EASTERN WASHINGTON UNIVERSITY

A Multi-Channel Data Transfer Protocol Using Asynchronous Technology

by

William Czifro

A research report submitted in partial fulfillment for the
degree of Master of Science
in the
Department of Computer Science

May 2018

Contents

List of Figures

List of Tables

Listings

Abbreviations

FTP	**F**ile **T**ransfer **P**rotocol
MCDTP	**M**ulti-**C**hannel **D**ata **T**ransfer **P**rotocol
NCP	**N**etwork **C**ontrol **P**rogram
RFC	**R**equest **F**or **C**omment
RTT	**R**ound **T**rip **T**ime
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol

Chapter 1

Introduction

Computer networking has a firm presence in many modern applications. Applications for social media, video streaming, and many other types of content rely on computer networking to serve content to the users of these types of applications. Whether the communication be via a server using the Hypertext Transfer Protocol (HTTP) or a server using the WebSocket Protocol (WebSockets), the common denominator is the Transmission Control Protocol (TCP) [1][2]. As an underlying protocol to HTTP and WebSockets, TCP provides these higher level protocols, as well as other protocols that use TCP, reliability in communication and data integrity [3]. These assurances offered by TCP have associated costs that directly impact transfer rate and network utilization, or bandwidth.

Many proposed methods for improving the transfer rate and network utilization include solutions like [4][5][6][7] as well as others mentioned in [7][8]. These solutions tend to focus on fine tuning TCP to use a large sliding window or change how the congestion control mechanism of TCP handles a packet loss event. As physical connections increase in bandwidth, fine tuning TCP may not be enough to improve performance, especially for connections that have large Round-Trip-Times (RTT) and/or under utilize bandwidth [9] on network links like [10] and other Gigabit links.

A different approach has been to focus on creating new higher level protocols, or Application layer protocols. Works like [11][12][13] used multiple TCP connections in parallel to try and improve performance without sacrificing reliability in communication. Parallelization poses two major challenges. The first is the additional work the application needs to do to handle multithreading in a way that addresses race conditions, deadlocks, and synchronicity. The second challenge is directly related to the degree of parallelism of a machine. For instance, if a machine has four hyperthreaded cores, it can support eight threads in parallel. In order to fully utilize the CPU, the application has to ensure that all eight threads are busy with work. If a thread goes idle, the application is not using the CPU to its fullest potential. Solutions that use parallelization may be able to improve bandwidth usage and transfer rate, but the shortcoming is under utilization of hardware rendering these solutions suboptimal.

Along with the use of parallel computing, there have been adaptations of the higher-level protocols and systems mentioned in [9] that utilize UDP as the transport for packets from sender to receiver [8][9][14][15][16]. Using a UDP transport can be a viable method to mitigating under utilized networks. The reason for this is, unlike TCP, UDP does not employ any type of congestion control or packet recovery [17]. However, this means that in scenarios where the network is already congested, packets may be lost because UDP will attempt to send packets with no regard for the network conditions. Solutions like these that are UDP-based have to tackles the issue of communication reliability themselves. With increasing capabilities of physical connections with respect to bandwidth, a new challenge presents itself.

Aspera has noted that the challenge with Computer Networking is not so much with slow physical connections, but is instead with the end systems not being fast enough for the connection [14][9]. The path Aspera took involved building a custom end system that could operate faster to handle faster connections. The decision Aspera made begs the question, is it possible to achieve a faster end system by using a more generic technology?

This paper reviews an experimental protocol called Multi-Channel Data Transfer Protocol (MCDTP). The experimental protocol seeks address the aforementioned question by using asynchrony. This report discusses the design of the protocol, the architecture of the implementation, the performances of MCDTP, and the challenges faced by this project. The goal of this project is to show the potential pitfalls of using asynchrony and to propose new ideas for addressing those challenges.

Chapter 2

Background

In order to approach the problem in a unique way, it was necessary to have an understanding of the networking protocols, existing technologies and solutions by other researchers. This background is presented in this section.

2.1 Networking

The Transmission Control Protocol (TCP) was invented in 1978 to provide a “reliable host-to-host" protocol for network communication [3]. By using packet reception acknowledgements and congestion control, TCP provided reliable communication atop the Internet Protocol and became the standard for network communication in 1983 [18]. The User Datagram Protocol (UDP) was created in 1980 as an alternative to TCP [17][19]. The purpose for designing UDP was for applications that needed faster communication.

The TCP transport protocol utilizes two major mechanisms to provide reliable communication: 1) a sliding window coupled with a packet acknowledgement system, and 2) a congestion control system to minimize the occurrence of a lost packet [3]. The sliding window mechanism works by having a view of x packets, referred to as window where x is the window size, of a larger packet buffer that are sent in series. The sender will transmit x packets to the receiver and wait for acknowledgements (ACK) by the receiver [3]. When the first packet in the window is acknowledged, the sender slides the window so that packets waiting to be transmitted enter the view of the window. A packet that is unacknowledged is retransmitted until acknowledged.

Retransmission events are the result of a lost transmitted packet, or a lost ACK packet [3]. These events trigger a congestion control algorithm to take over to try and mitigate the occurrence of another lost packet [20]. The algorithms part of the initial specification for TCP were slow start, congestion avoidance, fast retransmit, and fast recovery. Slow start and congestion avoidance used by the sender to minimize how the number "outstanding data being injected into the network" [20]. The receiver plays a role with the initiation of fast retransmit and fast recovery by notifying the sender that a segment of data has arrived out of order.

In 1977, a committe called Open Systems Interconnection (OSI) a designed architecture for computer communication (network stack OSI-model) [21]. This network stack provided a standard method for networking. The OSI-model consisted of 7 layers (bottom to top): Physical, Data Link, Network, Transport, Session, Presentation, and Application [21]. The OSI-model was later replaced by a 4 layer model (TCP/IP-model) that would become the requirement in order for computers to utilize the the Internet protocol [22]. The TCP/IP-model has the following layers (bottom to top): Link, Network, Transport, and Application.

2.1.1 Optimizing TCP

The congestion control system TCP-Vegas [4] uses RTT to measure throughput and a lower bound threshold and upper bound threshold to perform congestion control. If the lower bound is greater than the measured throughput, TCP-Vegas will increase the sending rate of the packets. If the measured throughput exceeds the upper bound threshold, it throttles how fast the packets are being sent. Consequently, this method reduces the number of retransmits and the sliding window of TCP can continually move forward [4].

Wei, David X and Jin, Cheng and Low, Steven H and Hegde, Sanjay designed FAST to uses RTT and packet loss as measures for congestion control. This method tries to maximize bandwidth usage by being agressive with increasing window sizes until the RTT gets close to a threshold. FAST takes an optimistic approach when exceeding the threshold. It will slowly decrease the window size in hopes that the RTT had degraded for a brief moment. Only when it worsens does FAST aggressively decrease window size.

As a result, all paths in a connection equally share the effects of a bottleneck and thus mitigating the ebbs and flows of traffic providing a consistent throughput [5].

The Binary Increase Congestion Control (BIC-TCP) [6] uses two mechanisms: binary search increase and additive increase. The binary search increase component uses the logarithmic nature of a binary search to rapidly increase the window size at the beginning of transmission and gradually decreases the rate of growth. The additive increase is used as a method of shifting the starting point of the binary search increase in fixed steps such that the logarithmic steps at the behinning are not too large. This component tries to prevent the binary search increase from being too aggressive to the network. The BIC-TCP congestion control system was able to achieve a 95% network utilization [6].

The CUBIC [7] is the successor to the BIC-TCP congestion control system. Rather than using a logarithmic function like BIC-TCP, CUBIC uses a cubic function to increase the window size. This provides a much steadier growth in window size inducing less stress on the network. The concave profile of the cubic function is how CUBIC starts transmission as well as recovers from a packet loss event. The convex profile is the behavior CUBIC takes once it has recovered and has exceeded the window size at the moment of packet loss. The CUBIC system achieves the same network utilization as BIC-TCP, however, the more granular control over window size imposes less stress on the network allowing for friendlier sharing of network resources with other machines [7].

2.1.2 Application Layer Protocols

The Transport layer protocols like TCP and UDP are the foundation for higher level protocols that attempt to optimize throughput at the Application layer [8][9][11][12][13][14] [15][16][23].

2.1.2.1 UDP Based Protocols

Tsunami [15] uses both UDP and TCP to transmit data. Tsunami uses UDP for sending the payload and TCP as a feedback loop. The TCP connection provides signaling for packet loss, retransmission requests, error reporting, and completion report. For retransmission, the server will interrupt the flow to resend the data block in which the packet loss occurred. This approach achieved a 400-450 Mbps throughput when transferring a 5 GB file [15].

Similar to Tsunami in its mechanics, RBUDP [8] uses a TCP connection to send messages between sender and receiver. The major difference is the retransmission mechanism. Retransmission for RBUDP uses the UDP connection after the “bulk data transmission phase" has finished, which is repeated until all packets have made it to the receiver. RBUDP was able to upper bounded packet loss to 7% and achieve a throughput of 550Mbps.

The UDP Data Transfer protocol [16] is strictly a UDP-based protocol that imposes a congestion control mechanism on top of that. There are two UDP sockets used by UDT, one socket labeled “Sender", the other labeled “Receiver". The receiver socket would not only receive data, but it would be used for sending control information. UDT used TCP like messages for congestion control and reliability insurance. During a transfer from Chicago to Amsterdam, UDT had a throughput of 940Mbps over a 1Gbps link.

