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

August 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 congrestion 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 tackle the issue of communication reliability themselves. However, with increasing capabilities of physical connections with respect to bandwidth, a different challenge presents itself regarding packet loss.

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 [9][14]. 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 to address the aforementioned question by using asynchrony. This report discusses the design of the protocol, the architecture of the implementation, the performance of MCDTP, and the challenges faced by this project. The goal of this project is to illustrate the effects of asynchrony with respect to data processing in the context of Computer Networking.

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 are used by the sender to minimize how the number of "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) designed an architecture for computer communication, referred to as 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 so that the logarithmic steps of the binary search increase at the beginning 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 [9][14] 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 retransmissions 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 with InfiniBand [10] (InfiniBand is a network stack that treats network traffic as communication rather than bussed data). ADTS uses a zero-copy communication system to reduce host latency. 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 speed up of 65% compared to using UDP.

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 PSockets. 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. PSockets achieved a throughput of 70 Mb/s.

2.2 Asynchrony

Asynchrony started out as a way for processes to communicate with each other using unbounded input and output buffer channels [24][25]. In a scenario where an asynchronous process P_1 has a queue full of input messages, if P_1 needs to send a message to another asynchronous process P_2, P_1 does not need to suspend itself to wait for a response message from P_2 [25]. This concept has worked its way into programming models and work scheduling [26][27][28].

2.2.1 Scheduling Policies

There are two major policies for scheduling asynchronous work, work-first and help-first [26]. 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 .NET Framework Asynchrony Model

The .NET Framework by Microsoft provides implementations of asynchrony in their languages C# and F# [27][28]. Both languages use the TPL library to incorporate asynchrony technology [27][29]. The TPL library was first introduced in 2009 and in later updates to C#, the keywords “async" and “await" simplified asynchronous programming [30]. In C#, these keywords instruct the compiler to transform user written code into tasks that operate asynchronously that are represented as *Task* objects [27]. The F# language uses computation expressions, a feature exclusive to F#, to build asynchronous work flows represented as *Async* objects [28]. It is worth noting that the underlying implementation of F#'s *Async* object uses C#'s *Task* object, which means that any difference in performance is marginal.

The TPL library represents an asynchronous process as a thread [27]. The thread receives *Task* objects as input messages on the input buffer channel. The snippet of code that the *Task* object encapsulates is executed on the thread and the return value is sent over the output channel to the parent process or *Task* object [27]. The TPL library uses a pool of threads, referred to as a *ThreadPool*, that collectively handle *Task* objects. The *ThreadPool* uses a work-stealing policy to schedule *Task* objects for execution to maximize its thoughput.

Chapter 3

Design

The architectural design of MCDTP uses both TCP and UDP connections. Like [8], [9], [14], and [15], 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, [9] [14], 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 a continuous data flow. Implementation details are further discussed in Chapter 4. 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 simple and lean 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 the sake of consistency and a simple design, control messages transmitted over the TCP connection are fixed to 15 bytes in length. 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 does not have a fixed size like the TCP packets. The size of the UDP packet is at minimum 13 bytes to accommodate for header fields, as shown in Figure 3.6. The packet size is contrained to a maximum of 64 kilobytes due to a limitation of the underlying Internet Protocol [31]. Unlike the TCP packets, UDP packets include payload size with packet size. Since UDP has the option of a payload, when one exists, a second read from the stream will be required to grab the payload. With the TCP 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.

The packet structures in Figure 3.2 correspond to the handshake exchanges in Figure 3.1. The *Request* message is how the server signals to the client that the server needs to know the specifications of the client. The specifications shared by the client are the ports of the open channels on the client. The client sends this data to the server using the *Exchange* packet as a header that informs the server of how many port values are being sent. The server uses this exchanged specification to decide how many channels will be used and which ports the client should listen on. When deciding the number of channels, the server will side with the host that supports fewer channels. Once the server decides the specifications the client should use, the server responds with a *Set* packet followed by the port values the client should use. The client will configure itself to use the channels corresponding to the received ports.

