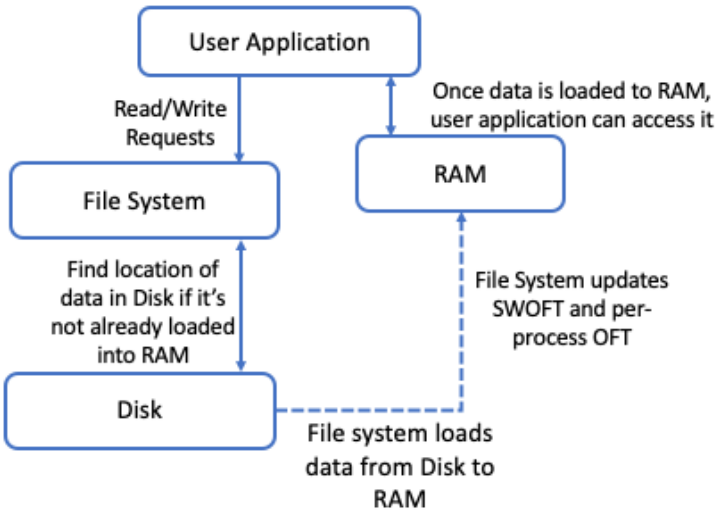

Figure 2: Reading and Writing to Persistent Data - Traditional OS

Once a file has been opened, bytes are read from it using the read system call. It must be noted that any of the file's bytes which a process needs to read must reside in memory during read or write operations to that file. The file system only loads an entire file into memory if the file is smaller than or equal to the size of a file block. Otherwise, the file system will load blocks of the file instead to conserve memory and allow multiple files to be open at the same time. Since a block holds several bytes, a system call is not necessary for each byte that needs to be read. As illustrated in Figure 1, anytime a read or write needs to be performed to a byte that is not part of a block which has been previously loaded into memory, the file system must first load that new block from the hard disk. The file system is also responsible for "page swapping," a process that swaps blocks out of memory that are not currently being

used. This makes room for new blocks that may be needed if there is no more space in memory. Page swapping is a very expensive algorithm that hogs the CPU from user processes. After the file is no longer needed by any user processes, the file system needs to update all the data structures that were needed to keep track of these files. It is also the file system's responsibility to update the hard disk directory in case a new file was created or if an existing file was written to.

It is clear that the kernel is incredibly involved in the persistent data path of a traditional operating system, as it greatly affects pathways needed to access a file. The file system must keep the hard disk directory up to date, it must manage both the SWOFT and per-process OFT, and it must swap pages in and out of memory as they are needed. This process is a necessary one given that user processes cannot directly access the hard disk. NVM, in contrast, overrides this need by being a byte-addressable persistent memory that allows user programs to have direct access to it. This diminishes the need for heavy kernel involvement in the persistent data path, which is preferable because it allows user processes to directly access persistent memory and increases CPU utilization. The Twizzler operating system enables this improvement by providing user programs with memory-style access to persistent data by rethinking the system stack. Twizzler exposes a data-centric programming model, where pointers to persistent data live as long as the data does. It also provides in-memory data structures that are used by programs to manipulate NVM directly. Load and store system call operations in Twizzler are mediated via a user-space OS library, so the need for the kernel to get involved in persistent operations is minimized.

Twizzler breaks down NVM into pages, much like memory in a traditional operating system, and defines persistent objects, giving each object a unique object-ID that is used as the unit of control to access NVM. Each object can consist of 4KiB (one page) to 1GB, and any larger data structures operate similar to a linked-list as they can span across multiple objects with references between them. The kernel's responsibility with persistent memory in the NVM system is simply to manage these data objects by mapping, creating, and deleting them. We no longer need kernel involvement to enable programs to read from and write to these persistent objects because once a program has a mapping from a virtual address to an object-ID, it can access NVM much like a program in a traditional OS accesses memory.

A persistent object consists of two parts: an FOT-index and the offset, which is shown in Figure 2. The FOT-index refers to an entry in the Foreign Object Table (FOT). The FOT, which resides in a known location of each object, holds references to foreign objects. Together, the persistent pointer and FOT make up an object-ID. If the FOT index is zero, the pointer is referencing data within its object, but if the FOT index has a different value, we then look up that value as an entry in the FOT. As shown in Figure 2, using this reference allows us to easily get data from foreign objects.

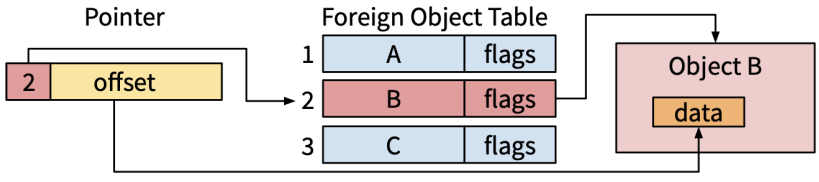

Figure 2: Twizzler pointer format [1]

This design makes Twizzler very unique and allows for very large offsets. Since the pointer does not hold the reference to a foreign object, but simply a reference to an entry in the FOT, we can have very large object-IDs. This design enables a majority of the bits in a pointer to hold the offset value within the object. Having a very large object-ID space allows us to create machine-independent data references, where these cross-object pointers reference objects in different physical locations.

As discussed, Twizzler is able to provide fast access to persistent memory without heavy kernel involvement. Twizzler is specifically designed to maximize usage of NVM's advantages to their fullest extent, which includes permitting user programs to have direct access to low-latency persistent data. The natural consequence of this is to research and investigate how we can make the best use of the Twizzler design of object-IDs to create a true global object space that spans across multiple physical machines.

2.2 FLIP: Flexible Interconnection Protocol

When the traditional TCP/IP stack was developed in the 1970s, networks consisted of interconnected machines whose processing power, storage, and communication capabilities were close to identical. But as the Internet grows and we envision the Internet of Things (IoT), we are moving towards a world where devices connected to the Internet range from very simple devices such as sensors, to devices that are extremely complex and powerful such as servers in a data center. The protocols in the TCP/IP stack were developed to ensure reliable data delivery over

extremely unreliable data paths, where end-hosts implemented most of the algorithms necessary to ensure that packets sent out the wire were received on the other end. The TCP/IP stack was developed with the End-to-End Argument[3] being held as a holy-grail. This argument stated that the underlying network should provide the minimum functionalities necessary to support an application, which added a lot of strain on end-hosts and required expensive resources to ensure reliable delivery.