Aspera’s fasp protocol [14][9] is a UDP-based protocol that uses a rate-based control system by using the RTT of a packet, referred to as “packet delay", to adjust the rate in which packets are being transmitted. fasp handles ratransmissions by resending packets at a rate that makes use of available bandwidth. This provides a continuous flow of data. Aspera is capable of transferring 1TB of data over the course of 2.2 hours [9].

2.1.3 Network Stack Optimization

The Advanced Data Transfer Service (ADTS) [23] is a replacement layer for the Network and Transport layer and is intended to be used of InfiniBand [10] (InfiniBand is network stack that treats network traffic as communication rather than bussed data). Host latency is reduced by ADTS using a zero-copy communication system. It mitigates the need for the CPU to copy data to different places in memory according to the application’s needs. Instead, a send operation will send data directly to an address in memory on the receiver. This mitigation of unnecessary CPU usage and allowance of direct memory access provided a 1GB data transfer in just over 1 second.

2.1.4 Optimization Through Concurrency

The multi-socket FTP (XFTP) [11][12] is a modified version of FTP [17] that uses multiple TCP connections to transfer a single file. The file is divided into records, with the condition that the number of records is greater than the number of connections. When a connection has the resources to send, it is assigned a record to transmit. The receiver reconstructs the original file by placing records in their correct positions. The XFTP could achieve a 90% network utilization using eight connections.

Sivakumar et al. designed PSockets [13] to be a socket programming library that enables the use of parallel TCP sockets. The library provides simple send and receive functions that abstract the mechanics of PSocket. The passed in data is segmented and sent across multiple TCP connections. The receiver reassembles the received data back into its original form. PSockets manages multiple sockets asynchronously so that sockets do not need to simultaneously share resources.

2.2 Asynchrony

Though this paper does not focus on the mechanics of asynchrony, it is a part of the design choices and implementation of this project and thus is worth reviewing. Asynchrony started out as a way for processes to communicate with each other using unbounded buffer channels [24]. A process uses in and out channels for messaging and sharing data with other processes. The purpose is to provide non-blocking behavior. A process does not need to wait for a message or a piece of data to continue doing work, it reacts to incoming messages. This concept has worked its way into programming models and work scheduling [25][26][27][28][29][30].

2.2.1 Scheduling Asynchronous Work

There are two major policies for scheduling asynchronous work, work-first and help-first [25]. The work-first policy, also known as work-stealing, schedules spawned tasks for immediate execution and any continuation of that task is placed in a queue where other workers can steal from. This policy is particularly useful in situations where workers are busy and rarely need to steal from other workers. The help-first policy takes the opposite approach. A worker will ask help from other workers to begin executing spawned tasks. The continuations of the task are executed by the original worker. This policy is useful when stealing is very frequent because stealing can be implemented in parallel and thus increases the throughput of scheduled tasks.

2.2.2 Asynchronous Programming Models

Implementing asynchrony into a language or project can be very challenging. The theory of asynchrony suggests that communication channels need to be unbounded [31]. It is not feasible for computers to have unbounded buffers due to the finite memory a machine has. Thus, integrating asynchrony is not an easy task.

This paper and project focus on newer technologies, but for the sake of posterity it is with mentioning former projects like Cilk and JSD that helped pave the way for implementing asynchrony [28][29][30].

2.2.2.1 .NET Framework Implementation

The .NET Framework by Microsoft provides implementations of asynchrony in their languages C# and F# [27][26]. The C# language received an update in 2009 that added the TPL library that provides asynchronous capabilities in C#. In later updates, C# was introduced with the keywords “async" and “await" that allowed for simpler asynchronous programming [32]. This update meant any synchronous method could be transformed into an asynchronous method by adding only these keywords because the compiler transforms the user code into truly asynchronous code. The asynchronous feature of F# uses the “async" keyword as well, however, the F# compiler provides more granularity in turning a block of user code into asynchronous code [27]. Both languages use the TPL library, which uses the work-stealing policy for scheduling tasks, [26][33], which is a paramount fact because the major performance advantage that F# has over C# is the better compiler [27].

Chapter 3

Design

The architectural design of MCDTP uses both TCP and UDP connections. Like [15], [8], [14], and [9], MCDTP uses the UDP transport protocol for the data transfer and the TCP socket is used for exchanging messages between the server and client. The experimental component of this protocol is the lack of congestion control. As mentioned here, [14] [9], the observed bottleneck resides on the hosts of an end-to-end transfer. MCDTP seeks to mitigate this through multiple asynchronous data channels and the number of channels is determined when implementing this protocol. A channel acts as a pipe from disk to socket and vice versa. This is how MCDTP tries to provide continuous data flow. As a disclaimer, the term “packet(s)" hereinafter does not refer to Transport or Network layer packets, the term is used to refer to an Application layer structure used by MCDTP.

3.1 Protocol Design

The MCDTP protocol focuses on being a very simple and lean protocol to try and minimize the needed time to process incoming messages or data. The protocol consists of three phases: a two step handshake phase, data transmission phase, and packet retransmission phase. For TCP communication, each packet consists of 15 bytes. Each packet is a header with an optional payload. All headers begin with two fields, “ptype" and “subtype". These two fields specify how the packet should be processed. The UDP packet is adjustable in size when implementing MCDTP. It is, however, required to be within the following scope: $13B < size < 64KB$. The 13 byte lower bound is to account for packet header. The 64 kilobyte upper bound can be found here [34]. Unlike the TCP packets, UDP packets include payload size with packet size. Since TCP has the option of a payload, when one exists, a second read from the stream will be required to grab the payload. With the UDP packet, it is guaranteed that a packet will be a set size and thus eliminates the need for a second read.

3.1.1 Two Step Handshake

As with many protocols, MCDTP has a handshake step. However, MCDTP performs two handshakes, the first upon connection and the second before a data transfer. The purpose for the two step handshake is for the sake of extensibility, i.e. another step or function could take place between both handshakes.

3.1.1.1 Specification Handshake

The specification handshake occurs when a client connects to the server. Figure 3.1 illustrates the type of communication that happens during the handshake.

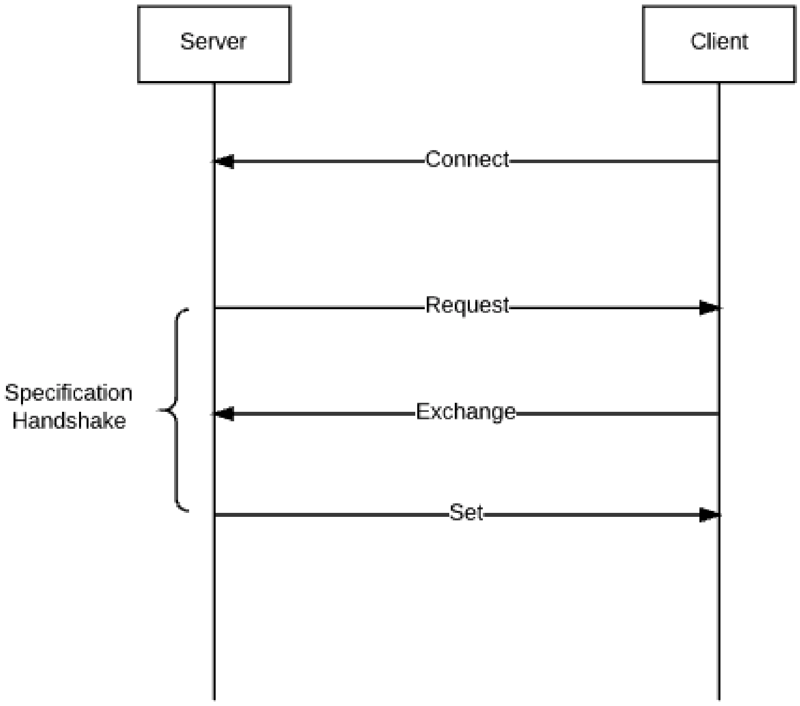

Figure 3.1: Illustration of Specification Handshake

The purpose of this handshake is ensure that both client and server use the same specifications for a data transfer. These specifications currently include the exchange and selection of ports the client should use for each UDP socket. As a side effect, the server can detect how many data channels the client supports, which is key since not all hosts running MCDTP will be capable of using, or configured to use, the same number of data channels. Thus both hosts need to agree upon how many data channels to use. If one host can support more channels than the other, both hosts will agree to use the smaller of the two. This option is optimal over possible alternatives such as increasing the smaller host to match the larger host, or coercing the transfer to work through mismatching channel count, “smaller host" refers to the host with the smaller amount of supported channels while “larger host" refers to the opposite. The aforementioned alternative could work if a host was smaller because it was configured to run conservatively even though the host has the resources to run at a higher performance. However, if the host is smaller because there are fewer resources, then this could potentially cause the host to perform beyond its capacity. Since neither scenario is distinguishable to the client nor server, this option does not work. Coercion could lead to more work needing to be done on the a host depending on which host was larger. Since the concept of a data channel is to pipe file data to network and vice versa, if the client were larger, the server would have to try and multiplex data in a way that the client got roughly equal flow through all channels. If the roles were reversed, the client would need to do additional processing of data to ensure packets were in there appropriate position in file. Ergo, the optimal choice is to simply reduce the larger host to match the smaller host.