The *Exchange* and *Set* packet only express the number of port values that are being sent. The suggested implementation for receiving the port values is to represent port

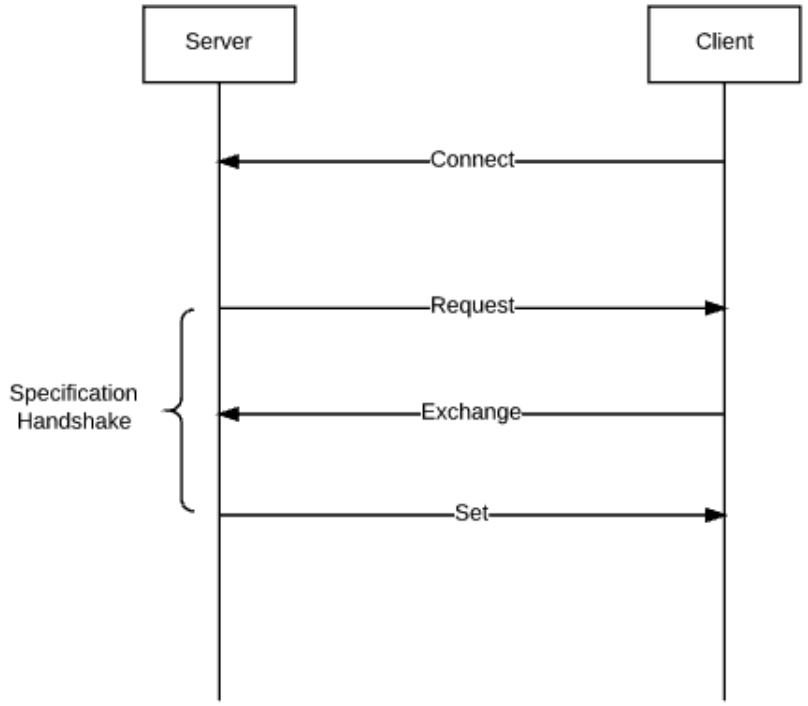

FIGURE 3.1: Illustration of Specification Handshake. This handshake is ensure that both client and server use the same specifications for a data transfer.

values as 32bit integers, or 4 bytes, and leverage that fact while reading each port off of the stream. A more optimal approach is to calculate the payload size using the value from the packet header and the size of a port to read in all ports at once.

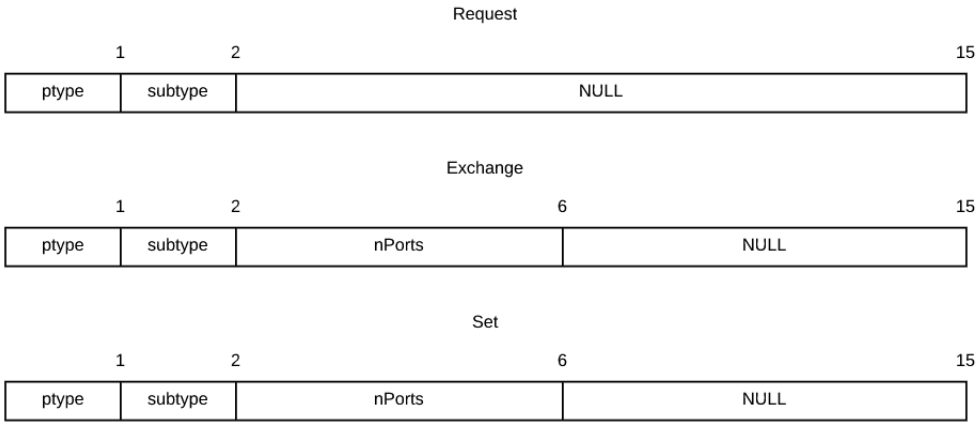

FIGURE 3.2: Packet Structure of Specification Handshake. The fields *ptype* and *subtype* are for identifying packets. The values of these fields are an implementation detail.

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.