Although the TCP/IP stack was a great solution for interconnecting computers with similar technology, we need better solutions as end-hosts evolve. For example, sensors often have limited power and storage capabilities and cannot handle a full TCP/IP communication. In fact, many of the TCP features, such as reliability, end-to-end connection setup, or even flow control, are not necessary in many new network topologies. In contrast, computers running on a server farm require reliable and fast communication that doesn't have too much overhead. Recently developed network infrastructure is much more reliable than it was in the 1970s, and much of the connection setup isn't necessary as the physical layer has become more reliable. With such reliable communication paths, much of the overhead previously included in traditional network protocols is no longer needed, and thus the assumptions made about the communication path needs to adapt to this change.

The network stack has evolved over the years with protocols such as UDP that allow for unreliable communication over the network medium. Although UDP is a great protocol for applications such as video conferencing, it stands at the complete opposite of TCP, giving application developers no features to ensure reliability or

even correctness. As the network stack continues to evolve, many of the protocols being developed are not flexible enough to give application developers the flexibility needed to use a stack that best fits their specific applications.

The Flexible Interconnection Protocol (FLIP) was developed at the University of California Santa Cruz to address needs of new networked environments that interconnect heterogeneous devices ranging from very simple sensors (e.g., temperature, humidity, etc.) to traditional computing devices such as laptops, desktops and servers. FLIP, which is meant to sit on top of the data-link layer, allows application developers to use custom headers for their applications; therefore, developers only use the functionalities that are needed up to the application layer. Due to the flexibility provided by its custom headers, FLIP can either be a very thin layer which provides minimum functionality, or it can be a heavy-duty protocol providing the functionality of TCP. Yet, what makes FLIP so unique is that the application developer has *complete* control over how much functionality is implemented on the network, removing the overhead that comes with unnecessary functionality. Not only is FLIP flexible, but its design allows new protocol functions to be easily added to an existing network, which promotes future growth and development for unforeseen application needs.

As Twizzler developers we have recently started to explore the possibilities of moving Twizzler to a networked environment. One of the goals envisioned for the Twizzler operating system is to provide a seamless transfer of data between machines by taking advantage of the global object space provided by NVM. As we explore the

different possibilities of the applications that will run on Twizzler, and which functions Twizzler will need from the network stack, it is apparent that FLIP's flexibility is a great option to serve not only the immediate, but also the future needs of the Twizzler stack. By using FLIP as the basis for Twizzler network stack we are providing the Twizzler developers with all the flexibility they need to explore new applications in a new data-centric operating system.

Chapter 3

Related Work

In order to keep up with new and emerging technologies in recent years, systems research has focused on improving both traditional operating systems and the network stack. In this section, we will discuss how the Twizzler operating system design is the product of research in operating systems and novel work in NVM support. We will also discuss various network stack designs and protocols that, much like FLIP, have been proposed to address the unique needs of emerging technologies which were not yet created when the TCP/IP was developed.

3.1 Microkernels and NVM Support

The design of minimalistic microkernels, such as Mach[11], greatly influenced the system design of Twizzler. With microkernel designs, the operating system only provides the necessary resource protection; this allows for a large part of the operating system's implementation to be placed in user space. Because kernel code runs as a single process and is not preemptable, the system's performance can be greatly improved by moving operating system functionality to user space. This design model influenced the way the Twizzler system stack was designed, so as to create a minimalist kernel. As we will discuss in Chapter 4, this design model also led to the decision of placing the Twizzler network stack into user space.

There have also been new storage system designs that provide operating system support for NVM. In the Moneta Project[12], a new hardware/software interface was designed to support applications running on a single-level store such as NVM. This approach aims to greatly improve performance for memory access by reducing the operating system overhead as much as possible. When redesigning the system stack, we leveraged these ideas to create a new-persistent data path into the Twizzler design.

3.2 Network Stacks Designs

One user case example where we've seen new research on network stacks is that of time-sensitive applications. In *Time-Sensitive Networking for Robotics*[4], the authors reveal a compelling trend; in order to achieve higher determinism, where packets are sent out as best-effort, real-time applications have substituted the TCP/IP stack with custom stacks. By examining protocols such as EtherCAT[5] and SERCOS III[6], we see that protocols have become increasingly specialized to meet specific requirements or tasks needed by sensors and actuators. However, efforts to achieve higher determinism have made protocols become too domain specific, which has resulted in their inability to meet all application needs of a network. This creates the need for overly-complicated network stacks that implement a variety of protocols. We must be aware of this trend as we design the Twizzler network stack so that we are not limiting the network's support for new applications that may be developed to run on the Twizzler operating system.

It is also important for us to consider work with user space network stack designs. *Fast userspace packet processing*[7] analyzes and compares different techniques for user space packet processing. This analysis demonstrated that kernel bypassing techniques cause a performance drawback because they hinder the line speed of high-speed links. However, there are solutions to address this performance drawback, such as Netmap, which delivers raw packet buffers straight to the user space. This technique is something that we must consider as we design our network stack, so as to achieve the best performance possible.

3.3 Protocols with non-fixed headers

In attempts to reduce network overhead, some researchers have experimented with various compression techniques, allowing protocols to define header compression. The Unified Header Compression Framework [8] is one such technique; unlike FLIP, however, these header compression techniques require a persistent state between the sender and receiver. Another drawback that comes with these compression techniques is that, due to end-hosts needing to keep state, they are only appropriate for point-to-point links. FLIP takes away the need for a persistent state with its meta-headers, creating a simpler protocol that can be interpreted by network devices.