Figure 3.2 shows the packet structure of each message, excluding connect, in the handshake. The value of “ptype" and “subtype" are as follow for each packet: $Request \rightarrow ptype = `s', subtype = `r'$, $Exchange \rightarrow ptype = `s', subtype = `N'$, $Set \rightarrow ptype = `s', subtype = `n'$. In this handshake, both Exchange and Set have an additional field, “nPort", and are followed by a payload. The payload consists of port values, which “nPort" specifies how many port values are in the payload. In order to be as language agnostic as possible, a port is represented as 4 bytes, or a 32bit integer. Given this, the payload size can be calculated so that an exact number of bytes can be read during the second read from stream.

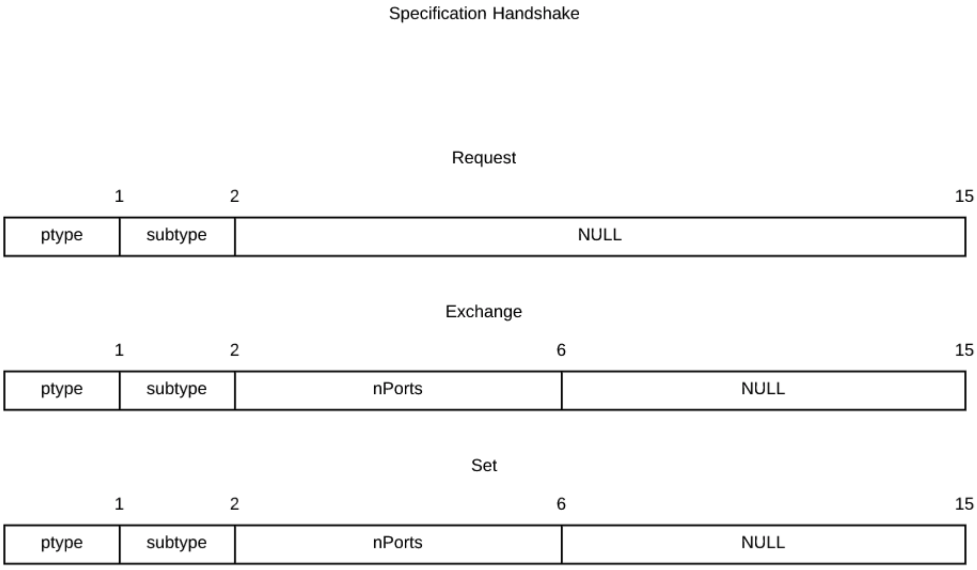

Figure 3.2: Packet Structure of Specification Handshake

3.1.1.2 FTP Handshake

Prior to transferring a file, MCDTP performs another handshake that is similar in design, as can be seen in Figure 3.3.

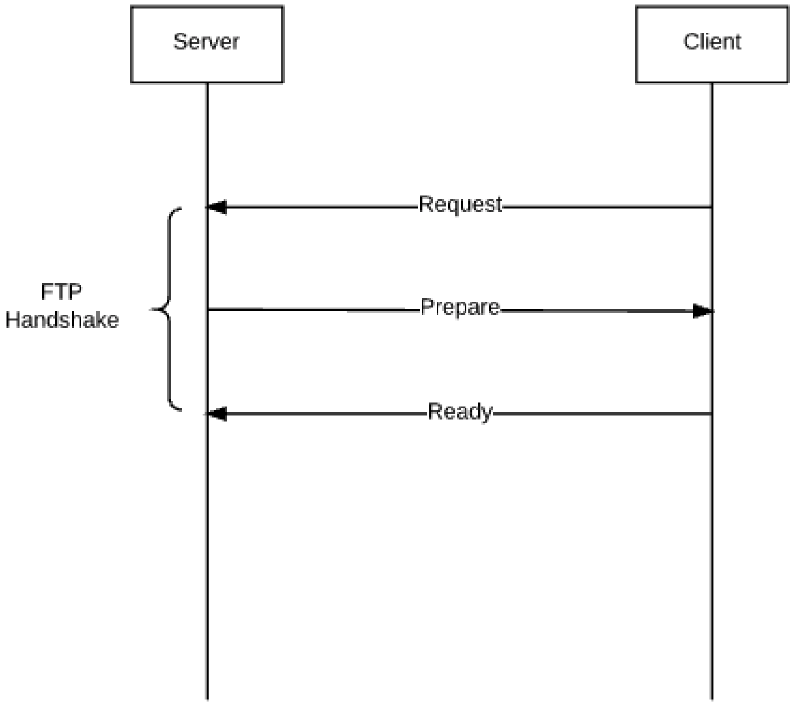

Figure 3.3: Illustration of FTP Handshake

This handshake acts as a concrete step before the data transmission phase and is very simple in design and purpose. This step is so that any asynchronous work that needs to be done first can finish as well as making sure both client and server are prepared for the data transmission phase, which is more of an implementation discussion and will be further discussed in chapter 4. Figure 3.4 is an illustration of the structure of each message in the handshake.

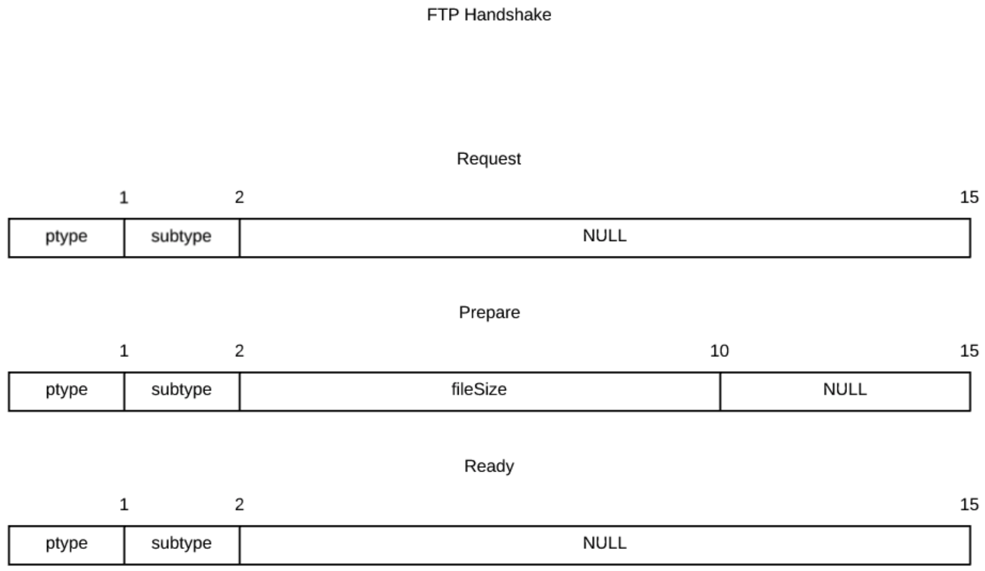

FIGURE 3.4: Packet Structure of FTP Handshake

The value of “ptype" and “subtype" are as follow for each packet: $Request \rightarrow ptype = `t', subtype = `r'$, $Prepare \rightarrow ptype = `t', subtype = `p'$, $Ready \rightarrow ptype = `t', subtype = `R'$. The only information exchanged in this handshake is the size of the file that will be transferred, from server to client. Note that handling file selection has been omitted and will be further discussed in chapter 6. Once the client is prepared for transfer and has signaled the server that it is ready, both hosts begin the data transmission phase of the MCDTP protocol.

3.1.2 Data Transmission

The data transmission phase is the second phase in the MCDTP protocol and is comprised mostly of the transmission of a file using the data channels setup in phase 1. The upper portion of Figure 3.5 illustrates the communication between server and client. The large arrow from server to client at the top represents the data flow for a single channel. The blue in this representation symbolizes the transmission of the file. The packet retransmission phase and this phase have slight overlap, which is further discussed in section 3.1.3.

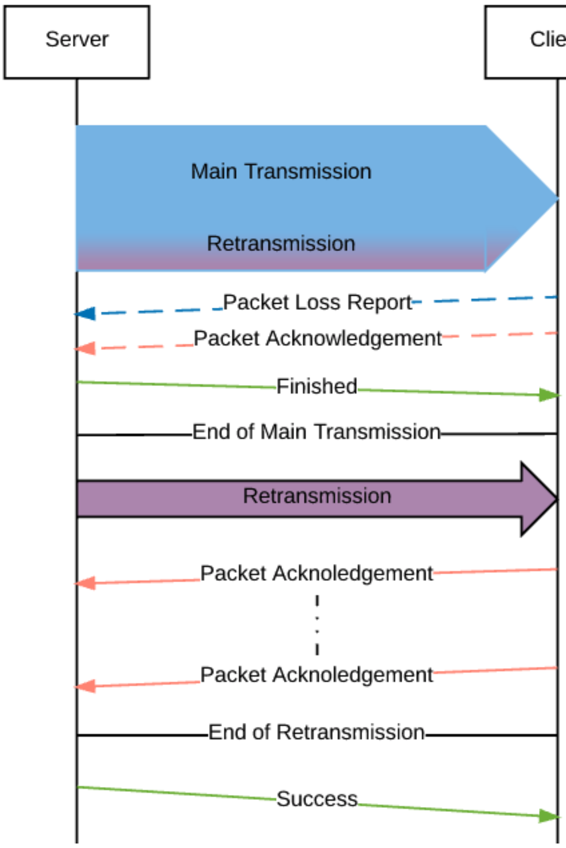

FIGURE 3.5: Illustration of Data Transfer

The packet structure of the data transmitted over each channel is shown in Figure 3.6. Note that the packet size is not definitively set. As stated above, the packet size can range from $13B < size < 64KB$ and is something that needs to be set at an implementation level. The "seqNum" field is the position in file that the data should be written to. To maintain packet size, "dLen" is used to determine exactly how much of "data" is meaningful to the transfer. The "flag" field acts as a control flag. When $flag = 0$, the packet is a regular packet. A retransmitted packet has a "flag" value of 1 and 2 is to signal to the client that the channel is switching to the packet retransmission phase. The final packet is illustrated by the green communication line labeled "Finished" in Figure 3.5, which is sent over the UDP channel.