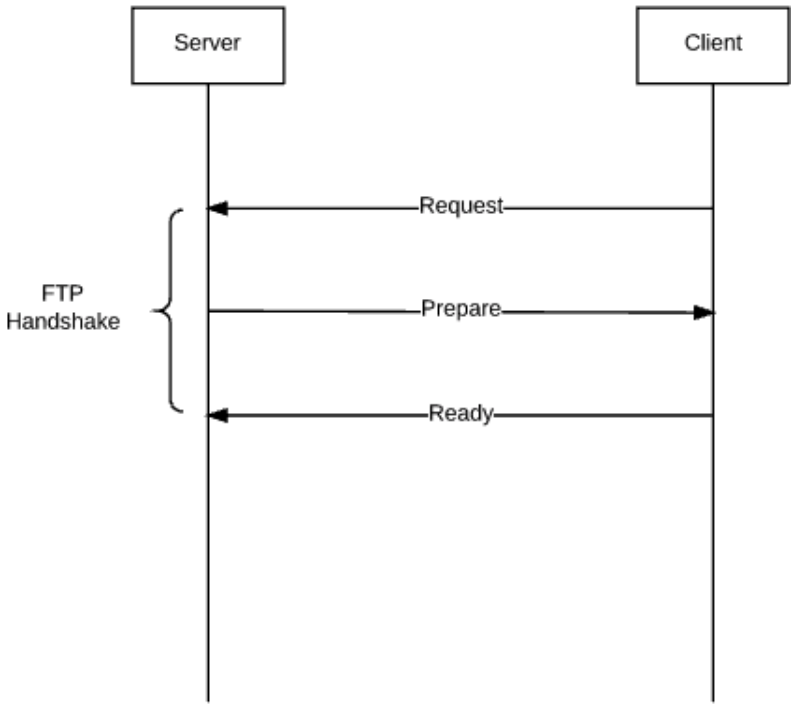

FIGURE 3.3: Illustration of FTP Handshake

This handshake acts as a concrete step before the data transmission phase and is minimal 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.

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.

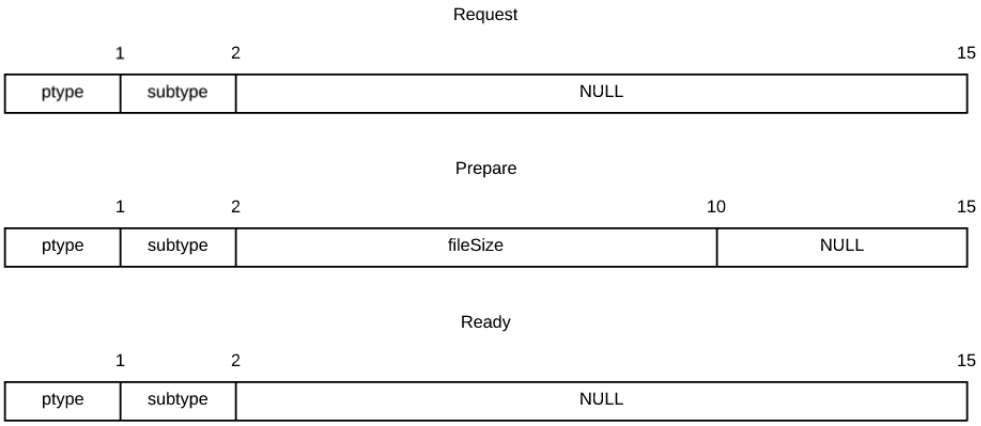

FIGURE 3.4: Packet Structure of FTP Handshake. The fields *ptype* and *subtype* are for identifying packets. The values of these fields are an implementation detail.

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 packet retransmission phase and this phase have slight overlap, which is further discussed in Section 3.1.3.

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 previously, the packet size can range from $13B \leq size \leq 64KB$ 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.