Chapter 4

Twizzler Network Stack Design

The Twizzler design model involves the creation of a minimalist microkernel where the complexity of the system is implemented in user space; this simplifies the design of the kernel itself. As we will show, we implement as much of the network stack as possible in user space to keep the Twizzler kernel simple and minimalistic. The network stack's current implementation was specifically developed to meet the immediate needs of Twizzler. Future extensions to the Twizzler network stack and its protocols will be discussed in Chapter 7.

There were two key factors that drove our decisions while designing the Twizzler network stack: first, support for a distributed object-ID space that spans across multiple machines; second, designing the network stack to be flexible enough to fulfill the needs of future applications. We envisioned a user-level network stack program that would be responsible for moving objects around the network by sending requests for foreign objects and responding to requests coming in from the network for local objects. The protocols that would enable this type of object movement could not only run on the Twizzler end-hosts but could also hand off some of their functionalities to the network by using programmable switches. Because of these features, user programs no longer need to send explicit network request for data that

is not local; in fact, they do not even need to be aware that the target object was not local.

As we began designing the Twizzler network stack, our first focus was to get Twizzler to a point where it could simply push bits out the wire. Or next goal was to add some structure to this data we were pushing out the wire so that it could be interpreted by the network. As previously discussed, this led to the decision of implementing Ethernet as our data-link layer protocol and FLIP as our Layer 3 protocol, which gave us a great starting point for our stack design. It is important to note that the current design is meant for a Local Area Network (LAN) environment and that more functionality will need to be added so that we can support Wide Area Networks (WAN).

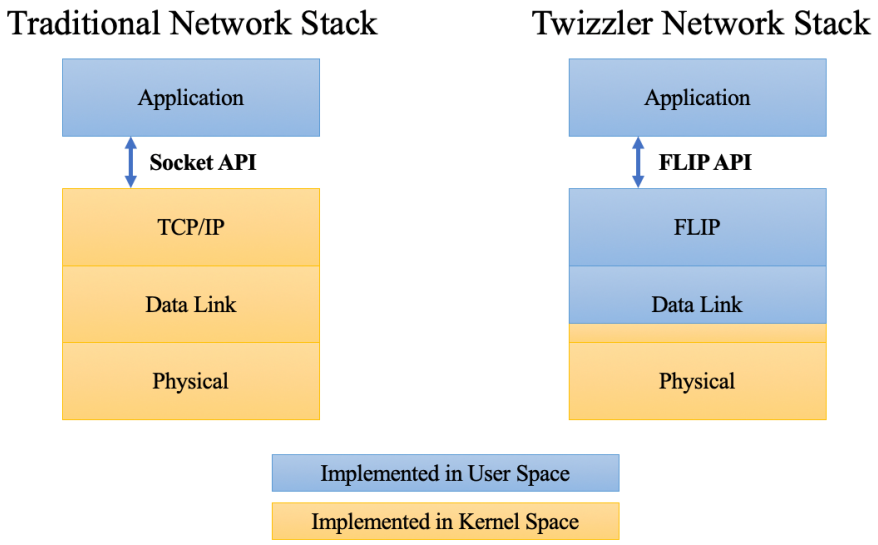

Figure 3: Traditional Network Stack vs Twizzler Network Stack

Figure 3 shows a side by side comparison between a traditional network stack and the Twizzler network stack. The Twizzler network stack has four layers: physical,

data-link, FLIP, and Application. Unlike a traditional network stack, the Twizzler stack implements all functionality in user space down to the framing functionality of the data-link layer. The user space program that implements networking will encapsulate the packet with the source and destination L2 addresses and the protocol type and place the packet in the NIC buffer. The NIC will then encapsulate the packet with the Preamble, Start of Frame Delimiter (SFD), and the Cyclic Redundancy Check (CRC) header fields. With a traditional network stack, the application layer communicates with the Socket API to send and receive network data, while the Twizzler stack proves the user application with a FLIP custom API for this same purpose. The FLIP layer has not yet been fully implemented, but current functionality meets the immediate needs of Twizzler and is described in detail in Chapter 5 below.

4.1 Data-Link Layer

As shown in Figure 4, the data-link layer follows the Ethernet IEEE 802.3 standard. The Preamble, SFD, and the CRC are implemented by the NIC itself. The destination and source addresses, along with the length and payload, are handled by the network stack user program.

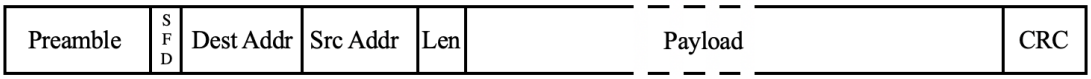

Figure 4: Ethernet packet

The source and destination addresses are six-byte fields used to represent a 48-bit MAC address. The source MAC address of a NIC card is exposed by the NIC to the Ethernet layer through an API, and the destination MAC address is learned

dynamically through the ARP protocol. The FLIP layer is responsible for providing the destination MAC address to the Ethernet layer. We decided to have the FLIP layer specify the destination MAC address, while the Ethernet layer encapsulates the packet with the correct destination MAC address because the FLIP layer knows which type of L3 destination address we are using. If we are using an IP address, FLIP can use the ARP protocol to perform address resolution, but if we use a different L3 address, we cannot use ARP to find the destination MAC. For this reason, and because L2 is not aware of the L3 address being used, we gave that responsibility to the FLIP layer.

4.2 FLIP Layer

The original developers of the FLIP protocol split the design into two layers: the first was the FLIP layer, which implemented Layer 3 functionality; the second was the Generic Transport Protocol (GTP), which implemented Layer 4 functionality. GTP has the same design as FLIP; for this reason, FLIP can be easily expanded to include all the features of GTP. Our immediate focus was to design and implement FLIP's Layer 3 functionality; this design allows for growth and expansion to support additional functions in the future.

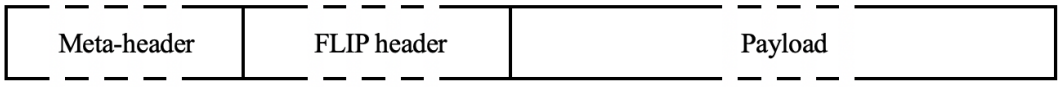

Figure 5: FLIP Packet

As shown in Figure 5, the FLIP protocol has two separate headers. The meta-header is a bitmap that tells us which fields are present and being used for encapsulation in the header. Figure 6 shows how the bitmap is split into three separate

bytes. The application programmer must provide the FLIP API with the meta-headers for their application, thus specifying which communication functions are needed. The FLIP protocol will then take these meta-headers and encapsulate the data accordingly.