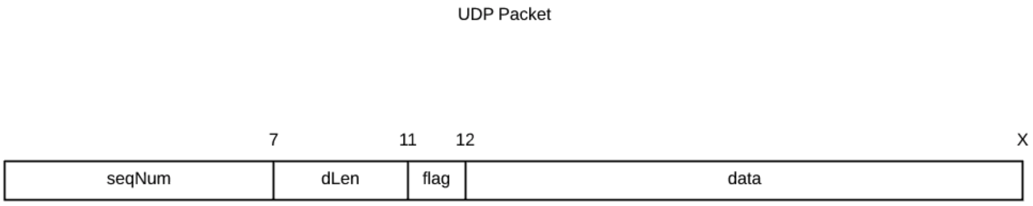

FIGURE 3.6: Packet Structure of Data Transfer

As previously mentioned, MCDTP is an experimental protocol. The experimental component attempts to perform a file transfer without the use of congestion control. The protocol tries to exploit the common operating system behavior that when a socket is created, it is given its own receive buffer. With each channel getting its own receive buffer, there is more space to hold incoming packets giving opportunity for handling more data at the Application layer.

3.1.3 Packet Retransmission

The packet retransmission phase is capable of starting during the data transmission phase. The purpose of this is to hopefully recover packets along the way to minimize the duration of this phase and ultimately achieve a successful transfer faster. As can be seen in Figure 3.5, retransmission consumes a very small portion of bandwidth so as not to impede upon the main transmission during the data transmission phase. The dashed communication lines labeled “Packet Loss Report" and “Packet Acknowledgement", which share the same structure as is evident in Figure 3.7, only occur when packet loss or packet recovery is detected and are sent over the TCP socket. The values for “ptype" and “subtype" are as follows: $PacketLossReport \rightarrow ptype = `t', subtype = `l'$ and $PacketAcknowledgement \rightarrow ptype = `t', subtype = `a'$. The “seqNum" field identifies the UDP packet the message is about and the “port" field identifies which channel the UDP packet belongs to.

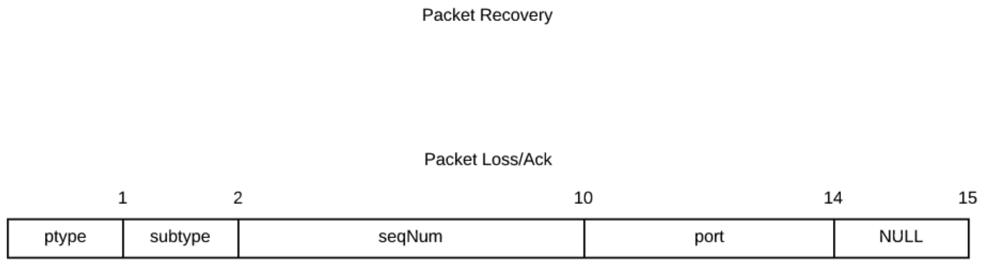

Figure 3.7: Packet Loss and Ack Report Structure

After the data transmission phase finishes, the packet retransmission phase is able to consume more bandwidth. The “Finished" packet is also placed in recovery to ensure the client receives this. Once the client sends an acknowledgement for this packet, that channel will be in the packet retransmission phase on both the client and server, which is represented by the middle segment of Figure 3.5.

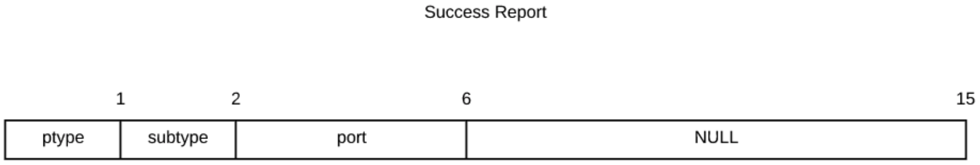

Figure 3.8: Packet Structure of Success Report

Like RBUDP [8], a channel will continue this phase until the server has received an acknowledgement for all packets that are in recovery for that channel. Once all packets have made it, the server sends a success report, Figure 3.8, over TCP thus concluding the work the channel specified by the “port" field. The “ptype" and “subtype" are set to ‘t' and ‘S', respectively. One all channels have succeeded, the session between the client and server is concluded.

Chapter 4

Implementation

There are several paradigms used to construct the implementation of MCDTP. The construction of the implementation uses a hybrid of Functional Programming and Object-Oriented Programming to provide a modular architecture as well as making the project reactive to I/O events. As a side effect to using the paradigms, the programming API is very simple to use in other applications and/or wrappers.

4.1 Why F#?

F# was the chosen language for this project for a couple of reasons. The initial reason is the .NET Framework. As discussed in [26] and [27], the .NET Framework has a very robust built-in library that provides support for asynchronous tasking, especially for C# and F#. What makes this feature unprecedented, especially in C# and F#, is the embedded domain specific language (EDSL), often referred to as “syntactic sugar", that the compiler uses to generate asynchronous code. The EDSL in F#, referred to as “computation expression" or “workflow" in the F# community, is especially powerful at composing asynchronous tasks.

Consider the following synchronous code:

```
let child() =
  System.Threading.Thread.Sleep(1000)
  printfn "Hello from child!"
```

```
let parent() =
  printfn "Parent calling child..."
  child()

parent()
```

Listing 4.1: Synchronous F# Example

The asynchronous equivalent is:

```
let asyncChild =
  async {
    do! Async.Sleep(1000)
    printfn "Hello from child!"
  }

let asyncParent =
  async {
    printfn "Parent calling child..."
    do! asyncChild
  }

Async.RunSynchronously asyncParent
```

Listing 4.2: Asynchronous F# Example

The transformation from synchronous code to asynchronous code is surprisingly simple thanks to the compiler, which is further discussed here [27]. The ease of composing asynchronous tasks helps simplify constructing a reactive implementation, which is further discussed in section 4.3.

The main reason for using F# is the multi-paradigm aspect of the language [35]. F# embodies Functional Programming—making it ideal for modular programming—as well as Object-Oriented Programming—which is great for reactive code—and Language Oriented Programming—enabling the creation of in-language DSLs to act as a declarative language for either configuring how something should behave, or providing a way to interact with an external resource or device without leaving F#. This attribute has made F# a prime candidate for constructing an implementation of MCDTP.

4.2 Modular Architecture

The modular approach to programming breaks a large codebase into smaller chunks called modules based on some criteria [36]. Typically modules address a specific problem. Modules are also agnostic to outside code. This enables the ability to swap out modules for others. Since modules are identified partly by a version number, modules can be updated and not break projects using a module because they will be using a specific version of that module. These attributes of modular programming are the reason the implementation discussed in this paper of MCDTP was constructed with a modular architecture.

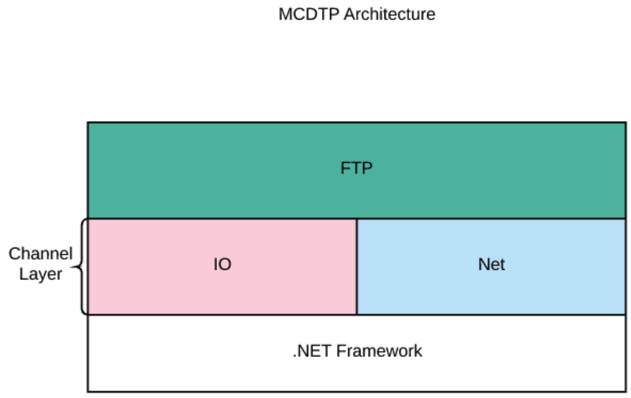

Figure 4.1: MCDTP Modular Architecture

Figure 4.1 illustrates the modular architecture of the MCDTP implementation, hereinafter referred to as MCDTPi. The top module is an FTP module (FTP) and depends on an I/O module (IO) and a Network module (Net). The .NET Framework is the base module of MCDTPi. This module is an external module that was not constructed while building MCDTPi; therefore, it will not be further discussed. Modules, and their respective submodules, will be reviewed in the order of dependency.

4.2.1 Helper Modules

There are two micro-modules that are used thoughout all modules and submodules in MCDTPi. The *Logging* and *Utility* modules, as seen in Figure 4.2, contain helper code that is commonly used throughout MCDTPi. The *Logging* module was created to manage console and log file outputs for easier debugging. This module provides two configurable loggers, a console logger and a network logger. The console logger writes messages to a console. This logger is more of a general purpose logger. The network logger logs in app actions that relate to a socket I/O operation as well as packet loss. This logger writes to a file. Both loggers can be configured to log certain messages based on a priority level. *Logging* provides a level for handling errors so that MCDTPi can be built more robust.