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

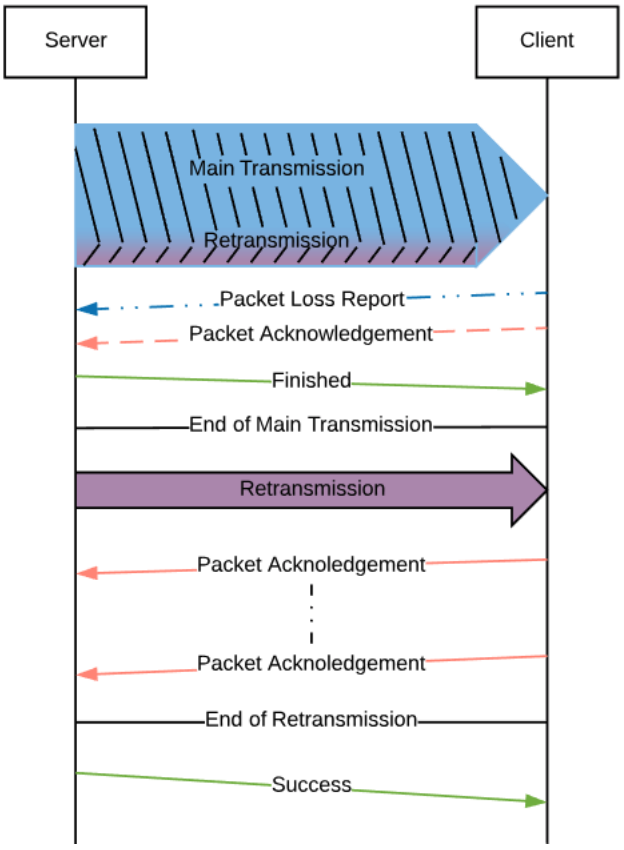

FIGURE 3.5: Illustration of Data Transfer
The large arrow from server to client at the top represents the data flow for a single channel. The blue in this representation symbolizes the transmission of the file.

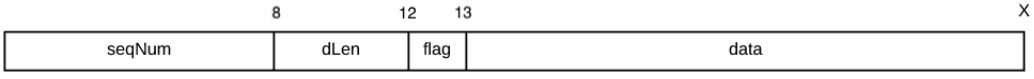

FIGURE 3.6: Packet Structure of Data Transfer (UDP Packet). The packet is variable in size.

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 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 in regards to and the “port" field identifies which channel the UDP packet belongs to.

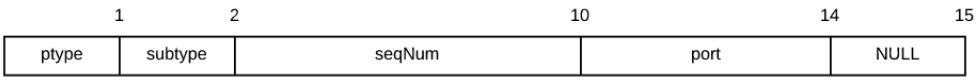

FIGURE 3.7: Packet Loss and Ack Report Structure for Packet Recovery

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.

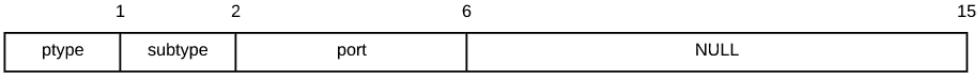

FIGURE 3.8: Packet Structure of Success Report

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. Once all channels have succeeded, the session between the client and server is concluded.

Chapter 4

Implementation

Modular and asynchronous programming paradigms are employed by the implementation of MCDTP. This Chapter discusses how the implementation of MCDTP uses these paradigms.

4.1 Why F#?

The F# language, like C#, can run atop the .NET Core Framework [27][28]. F# was chosen over C# because of the features that are exclusive to F#. The key aspect of F# is the embedded domain specific language (EDSL) [28]. The EDSL in F# adds a component to F# called "computation expression" that provides an abstraction to using features like asynchrony [28]. The F# compiler allows for custom computation expressions to be defined as will be shown in Section 4.4.

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 straight forward thanks to the compiler, which is further discussed here [28]. The ease of composing asynchronous tasks helps simplify constructing an asynchronous workflow, which is further discussed in Section 4.3.

4.2 Modular Architecture

The modular approach to programming breaks a large codebase into smaller chunks called modules based on some criteria [32]. Typically modules address a specific problem. Modules are also agnostic to outside code. This enables the ability to swap out modules for others. The implementation of MCDTP uses a modular architecture to help manage the complexity the codebase.