The continuation bit is the first bit of each byte of the meta-header, and it tells us whether there is another byte of the meta-header following the current byte. The flexibility in the meta-header design helps reduce unwanted overhead while also allowing for more functionality to be added as needed. Although we have kept the original order of the meta-header fields, which was chosen by FLIP designers for simple applications and devices, we have the ability to move them around in the future as we see fit.

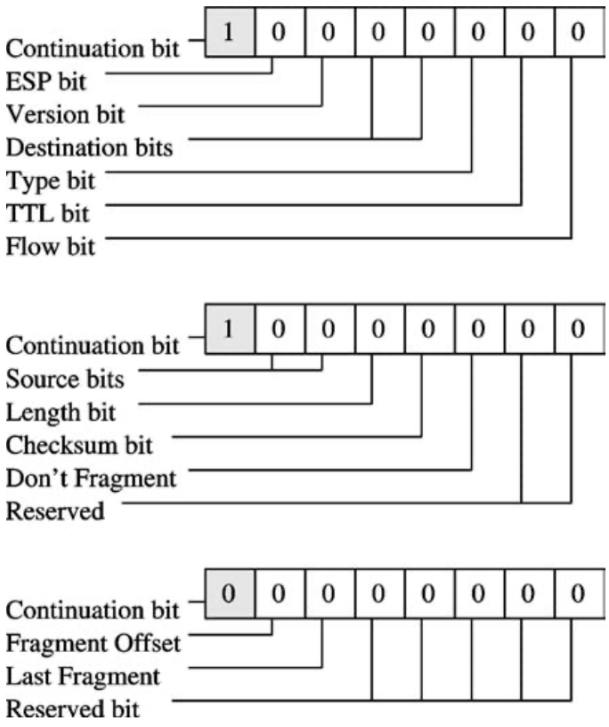

Figure 6: FLIP meta-header structure [2]

Currently, we don't support all fields of the FLIP packet; however, we have implemented the necessary features that will support the implementation of the ARP protocol. The definition of the FLIP packet fields are as follows:

- Version: this is a one byte field, where the higher four bits represent the version and the lower four bits represent the packet priority. The current FLIP verison is 0, and the priority is assumed to be zero for all packets.
- Destination: this two bit field represents the destination address, where 00 means that the destination field is not present, 01 represents a two-byte address, 10 represents a four-byte address, and 11 indicates a 16-byte address. Although the original FLIP design indicated a 11 as a variable length address, we chose to use 11 to represent a 16-byte address; this allows us to use an object-ID as an L3 destination address.
- Type: the type field has also been modified from the original FLIP design, which was a one-byte field to indicate the protocol type. We increased this field to be a two-byte field in the FLIP header. Currently, there isn't much distinction between the L2 type field in the Ethernet header and the type field in the FLIP header; however, we expect this to change as we create more applications and protocols to run on the FLIP stack.
- Time to Live (TTL): this is a one byte field meant to control how long a packet can travel the network. This feature is not yet supported by our FLIP implementation, but it can easily be enabled when and if the need arises.
- Flow: the flow field is a four-byte header that will be used for flow control.

- Source: the source address is implemented using the same protocols as the destination address, with 01 representing a two-byte address, 10 representing a four-byte address, and 11 indicating a 16-byte address
- Length: this two-byte field represents the length of the payload encapsulated in the FLIP packet.
- Checksum: this two-byte field will be used in the future for error checking of the packet payload.
- Don't fragment, fragment offset, and last fragment: the field of fragmentation is not yet supported by the Twizzler network stack.
- Reserved: the field of reserved bits are not yet defined and are one mechanism which will support future growth. We used reserved bits on the second byte of the meta-header for the ARP protocol implementation, but this usage is only relevant for an ARP packet type.

As shown in Figure 6, the second field of the meta-header's first byte corresponds to an Extra Simple Packet (ESP). This special packet was defined by FLIP designers as a short packet that only carries data. If the ESP field is set on the first byte without a continuation bit, then the responsibility of holding the packet's payload falls upon the rest of the bits in the meta-header. In contrast, if the continuation bit and the ESP bit are set, then the next 14 bits represent the payload. While it's beneficial to have a simple ESP design, it does not provide us with any L3 functionality. We decided to keep the first bit of the meta-header solely for ESP use but have not fully implemented encapsulation and decapsulation of ESP packets. We also left the bit

there to allow future developers, who may want to send ESP packets into a Twizzler network, to test the network stack.

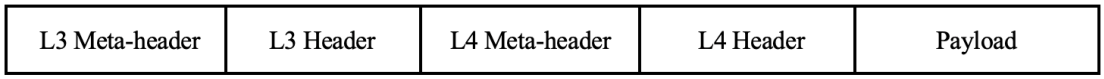

Figure 7: FLIP design, splitting L3 and L4 meta-headers

Figure 8: FLIP design, combining L3 and L4 meta-headers

The current design of the FLIP protocol can also be modified to include transport layer functionality. Figure 7 shows that we can add a second set of meta-headers, following the FLIP headers, to represent the transport layer. Another possibility is to simply expand the fields of the FLIP meta-headers, as shown in Figure 8. The first design option follows a more traditional network stack, where there is a clear handoff from the network layer functionality to the transport layer functionality. Using the design of Figure 7, allows for both the network and transport layers to have headers that can vary in length.