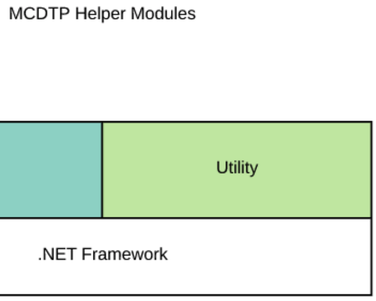

Figure 4.2: MCDTP Helper Modules

The *Utility* module provides code for type conversion, which is mostly used for serialization and deserialization purposes, as well as code for wrapping a function with a semaphor type data structure. This module is used mostly to provide abstractions to reocurring code snippets found throughout MCDTPi.

4.2.2 IO Module

The *IO* module is one of the major modules to MCDTPi. The module handles all I/O related operations, except for socket I/O—see section 4.2.3 for more details. This module is composed of three submodules, as seen in Figure 4.3.

4.2.2.1 MemoryStream Sub-Module

The $MemoryStream$ submodule is largely based on the $MemoryStream$ data structure found in the .NET Framework. The difference is that this submodule provides an unbounded buffer. The .NET Framework data structure has an upper bound on how much data can be written to the $MemoryStream$, which is $\frac{2^{32}}{2} - 1$—or the max value of a signed 32-bit integer. This would be a performance issue because it would restrict the amount of data that can be held in memory to 2GB. This is fine for low end machines,

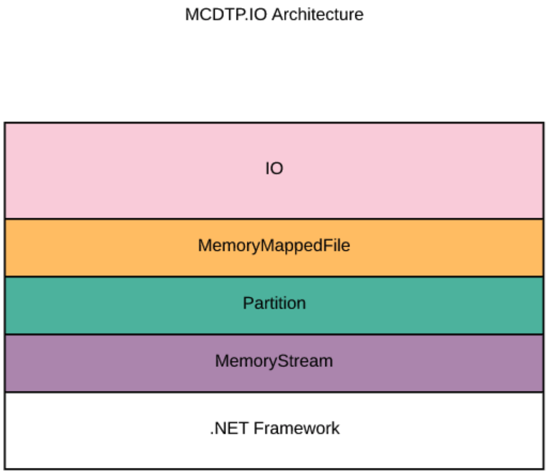

Figure 4.3: MCDTP IO Architecture

but many machines have 8GB or more of memory. The imposed upper bound needed to be mitigated through the use of a custom $MemoryStream$.

Instead of using a single byte array as the underlying container, MCDTPi uses a linked list of byte arrays to provide an unbounded container. Bytes are written to the end of the stream just like the original data structure; however, when a byte array fills up, an empty byte array is added to the end of the list so that more bytes can be added to the stream. As bytes are being read from a stream, the first array shrinks until it is empty, then it is removed from the list and the next byte array will be consumed. Though this stream is unbounded, its API uses bounded arrays as arguments and return values. The context in which this submodule will be used means that it should accept a bounded array as argument for writing to stream as well as returning a bounded array for read operations.

4.2.2.2 Partition Sub-Module

The $Partition$ submodule is actually a submodule to $MemoryMappedFile$ and uses $MemoryStream$ as a dependency. This submodule is used to read and write on part of a file. The “part", or partition, has a $PartitionHandler$. This data structure is used to manager a file pointer within that part of the file. The $PartitionHandle$ keeps track of the state of the buffer and file pointer to ensure that any asynchronous I/O task, involving either the $MemoryStream$ or disk, do not overlap. This submodule is configurable with respect to how frequent a disk I/O operation should happen, whether it is read only or write only, and the console logger configuration it should use. Specifications like where the partition begins in the file and how large the partition is can be configured as well; however, it is highly recommended that those properties be set by the parent module *MemoryMappedFile*.

Additionally, *Partition* handles special writes for data snippets in one of two ways. The initial method is to try and amend the buffer. The position in buffer is determined by the position in file the data snippet belongs to. The position in file is offset by the position of the beginning of the buffer. This gives how far into the buffer the snippet needs to be written. If the beginning of the buffer is positioned after the data snippet in file, then a write needs to happen that involves temporarily moving the file pointer to write the data snippet to disk and returning the file pointer back to its previous position.

4.2.2.3 MemoryMappedFile Sub-Module

The *MemoryMappedFile* submodule is the top submodule in the *IO* module. This submodule is the parent of *Partition*. It is inspired by the .NET Framework data structure of the same name. However, the difference lies in the functionality. Like the original data structure, *MemoryMappedFile* is used to store large portions of a file in memory from different locations within the file. This submodule treats these portions as “partitions", whereas the data structure treats them as “views". A view in the data structure is a portion of a file that is loaded into memory. A partition is a portion of a file that has a dedicated file pointer and the partition is only partially loaded into memory—which would be the view of the partition.

Though it seems like a very insignificant difference, the major reason for creating this submodule was the ability to shift the view to a new position in file. The original data structure does not provide such functionality. Instead, to move a view would require creating a whole new view. This could lead to a struggle in managing memory. To prevent using too much memory, views would need to be smaller. Smaller views would mean more disk I/O operations to try and mitigate any interruption that may occur by either depleting the buffer representing the view, or filling buffer to capacity. This is why a custom submodule was constructed. To provide the ability to have a sliding view.

This functionality is offloaded to the child submodule *Partition* since it is an action that happens to a partition of a file. *MemoryMappedFile*, instead, handles partitioning a file equally so that each partition is roughly of equal size. The configurable properties of a *MemoryMappedFile* are the file name—used to identify a file to read or a new file to write to—the number of partitions to create—though it is recommended that *FTP* set this—the *Partition* configuration to use, and whether this file should be opened as read only or write only.

4.2.3 Net Module

Another core module to MCDTPi is the *Net* module. This module handles network related actions, such as application level packet managing, parsing and composing raw data, and performing socket I/O operations. Like the *IO* module, *Net* is comprised of multiple submodules, as shown in Figure 4.4. Notice that the submodules are side-by-side instead of stacked like the *IO* module. This means that the modules are independent of each other.

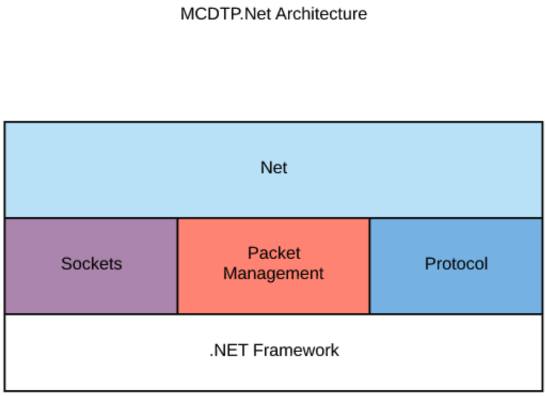

FIGURE 4.4: MCDTP Net Architecture

4.2.3.1 Protocol Sub-Module

The Application Layer protocol that MCDTP defines is a very simplistic protocol. However, the implementation is a little more complex and thus, to help with maintainablility, MCDTPi has split the protocol into two modules and the *Protocol* submodule is one of them. Divvying up the protocol like this means that any future work that changes the messages being sent and received only effects this submodule. This translates to faster compiling of code and in a production scenario means shorter down times. This submodule handles raw data that either came off the wire, or will be sent across the wire. *Protocol* parses and composes raw data, or more commonly known as deserialize and serialize—sometimes called “SerDe"—a data structure, respectively. Both of TCP and UDP packet structures discussed in section 3.1 are handles by this submodule. *Protocol* uses *Utility* to convert raw bytes to more understandable primitive types. *Logging* is also a dependency to provide output for error messaging so that when a message does not parse or compose correctly, the invalid input can be fetched for analysis if there is the need. The API of this submodule will be discussed in section 4.4.

4.2.3.2 Sockets Sub-Module

All socket operations are housed in the *Sockets* submodule. A configurable type class called *SocketHandle* that wraps a socket is provided by this submodule. This type class is capable of performing pre- and post- socket I/O operations. This is more for compatability with *Protocol*. The pre- and post-operations are configurable so that any message handling service can be performed upon sending raw data or receiving raw data. Sending data is an asynchronous operation. Firstly, it is uses the provided asynchronous send function provided by .NET Framework. Secondly, it queues messages if an asynchronous operation is running. The receive operation is asynchronous as well thanks to the .NET Fremework, however, queueing receive requests is not supported. *SocketHandle* is used for both TCP and UDP sockets and can be configured to use either protocol. IPv4 is the only supported version of the Internet Protocol at the moment. Section 4.4 will provide a code snippet on how the *Sockets* module is used.

4.2.3.3 PacketManagement Sub-Module

The second submodule that implements the remainder of the UDP component of MCDTP is *PacketManagement*. This submodule is very complex. *PacketManagement* is used to manage UDP packets. From the server’s perspective, data is loaded from a source and converted to packets using *Protocol*. Packets are buffered and made accessible through a function call. When enough packets have been depleted, another batch is loaded. When there are no more packets to be loaded, the server buffers the final packet as per the design of MCDTP data transmission phase. The final packet is submitted to the front of the retransmit queue to ensure the client receives that packet.