In contrast, the second design option is easier to program because of FLIP's flexible rather than fixed headers. Having all meta-headers available in the first layer of encapsulation makes it easier to calculate the packet length and allocate enough buffer space for all headers. However, there are certain challenges with this design, like not knowing which bits of the meta-headers correspond to L3 header fields and which ones correspond to L4 header fields. A viable solution to this challenge is to have a reserved bit on each byte of the L3 meta-headers, much like the continuation bit; this could tell us the subsequent byte is a L4 meta-header. Both designs are viable

options, but we ultimately aim to choose the design based on two factors: if we need a distinct separation of L3 and L4, or if we need the simplicity of having a single FLIP layer that provides both functionalities. Since Twizzler will eventually have two network stacks (a traditional TCP/IP stack, and a data-centric stack using FLIP) we may choose to have one FLIP layer that provides all the needed functionality for our data-centric applications.

4.3 ARP Protocol Implementation using FLIP

As previously discussed, by adding ARP as our first network application we provide the FLIP stack with the ability to perform address resolution. By implementing ARP with FLIP, we can easily modify ARP to perform resolution on object-IDs, allowing for an object's location to be discovered and therefore encouraging the early stages of testing to begin in the distribution of the object space. We can leverage our existing FLIP design to easily implement ARP without having to define separate ARP headers. Because of FLIP's unique ability to allow developers to control meta-headers, we are able to send FLIP packets to perform full address resolution without needing to send any payload. Figure 9 shows how the FLIP meta-headers have been hard-coded for ARP request and reply messages, with only the necessary fields being included for address resolution.

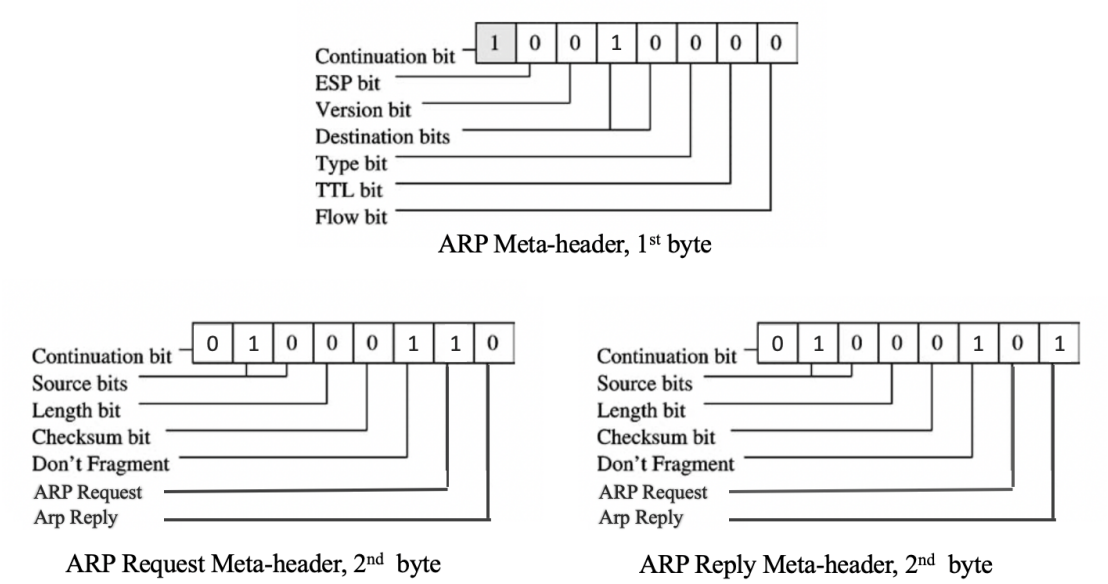

Figure 9: ARP meta-headers. Image adapted. [2]

ARP messages have a type field of 0x0806 for both the Ethernet and the FLIP layers. When packets are received with this type field, they are processed by the ARP protocol and the last two bits of the meta-header, which were previously reserved bits, are now used to decipher between an ARP reply and an ARP request message. We will discuss the details of the ARP protocol implementation and the interactions between the different layers of the Twizzler network stack with the ARP protocol in Chapter 5.

Chapter 5

Implementation

The network stack for the Twizzler operating system was designed using packet buffers, thus making the hand off from one layer of the network stack to the next an easy and seamless process. When implementing the system stack, we started with the Ethernet layer and then moved onto the FLIP layer. For simplicity, we tie these implementations together and examine a packet's ingress and egress journey through the Twizzler network stack.

5.1 Initialization of Network Objects

There are three Twizzler persistent objects that need to be initialized before any packets can be sent or received: Interface, Tx-queue, and Rx-queue. The Interface object holds a data structure that is used to keep track of the various interfaces' network properties, such as its MAC address and its IP address. The Interface data structure is populated through an API that initializes the MAC and IP address fields. The MAC address is exposed to the API by the NIC, and an IP address is provided as a program argument when the network program is initialized.

The Rx and Tx-queue objects are used by the NIC to process packets. The Ethernet layer will place packets into the Tx-queue and remove packets from the Rx-queue. In our current implementation, an important feature that will need to be changed is how the reference to the Rx-queue object is passed between layers and

down the stack. This feature is desirable for testing purposes because we have not restricted which layer it can place a packet into the Rx-buffer, which means any layer of the stack can place data into the network for experimental purposes. For security reasons, this will need to change as we get the network stack ready for use by application developers.

5.2 Packet Encapsulation

Once the persistent objects have been initialized, the network stack is ready to send and receive data. The FLIP layer, which is exposed to applications by a Send-FLIP API, must be provided with the following:

- FLIP meta-header bitmaps with the functionally needed for this communication. Note that the Send-FLIP API must be provided at least the first byte of the bitmap and that the second and third bytes are optional fields.
- A reference to the interface-object holding the network properties of the physical interface that will be used to push data out the wire. The FLIP stack currently supports a single NIC, but it can be improved upon by implementing routing and incrementing the number of interfaces a single machine can have. Instead of having the application layer specify the interface that will be used for communication, we can automate this process by having the FLIP layer choose the correct interface based on the subnet an interface is assigned to and its destination address.