When a packet is lost, the server gets a report that a packet, identified by a sequence number, has been lost. When *PacketManagement* gets this report, the sequence number is submitted to a queue for processing, and, if the retransmit processor has not been initialized, it is setup to run on a configured interval. As reports come in, the retransmit processor will take a report and fetch the data from source and queue it up for retransmission. *PacketManagement* is preconfigured to have 40 packets available for retransmission and send only 5 at a time, which is mostly to not obstruct the flow of the data transmission phase. The interval the retransmit processor executes is configurable. When the time is due to run the retansmit processor, it will copy up to 5 packets and push them on to the front of the main queue. During this time, the processor will also check to see if the retransmit queue needs to be reloaded. If so, any reports that are pending will be processed and moved to the retransmit queue. If there is no data to load and the rtransmit queue is empty, the retransmit processor will uninitialize itself, otherwise, it reinitializes itself for another interval. The acknowledgement system is fairly straight forward. Any acknowledged packets are removed from either queue.

On the client, packet loss is detected by gaps in the packet sequence numbers. Sequence numbers increase by the size of a packet. This is reliable because all packets are of the same size with the exception being the last packet. Thus, when a sequence number increases by more than the size of a packet, a packet is has been dropped. This detection occurs during the flush event. The flush event occurs when the buffer size exceeds a configured threshold. As a packet loss is detected, missing data is temporarily filled in with null values to ensure data remain in its correct position and a report list is compiled of all packets that have been lost. This list is sent over TCP to ensure the server knows which packets need to be retransmitted. When a packet is recovered, an acknowledgement is sent to the server. The data in the recovered packet gets submitted to *Partition* for handling.

4.2.4 FTP Module

The *FTP* module is the top module of MCDTPi. It interacts with the lower modules and helps *Net* interact with *IO*, since they are side-by-side modules, by providing callback functions that can be used by either *Sockets* or *PacketManagement*. *FTP* manages data channels. It ensures that a data channel on the server matches a data channel on the client by pairing the start position of each partition to each port used by a UDP socket in ascending order–each channel is identified by the port. This is reliable because as per MCDTP design, ports are exchanged during the handshake. This module also handles all TCP communication and performs the associated action for a TCP packet. For instance, a packet loss report is directly handled by FTP and forwards the report to the channel with the matching port identification. FTP also prepares the $MemoryMappedFile$ submodule on both client and server. When both hosts are ready, FTP initiates the transmission process. While this is running, FTP exposes a state that represents the state of all channels. When all channels have succeeded, FTP assumes the succeeded state. This module is configurable and propagates the configurations for the lower modules onward.

4.3 Reactive

Reactive Programming, also known as event-driven, is a paradigm that is used throughout MCDTPi. Reactive Programming seeks to make programs “react" or respond to an event that has happened. As a result, there is never any code that is waiting for something to happen. Code only gets executed once an event has triggered it. The reason this is used is to more effectively use threads. Since code is never running waiting for something to happen, threads are not being wasted. With the exception of MCDTP.Net.Protocol and MCDTP.Utility, all modules and submodules are reactive.

When a UDP packet is received on the client, it gets processed by *Sockets* and *Protocol* and is submitted to *PacketManagement*. *PacketManagement* decides if this packet will trigger a flush event or not. If it does not, the packet is simply added to the buffer. If it does, *PacketManagement* asynchronously flushes the buffer by submitting the received data to *Partition*. If this action trigger a flush event within *Partition*, an internal asynchronous task will flush data to disk. This entire chain is reactive. Buffer flushing is an event, not something that is continuously running to see if data needs to be flushed.

The server follows a similar pattern. When a packet was successfully sent, an asynchronous event is triggered that queries a packet from *PacketManagement*. The query to *PacketManagement* tries to pull a packet from buffer to send. This action can trigger an event within *PacketManagement*. If the buffer length falls below a threshold, a replenish event is triggered to pull data from *Partition* and prepare packets from that data asynchronously. *Partition* uses a similar event for the same task. When a packet is queried from *PacketManagement*, it is submitted to *Sockets* and thus repeats the loop. This loop is asynchronous so any packets waiting to be sent will be processed in the meantime. Since this loop is reactive, *FTP* jump starts the loop to get the process going.

4.4 API

F# provides the ability to create computation expression like the *async* keyword in listing 4.2. MCDTPi takes advantage of this to provide a simple API for generating configurations used to configure the modules and submodules. The following code examples illustrate how simple it is to set up an MCDTPi instance on a server and client.

```
// create logger configurations
let console =
  loggerConfig {
    useConsole
    loggerId "Simple Console"
    logLevel LogLevel.Info
  }
let network =
  loggerConfig {
    networkOnly
    loggerId "Simple Network"
    logLevel LogLevel.Info
  }

// create socket configurations
let tcp = socketHandle { useTcp }
let udp = socketHandle { useUdp }

// create partition configuration
let partitionConfig =
  partition {
    // when buffer falls below 50MB, load another 50MB
    replenishThreshold (50 * 1000 * 1000)
    isReadOnly
    attachLogger console
  }

// create memoryMappedFile configuration
let mmfConfig =
  mmf {
    usePartitionConfig partitionConfig
    handleFile fileName
    isReadOnly
  }

// create ftp configuration
let ftpConfig =
  ftp {
    serverMode
    useConsole console
    monitorNetwork network
    configureUdp udp
    configureTcp tcp
    useParser Tcp.Parser.tryParse
    useComposer Tcp.Composer.tryCompose
    channelCount 4
  }

let session = Ftp.acceptNewSessionFromConfig ftpConfig
session.BeginHandshakeAsServer()
```

Listing 4.3: Server Example

```
// create logger configurations
let console =
  loggerConfig {
    useConsole
    loggerId "Simple Console"
    logLevel LogLevel.Info
  }
let network =
  loggerConfig {
    networkOnly
    loggerId "Simple Network"
    logLevel LogLevel.Info
  }

// create socket configurations
let tcp = socketHandle { useTcp }
let udp = socketHandle { useUdp }

// create partition configuration
let partitionConfig =
  partition {
    // when buffer exceeds 50MB, flush to disk
    flushThreshold (50 * 1000 * 1000)
    isWriteOnly
    attachLogger console
  }

// create memoryMappedFile configuration
let mmfConfig =
  mmf {
    usePartitionConfig partitionConfig
    isWriteOnly
  }

// create ftp configuration
let ftpConfig =
  ftp {
    serverMode
    useConsole console
    monitorNetwork network
    configureUdp udp
    configureTcp tcp
    useParser Tcp.Parser.tryParse
    useComposer Tcp.Composer.tryCompose
    channelCount 4
  }

let session = Ftp.connectWithConfig ftpConfig
session.BeginHandshakeAsClient()
// wait for handshake
while session.State = FtpSessionState.Handshake do
  System.Threading.Thread.Sleep(2000) // suspend thread

session.RequestTransfer()
```

Listing 4.4: Client Example

The use of a computation expression eliminates the need to pass around a configuration to numerous functions. As a result, the only function call is to the *FTP* module that converts the *FtpConfiguration* to a *FtpSession*. The *FtpSession* is a handle so that the state of the transfer can be monitored and provides a single point for disposing of all resources allocated by MCDTPi. As a note, *PacketManagement* has a computation expression as well. It is not used in the examples because *FTP* uses it internally. Therefore, it is unnecessary to configure *PacketManagement* externally in these examples.

Chapter 5

Performance Test

As with many network protocols designed to transfer data quickly and reliably, there needs to be performance analysis. The performance of MCDTP is measured by throughput and packet loss.

5.1 Test Environment

The test environment used to test MCDTPi used commodity servers rented from a cloud computing company called DigitalOcean. Two servers were rented, one deployed in New York, USA and the second deployed in Amsterdam, Netherlands—the distance was deliberate to test MCDTPi over WAN. The rented servers were configured with 4 CPUs, 8GB of RAM, 80GB SSD, and 5TB of transfer and run Ubuntu 16.04LTS. DigitalOcean does not disclose any server specifications beyond this. The servers were further configured with the cross-platform version of .NET Framework, .NET Core 1.0.3, in order to execute the tests for MCDTPi.

The Ethernet link of both servers places a constraint on the size of transmitted packets. This constraint is know as the Maximum Transmission Unit (MTU). The MTU for both servers was 1500 bytes. The largest a packet can be is 1500 bytes before it gets fragmented into smaller packets. Fragmentation is not handled by MCDTP, so MCDTPi was compiled with a UDP packet size set to 1400 bytes to account for any packet headers.

5.2 Testing

There were four test configurations applied to MCDTPi. All configurations used a 1GB file as the data source. Since both servers had 8GB of main memory, for each test, MCDTPi was able to hold the entire file in memory. Thus disk I/O was not a factor in performance. The variable between each test was the number of channels MCDTPi used on each test—single-, dual-, quad-, and octa-channel configurations.

Since throughput and packet loss are used as performance measurements, only the Data Transmission phase of MCDTP is tested. This is to test the unreliability of MCDTP. By measuring unreliability, it gives insight as to how much work would need to be done to provide reliability.

5.2.1 Test Results

For the following tests, the server in New York played the role of client and the Amsterdam server played the role of server, which was arbitrarily decided since exact specifications on the servers are unknown and thus could not weigh in on this.

Figures 5.1 and 5.2 show the throughput in bytes for single-, dual-, and quad-channel configurations for both client and server hosts, respectively. These results are discussed in section 5.3.