- Destination address. This is an optional field for applications that require this functionality, which will be indicated by the meta-header bitmap.
- The protocol type. This field is not optional and is needed by our implementation. Since we are using the same protocol type for L2 and L3, the Ethernet layer will not know how to decapsulate a packet at the receiving end if the type is not provided by the sender. The FLIP protocol type that we have chosen is 0x2020.
- If a broadcast message is being sent, a boolean flag must be passed to the FLIP layer to indicate the encapsulation of a broadcast packet.
- If a payload is to be encapsulated, the application must pass that data to the FLIP layer in the form of a character pointer.

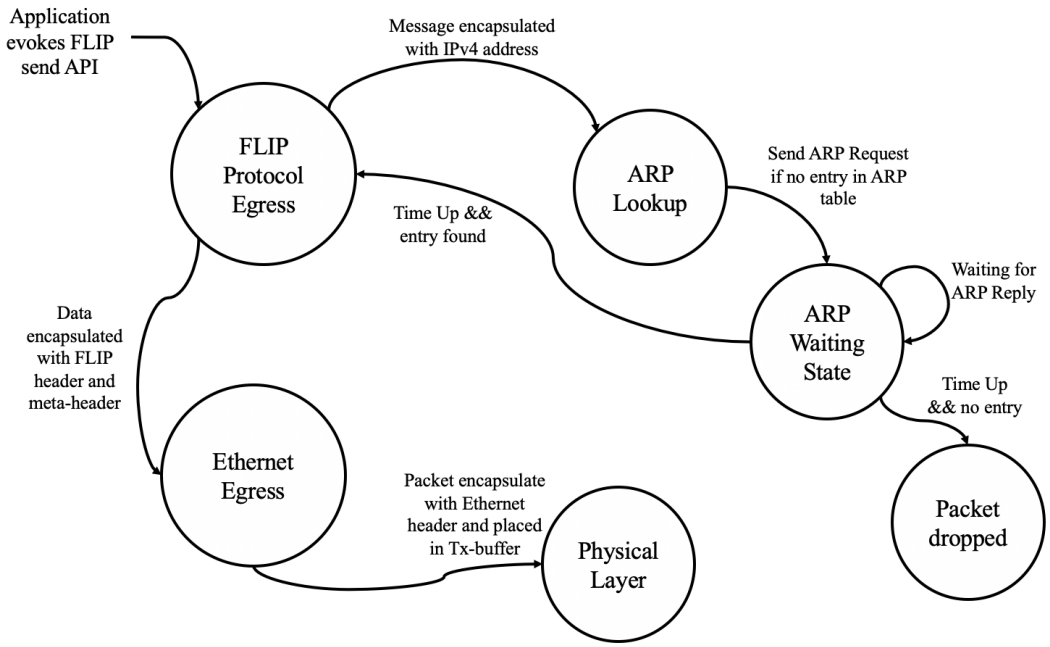

Figure 10: A packet's egress journey

The FLIP layer will take the provided meta-headers and calculate the size of the FLIP header based on the functionality set by the meta-header bitmap. Once the size of the FLIP header is calculated, the size of the complete packet is then allocated as the packet buffer. The packet buffer will be big enough to fit the payload, the FLIP header and meta-header, and the Ethernet header. Once the packet buffer is created, the payload is placed into the right-most end, leaving empty space to the left of the buffer for encapsulation by all layers.

The FLIP layer then creates a pointer that will reference the byte in the packet buffer where the first meta-header should be placed. There will be just enough space to the left of this pointer to place the Ethernet headers. Next, the FLIP layer can encapsulate the payload by placing the meta-headers and necessary header fields at the starting pointer position. As shown in Figure 10, if the FLIP header has a set four-byte destination address, it is assumed that this is an IPv4 address. Unless this is a broadcast packet, address resolution must be done to find the destination MAC address based on the destination IPv4 address provided. Before the FLIP layer can pass a pointer to the packet buffer and down to the Ethernet layer, it must successfully resolve the MAC address. If the MAC address cannot be resolved, this packet cannot be processed any further and will be dropped.

The ARP protocol is invoked when an IPv4 destination address is a field in the FLIP header; the first step in this protocol is for the ARP-table to be checked to see if the MAC address associated with the provided IP address is known. The ARP-table is simply a volatile hash table that maps IP addresses to MAC addresses. If the

MAC address is unknown, the ARP protocol will send out an ARP request message using the FLIP protocol and the message will be set for the 0x0806 ARP type with the meta-headers for an ARP-request message. Since this message will be a broadcast message, FLIP will not attempt any address resolution with the destination IPv4 address; this prevents the FLIP protocol from being stuck in a loop state where it continuously attempts to resolve the IP address. The ARP protocol will then wait for two seconds before doing another lookup on its ARP table (note that the time the ARP waits for resolution will need to change in the future to reflect a network's RTT). If the address cannot be resolved, the packet will be dropped by the FLIP layer and an error message will be displayed; but, if ARP resolution is successful, the packet will continue its egress journey.

Once the FLIP layer is done encapsulating the packet, it is ready for the Ethernet layer to add its headers. With packet-buffers, a pointer reference is passed between layers to where the next layer's headers should be placed. In the case of the FLIP layer, it must give the Ethernet layer a pointer to the beginning of the packet buffer where the Ethernet header will be placed. FLIP must ensure that its headers and meta-headers are placed to the right side of the packet buffer, preceding the payload.

In order for the Ethernet layer to encapsulate the packet, it must be provided with the protocol type, the destination MAC address, and the length of the packet buffer, which must be passed to the NIC. The Ethernet layer will place its header at the location of the pointer provided by the FLIP layer. It will then use the interface's

object to populate the header with the source MAC address and information provided by the FLIP layer to populate the destination MAC address and packet type. A packet queue entry data structure is then created. This data structure must be provided with the pointer that references the packet buffer, along with its length; this data structure is then placed in the Tx-buffer to be processed by the NIC.

5.3 Packet Decapsulation

When a packet is received by the NIC, it is encapsulated with a NIC specific header data structure, which contains control information added by the NIC itself, and is then followed by the actual Ethernet header. The Ethernet layer moves the pointer from the beginning of the NIC's header to the beginning of the Ethernet packet. Next, it creates another pointer that it will pass to the upper layers, which references the beginning of the Ethernet payload. Then, the Ethernet layer checks protocol type in the Ethernet header: if it's a FLIP type (0x2020), the payload pointer is passed to the FLIP layer; if it's an ARP type (0x0806), the packet will be processed by the ARP protocol and received by the ARP API. The Ethernet layer must also provide the upper layers with a reference to the object of the receiving interface and the source MAC address.