Tables 5.1 and 5.2 show further statistics on performance. Figures 5.3, 5.4, and 5.5 are the graphical representation of this data. These results are discussed in section 5.3.

Table 5.1: Server Performance

	Single	Dual	Quad	Octa
Average Throughput	1.146Mbps	2.893Mbps	0.545Mbps	0Mbps
Max Throughput	4.385Mbps	6.998Mbps	12.065Mpbs	0Mbps
Packet Loss	14.06%	19.64%	87.97%	100%

Table 5.2: Client Performance

	Single	Dual	Quad	Octa
Average Throughput	1.018Mbps	2.398Mbps	0.52Mbps	0Mbps
Max Throughput	3.879Mbps	5.934Mbps	11.755Mpbs	0Mbps

5.3 Analysis

The expected behavior of MCDTPi is that as resources increase with more channels, throughput and packet loss improve. The test results show that this behavior is partly true going from single-channel to dual-channel. Throughput improved two fold while packet loss had marginal degradation. However, performance worsened in both measurements for quad- and octa-channel configurations. Octa-channel was omitted in Figures 5.1 and 5.2 because tests stalled and yielded no data. Though the Figure 5.4 shows quad-channel achieving the highest throughput, it had an average throughput that was half that of the single-channel. Quad-channel had over 6 times as much packet loss as well. This is all around worse.

This outcome seems counter intuitive to what is expected. Nonetheless, this can be explained. To start, the single-channel itself is severely under-perfomant compared to other protocols like RBUDP and Tsunami. Those protocols achieve 300-500 times as much throughput compared to MCDTP. The major difference between this project and those is that MCDTPi experiments with using asynchrony to enhance performance.

The TPL library is the foundation for asynchrony in the .NET Framework [26]. As discussed in this blog [37], the TPL can bog down the performance of a program if too many tasks are created. The TPL library may provide lightweight task management, especially when compared to multi-threading; however, to manage numerous tasks begins to add overhead thus impacting performance.

The granularity of asynchrony in MCDTPi is at the packet level. This means that for every packet, there is an asynchronous task. For these tests, there are nearly 800,000 asynchronous tasks created to handle only sending packets. The lifespan of a packet requires two more asynchronous task on a single host. This means that for a single file transfer 2.4 million tasks are created, causing significant overhead.

Single-channel and dual-channel were able to operate since the number of scheduled asynchronous tasks was more spead out. Quad-channel and octa-channel scheduled tasks more rapidly and either caused very low throughput, or a complete halt to the program.