If the message received is an ARP message, it will be processed by the ARP protocol. First, the source IP address must be added to the ARP table. If the ARP table already has an entry for that IP address, it will be overwritten to ensure the latest MAC-to-IP mapping. If this message was an ARP reply message, the ARP protocol has completed its handshake and no further processing is needed. In contrast, if it was

an ARP request message, then an ARP reply message would be sent using the FLIP layer and include the corresponding meta-headers for an ARP Reply message. The ARP reply message is a unicast message, where the source MAC and IP addresses of the incoming message now become the destination MAC and IP address of the outgoing message.

If the received message is a FLIP type, the header is decapsulated based on the active fields of the meta-header by the FLIP layer. If the header has an IPv4 source address, an entry is added to the ARP table using the source MAC provided by the Ethernet layer and the source IPv4 address in the FLIP header. At this point, we have reached the end of the network stack and the payload of the FLIP packet is displayed to the user. As we develop applications that will use the network stack, we can simply pass a pointer that references the beginning of the payload to the correct application. To be able to support multiple applications receiving data from the FLIP layer, we will need to implement port numbers as part of the FLIP protocol.

Chapter 6

Testing

As each layer of the Twizzler network stack was implemented, we ran manual tests to ensure correctness of our protocol. In this section, we will go through the tests executed through each stage of the network stack.

Test 1: Ethernet Header Implementation Test

The first test that was done to verify the network stack's successful implementation was to execute encapsulation and decapsulation of Ethernet packets. The test was run on a non-Twizzler machine using standard sockets as messages were passed between processes. Although the code for the Ethernet layer needed to be modified when we transferred it to a Twizzler machine, running the test in this way allowed us to assess our understanding of the Ethernet layer implementation and ensured that we were able to correctly encapsulate and decapsulate Ethernet data packets.

Test 2: Producer/Consumer Buffers

The producer/consumer buffers test was created to verify the Ethernet code in the Twizzler OS. It involved transferring the Ethernet code previously implemented in a non-Twizzler OS to the Twizzler OS. The producer/consumer buffer was also created as a destination where Ethernet packets could be placed, to ensure that two processes could communicate and interpret Ethernet packets. Encapsulated messages

were put into the buffer and then the consumer removed the messages and decapsulated them. We ensured the payload from the sender process was the same received by the receiver process and that it contained hard-coded fields of the Ethernet header.

Test 3: Testing NIC Tx-buffer

To test the communication between the Ethernet layer and the NIC, the Ethernet layer was modified to place packets into the Tx-queue. Packet captures were taken using Wireshark to ensure that the NIC was able to push the packets in the Tx-queue out the wire. This test was also helpful in ensuring that the Ethernet packets met the IEEE standards by ensuring that a third-party software could also interpret it.

Test 4: Testing NIC Rx-buffer

To test the NIC's Rx-buffer and its communication to the Ethernet layer, we connected two Twizzler machines together in order to send Ethernet traffic back and forth. Figure 11 shows how we connected the two Twizzler VMs together through an Open vSwitch using virtual bridge NICs; a suggested configuration from *Software-Defined Networking (SDN) with OpenStack*[9]. This topology setup allowed us to not only send traffic between the Twizzler VMs, but to also ensure correctness by capturing the traffic with Wireshark. This ensured that we could connect Twizzler to a virtual switch, which will be a great advantage as we run virtual tests with more complex topologies using network emulators such as Mininet.

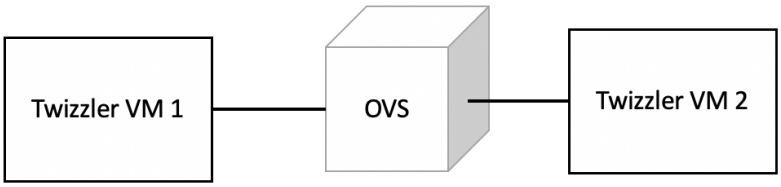

Figure 11: Twizzler test topology

Test 5: Testing FLIP

Once the FLIP layer was implemented, we tested the communication between the Ethernet and FLIP layers both up the stack and down the stack by sending FLIP messages between the two VMs. Wireshark could not interpret FLIP headers since doing so is not a standardized protocol. This requires us to manually check the packets bit-by-bit to ensure that the FLIP headers and meta-headers were correctly implemented.

Test 6: ARP Learning

If an incoming FLIP message contained an IPv4 source address, we tested to verify that MAC addresses were being learned and that the ARP table was being populated. Given that address resolution was not yet implemented, we hard-coded the destination MAC addresses to allow us to execute this test.

Test 7: ARP exchange

In the last implementation and testing stage, we tested our implementation of the ARP protocol to ensure it would send ARP request messages and reply to received ARP requests. At this point, we were no longer providing the network stack with the destination MAC address for unicast messages, but instead letting the ARP protocol find it for us. We captured this message exchange on Wireshark and

confirmed an ARP hand-shake was executed before a message was sent. We then manually checked these packets' MAC and IP addresses to verify that the ARP exchange was correctly implemented.

Chapter 7

Future Work

The Twizzler network stack has evolved significantly since its early days of not having the ability to push bits out the wire, and we hope our work will propel this technology forward towards creating a full-fledged network stack. There are many different directions we can focus on to continue in the expansion of the network stack. We may choose to explore how we want to push the data bus into the network stack, to move Twizzler closer to our vision of a global object space. We must also keep improving the network stack and the functionalities it provides so that we can support a variety of Twizzler's application needs.

One area that still needs to be improved upon is the implementation of the FLIP layer itself. Many of the functionalities we discussed in Chapter 4 are not yet supported and will need to be implemented in the near future. However, we should prioritize the functions we chose to implement, following the procedure we used for implementing the functions needed to support ARP protocol. We must also consider the implementation of Layer 4 functionality to the FLIP protocol. Figure 11 shows an example of the Layer 4 meta-header used by the FLIP developers when they implemented GTP. When implementing Layer 4 using meta-headers, we can benefit in the same way as we did when implementing the Layer 3 functionality. As previously discussed, we must decide whether we want a single layer implementation,

which will provide all the functionality needed between the Data-link layer and the Application layer, or if we want to create a distinction between the two layers, which would resemble a more traditional network stack.

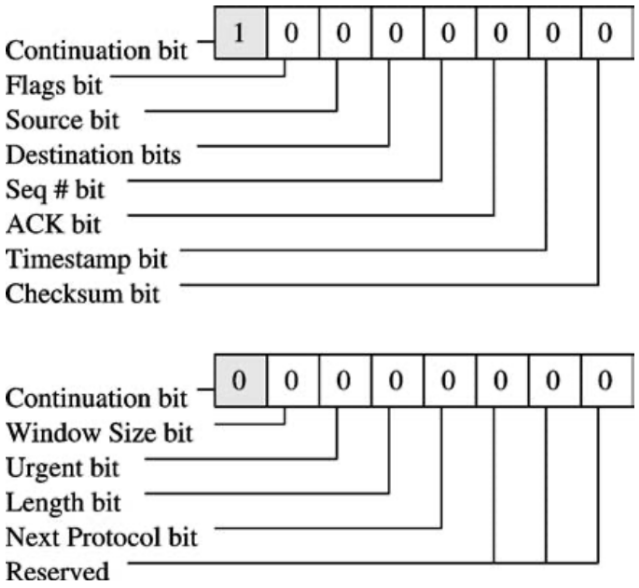

Figure 11: GTP meta-header [2]

By implementing ARP we are now able to send requests for MAC addresses, and we can expand this protocol to send requests for object-IDs. The Twizzler operating system throws a fault when a user attempts to access an object that is not local. The network stack helps by catching this fault and sending out a modified ARP request, where the resolution is for an object-ID rather than for a MAC address. By connecting Twizzler machines to programmable switches, our switches can learn the location of object-IDs and cache them for future requests; this minimizes the time a user program needs to wait to access a piece of persistent data that is not locally available.

Since we are able to discover objects that exist in different locations within the network, we can now consider options for dealing with these persistent objects. There are many research questions that should be answered as we move forward:

- Do we want to move an object to the local machine when it is discovered, or do we make a copy of it?
- How much of the object do we move: a page, or the entire object?
- If parts of our object are in different locations, how do we address these multiple copies and should an object get a new object-ID as it relocates around the network?
- What about keeping replicas of objects consistent?
- How much of the network's functionality will be used to assist in the implementation of a consensus protocol?
- Do we want to implement a protocol such as NOPaxos [10] where we decapsulate the responsibilities of the Paxos protocol by relying on the network for ordering?

These are just some of the research questions that need investigating as we continue to expand our new and innovative data-centric network stack.

Chapter 8

Conclusion

This report presents the design and steps taken to implement the first version of Twizzler's network stack. With this network stack, we were able to use FLIP in a new and unforeseen use case then intended by the original FLIP designers. This implementation has provided Twizzler developers a flexible network stack, where custom headers can be used to support specific application needs. We also implemented the first network application using FLIP and the ARP protocol. This is able to resolve IPv4 addresses but can be easily modified to resolve object-IDs. By implementing ARP, we have provided developers with the necessary tools to start working on the research of distributing the object space.

We laid the groundwork that is necessary to expand the Twizzler network stacks into multiple directions to address various research questions. The FLIP layer can be expanded to provide new network functionality as the need arises and it allows for new applications to be developed to utilize the network stack. We are at a point where we can start expanding the data-bus into the network. We hope that with this groundwork laid out, developers are encouraged to expand this work as we continue to revolutionize our data-centric network stack.

Bibliography

[1] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, & Ethan L. Miller (2020). Twizzler: a Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (pp. 65–80). USENIX Association.

[2] Solis, I., Obraczka, K. FLIP: A Flexible Interconnection Protocol for Heterogeneous Internetworking. Mobile Networks and Applications 9, 347–361 (2004).

[3] J. H. Saltzer, D. P. Reeed, D. D. Clark, "End-to-End Arguments in System Design," 2nd International Conference on Distributed Computing Systems, Paris, (April 1981), pp. 509-512.

[4] Gutiérrez, Carlos & Juan, Lander & Ugarte, Irati & Vilches, Víctor. (2018). Time-Sensitive Networking for robotics.

[5] Jansen, D.; Buttner, H.: 'Real-time Ethernet: the EtherCAT solution', Computing and Control Engineering, 2004, 15, (1), p. 16-21, DOI: 10.1049/cce:20040104 IET Digital Library, https://digital-library.theiet.org/content/journals/10.1049/cce_20040104

[6] Schemm, E.: 'SERCOS to link with Ethernet for its third generation', Computing and Control Engineering, 2004, 15, (2), p. 30-33, DOI: 10.1049/cce:20040205 IET Digital Library, https://digital-library.theiet.org/content/journals/10.1049/cce_20040205

[7] T. Barbette, C. Soldani and L. Mathy, "Fast userspace packet processing," 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Oakland, CA, 2015, pp. 5-16, doi: 10.1109/ANCS.2015.7110116.

[8] J. Lilley, J. Yang, H. Balakrishnan and S. Seshan, A unified header com- pression framework for low-bandwidth links, in: *International Conference on Mobile Computing and Networking (MobiCom)*, ACM (August 2000).

[9] Subramanian, Sriram, and Sreenivas Voruganti. Software-Defined Networking (SDN) with OpenStack Leverage the Best SDN Technologies for Your OpenStack-Based Cloud Infrastructure. Packt Publishing, 2016.

[10] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, & Dan R. K. Ports (2016). Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (pp. 467–483). USENIX Association.

[11] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 93-112, Atlanta, GA, 1986. USENIX.

[12] Adrian M. Caulfield, Arup De, Joel Coburn, Todor Mallov, Rajesh Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, pages 385-395, 2010.