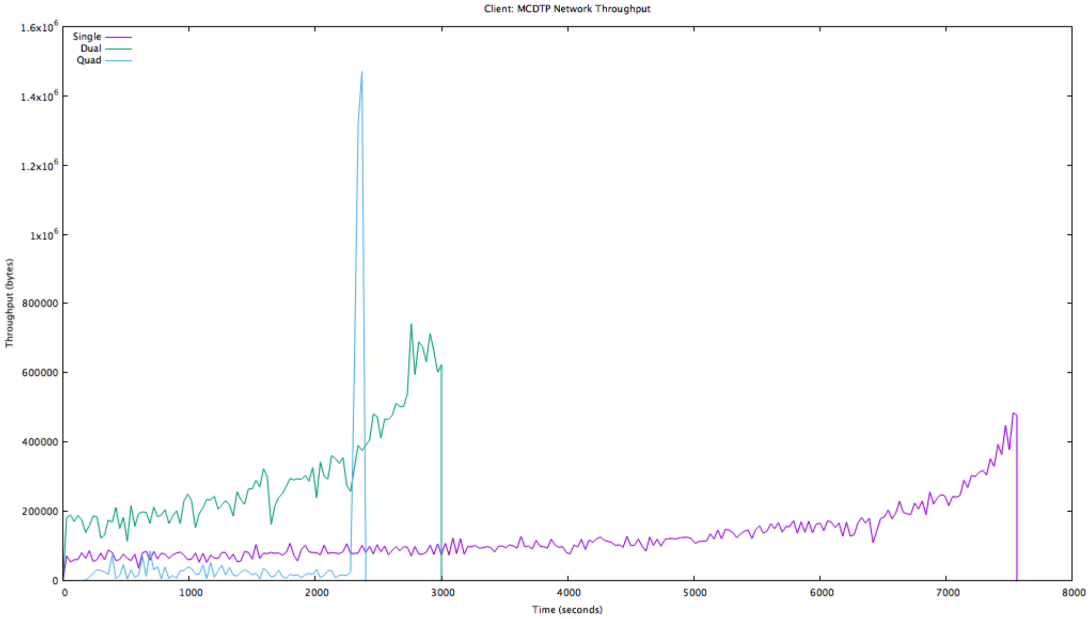

Figure 5.1: Client Throughput

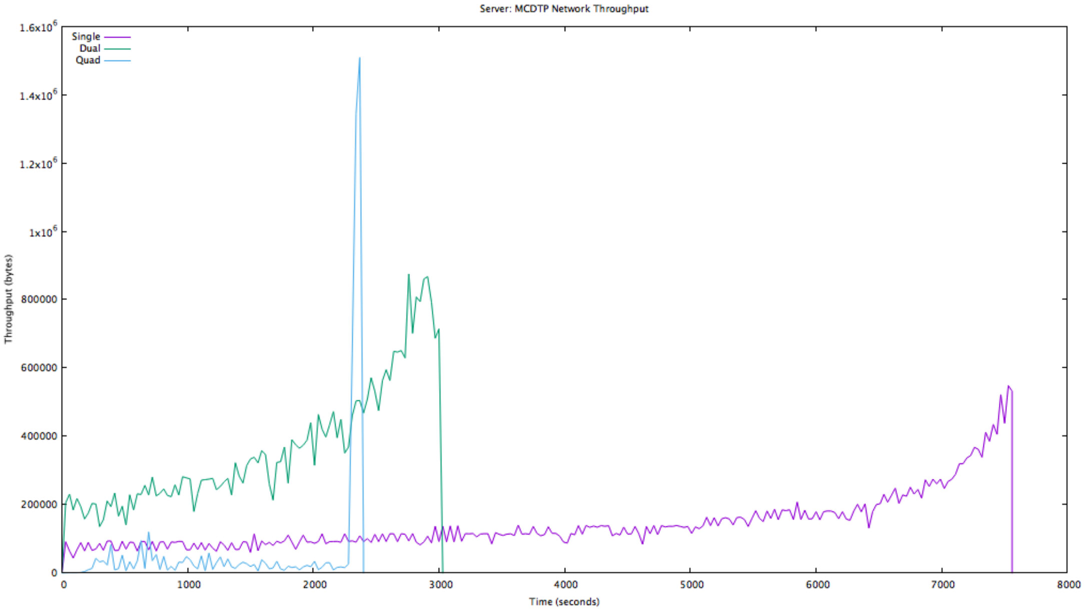

Figure 5.2: Server Throughput

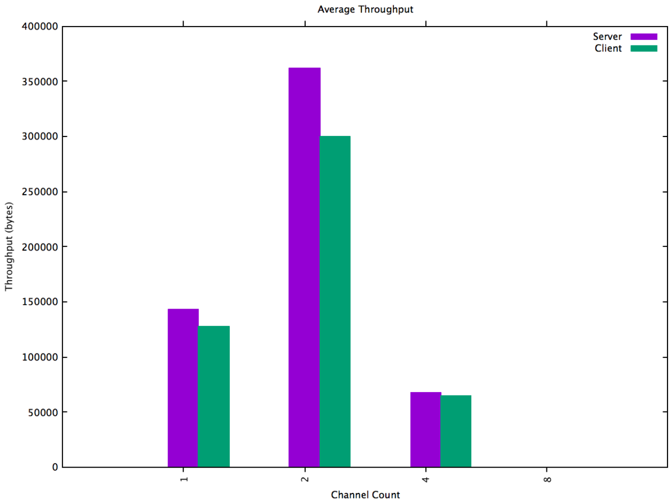

Figure 5.3: Average Throughput

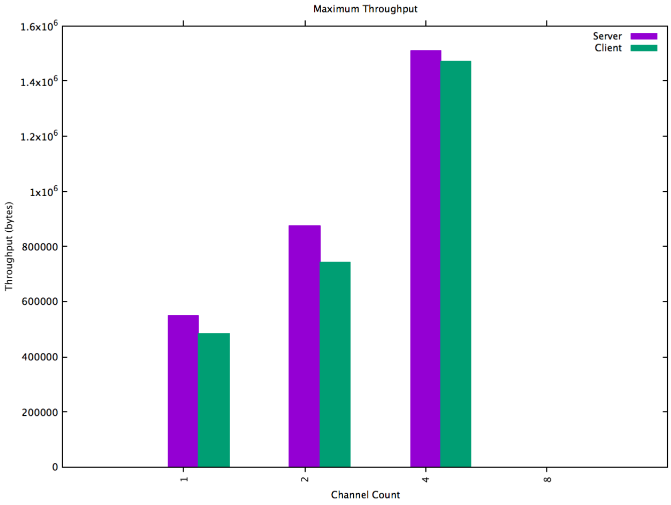

Figure 5.4: Max Throughput

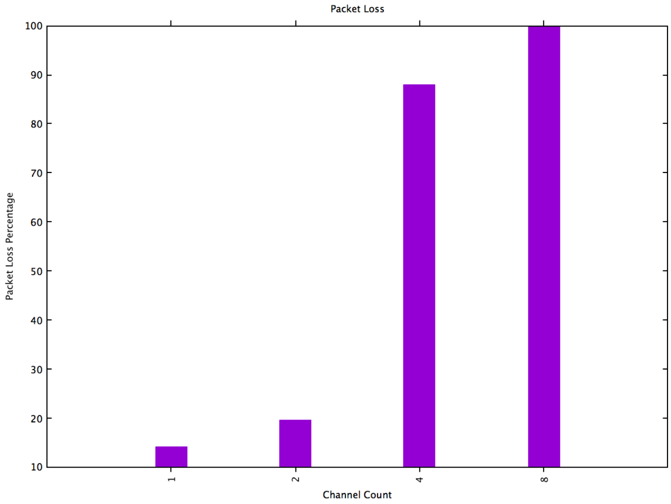

Figure 5.5: Packet Loss

Chapter 6

Challenges and Future Work

6.1 Challenges

There were a number of challenges faced with this project. Some challenges derived from the language, others from poor design choices in early versions of MCDTP and MCDTPi. Challenges related to the language also related to the .NET Framework. The .NET Framework is transitioning to cross-platform support. Tooling targeted C# first and F# second. When a new pre-release was available of .NET Framework, now called .NET Core, F# features would not always work and IDE support was spotty during the early pre-releases. It was not until recent that .NET Core stabilized and the F# support became reliable. Additionally, many features found on Windows are not yet available on macOS or Linux. The development environment used for this project was macOS and the test environment Linux. This placed a hindrance on testing MCDTPi with performing I/O using no kernel buffering. Attempts at using native C to perform these actions was not fruitful due to the inability to link a C library to the F# library. As a result, MCDTPi is currently confined to .NET Core in terms of platform specific feature support.

With respect to MCDTP and MCDTPi design challenges, early iterations had performance challenges. Some challenges were with the reactive programming. The built-in asynchronous feature in F# is not sufficient for implementing reactive code. Early versions suffered from having stacking flush operations that would lead to a disk I/O operation blocking the next packet receive operation resulting in higher packet loss.

This required restructuring MCDTPi to use reactive programming. Also, the design of MCDTP was affected by the poor structure. MCDTP was originally designed to use file checksumming to determine which portions of the file needed retransmission. However, this proved to be very inconsistent, partly due to the slowness, and was replaced with the $PacketManagement$ system discussed in section 4.2.3.3.

6.2 Future Work

To focus on the experiment of MCDTP, a major feature left out, a user interface. However, as noted in section 3.1.1.2, MCDTP was designed with extensibility in mind. MCDTP has a two-step handshake, one for specifications, another to prepare for an FTP session. A user interface can be injected between the two handshakes, which could either be a CLI or GUI. There is also room for this with the current MCDTPi. The API separates the two handshakes as separate functions opening the possibility for another process to happen. Adding this feature would increase usability.

The asynchronous tasking is an overhead that could be mitigated. Restructuring MCDTPi to more efficiently use asynchrony would reduce the performance impact of TPL. Asynchronous tasks would need to be used on packets in bulk rather than per packet. This is only applicable to MCDTPi. The .NET Framework handles sending packets asynchronously and thus is infeasible to modify. Another option would be to implement a custom asynchronous module that tries to mitigate the effects of using TPL. These options would need to be explored in separate work.

The data transmission phase could be optimized as well. As outlined in the design of MCDTP, a UDP packet can range from $13B \rightarrow 64KB$ in size. However, MCDTPi, is currently limited to $1500B$ in size due to the MTU of the network. Packets larger in size are fragmented to fit this within this limit. This limitation effects bandwidth usage. To circumvent this, packets could be aggregated together to fit within the $64KB$ limit. Then, when a packet is fragmented, it is broken up into the smaller packets and thus maintaining data integrity. This would need to be tested in separate work to observe what performance gain may or may not occur.

Chapter 7

Conclusion

There are many ways to address the demand for reliable and fast network communication. Many solutions default to using the TCP Transport protocol to address reliability and simply tune TCP to improve speed [4][5][7]. Other protocols seek to harness the speed that UDP provides and impose a packet recovery system to ensure reliability [8][14][9][15][16]. Multi-socket options try to divvy the work required to transfer data across multiple TCP sockets [11][12][13]. This paper reviewed a new protocol that sought to use a multi-socket UDP-based protocol using asynchronous technology.

The design of the protocol is lean to minimize protocol overhead. MCDTP employs a phase system like RBUDP and crosses it with Tsunami's packet recovery system. Additionally, MCDTP uses asynchrony to manage multiple data channnels. The data channels asynchronously flow data from disk to socket.

While asynchrony has many benefits, this paper has shown that asynchrony can be a detriment to host performance when used liberally. The performance of MCDTP was doubly impacted by asynchrony. Throughput in general was hindered and in a multi-channel setup, packet loss worsened.

The goal of this project was to shed light on how asynchrony impacts network protocol performance. MCDTP and the performance results of MCDTPi has achieved that goal.

Bibliography

[1] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[2] Ian Fette. The websocket protocol. 2011.

[3] V. Cerf and J. Postel. Specification of internetwork transmission control program. *TCP Version*, 3, 1978.

[4] L.S. Brakmo and L.L. Peterson. Tcp vegas: End to end congestion avoidance on a global internet. *IEEE Journal on selected Areas in communications*, 13(8):1465–1480, 1995.

[5] David X Wei, Cheng Jin, Steven H Low, and Sanjay Hegde. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM transactions on Networking*, 14(6):1246–1259, 2006.

[6] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control (bic) for fast long-distance networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2514–2524. IEEE, 2004.

[7] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[8] E. He, J. Leigh, O. Yu, and T. A. DeFanti. Reliable Blast UDP: Predictable high performance bulk data transfer. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, 2002-January(March):317–324, 2002.

[9] X. Fan and M. Munson. Petabytes in motion : Ultra high speed transport of media files. In *SMPTE Annual Technical Conference*, pages 2–13, 2010.

[10] G.F. Pfister. In introduction to the infiniband architecture. *High Performance Mass Storage and Parallel {I/O}: Technologies and Applications*, (42):617–632, 2001.

[11] M. Allman and S. Ostermann. Data transfer efficiency over satellite circuits using a multi-socket extension to the file transfer protocol (ftp). *Proceedings of the ACTS Result Conference*, 1995.

[12] M. Allman and S. Ostermann. Multiple data connection ftp extensions. Technical report, 1997.

[13] H. Sivakumar, S.M. Bailey, and R.L. Grossman. Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks. *ACM/IEEE SC 2000 Conference (SCâĂŹ00)*, (April 2016):38âĂŞ38, 2000.

[14] Aspera. Aspera fasp high speed transport - a critical technology comparison. Technical report, 2016.

[15] M. Meiss. Tsunami: A high-speed rate-controlled protocol for file transfer. Technical report, 2004.

[16] Y. Gu and R.L. Grossman. Udt: Udp-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.

[17] J Postel. User datagram protocol. Internet Request for Comments, August 1980.

[18] E. Andrews. Who invented the internet?, 2013.

[19] C. Kozierokr. Udp overview, history, and standards, 2005.

[20] Mark Allman, Vern Paxson, and Ethan Blanton. Tcp congestion control. Technical report, 2009.

[21] Hubert Zimmermann. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, 28(4):425–432, 1980.

[22] Robert Braden. Requirements for internet hosts-communication layers. 1989.

[23] P. Lai, H. Subramoni, S. Narravula, A. Mamidala, and D.K. Panda. Designing efficient ftp mechanisms for high performance data-transfer over infiniband. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 156–163. IEEE, 2009.

[24] Mark B Josephs, C A R Hoare, and He Jifeng. A Theory of Asynchronous Processes. Technical report, 1989.

[25] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. Work-First and Help-First Scheduling Policies for Terminally Strict Parallel Programs. *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS'09)*, pages 1–12, 2009.

[26] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. *ACM SIGPLAN Notices*, 44(10):227, 2009.

[27] D. Syme, T. Petricek, and D. Lomov. The f# asynchronous programming model. In *International Symposium on Practical Aspects of Declarative Languages*, pages 175–189. Springer, 2011.

[28] A. Sutcliffe. *Jackson system development.* Prentice Hall International (UK) Ltd., 1988.

[29] J. R. Cameron. An Overview of JSD. *IEEE Transactions on Software Engineering*, 12(2):222=240, 1986.

[30] M. Frigo, C. Leiserson, and K. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the 1998 ACM SIGPLAIN Conference on Programming Language Design and Implementation (PLDI)*, volume 33, pages 212–223, 1998.

[31] J. He, M. B. Josephs, and C. A. R. Hoare. A Theory of Synchrony and Asynchrony, 1990.

[32] A. Davies. *Async in C# 5.0*, volume 1. O'Reilly Media, Inc., 2012.

[33] Microsoft Research Group. *F#*. Cambridge, United Kingdom, 2017.

[34] J Postel. Internet protocol. Internet Request for Comments, September 1981.

[35] Tomáš Petříček. F # Language Overview, November 2007.

[36] Sun Microsystems. The Benefits of Modular. chapter 2, pages 9–19. Sun Microsystems, 2007.

[37] Eric Lippert. Asynchrony in C # 5 Part Five: Too many tasks, November 2010.