The top module is an FTP module and depends on an I/O module and a Network module. The .NET Framework is the base module of MCDTPi that uses the cross-platform variant called .NET Core Framework [33]. MCDTPi was built using version

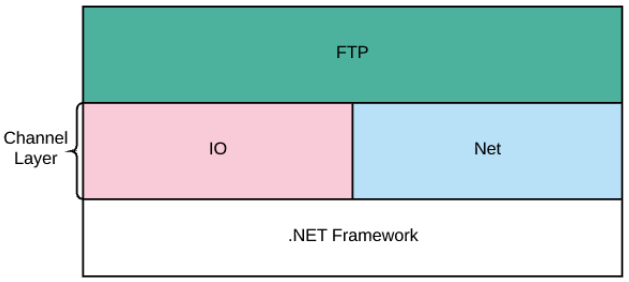

FIGURE 4.1: The architecture of the MCDTP implementation, hereinafter referred to as MCDTPi

1.0.3 of the .NET Core Framework [34]. Its source code can be found here [35]. The MCDTPi 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 is a general purpose logger that writes trace messages and debug messages to the console to provide real-time diagnostics of MCDTPi. The network logger logs events pertaining to Socket I/O operations like packet transmission or reception as well as packet loss events to a file for each channel. Both loggers support log level configurations to filter out messages below a certain priority. *Logging* provides a level for handling errors so that MCDTPi can be built more robust.

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.

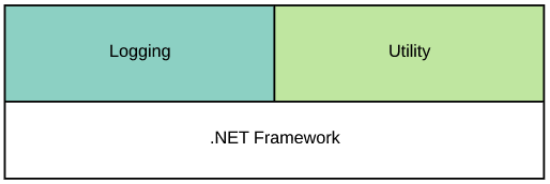

FIGURE 4.2: MCDTP Helper Modules

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.

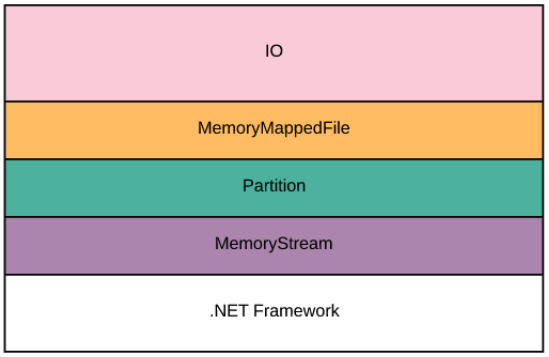

FIGURE 4.3: MCDTP IO Architecture

4.2.2.1 Memory Stream 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,

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 Memory Mapped File 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 the differences seem marginal, 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. Views would also need to be smaller to minimize the footprint on memory required to hold at most two views in memory. 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 for reducing complexity of managing data in memory.

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.

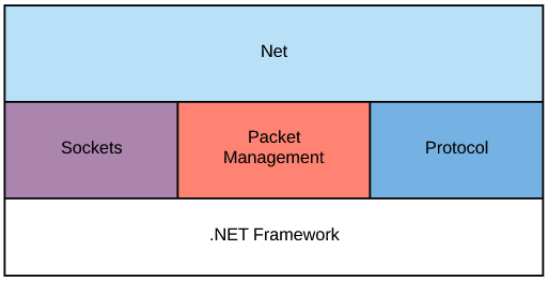

FIGURE 4.4: MCDTP Net Architecture

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 Packet Management Sub-Module

The second submodule that implements the remainder of the UDP component of MCDTP is *PacketManagement*. This submodule has many working parts, as will be discussed. *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 the 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 Asynchrony

Asynchrony is used extensively throughout MCDTPi. It is used not only at the implementation level, but at an architectural level as well so that modules that a siblings do not impact the either's performance.

For instance, 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. When the *Socket* module submits data to *PacketManagement* module, the *Socket* module is not blocked by *PacketManagement* while it decides what to do. The *Socket* module can return to receiving packets. The same interaction exists between *PacketManagement* and *Partition*. This data flow from *Socket* to *Partition* is on the client. The server's data flow is in the opposite direction, from *Partition* to *Socket*, but has the same non-blocking characteristic.

4.4 API

F# provides the ability to create computation expressions like the *async* keyword in Listing 4.2. MCDTPi takes advantage of this to provide a concise API for generating configurations used to configure the modules and submodules. The following code examples illustrate how straight forward 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 {
```

```
    clientMode
    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 was comprised of two servers rented from a cloud provider called DigitalOcean: a server in New York, USA and another server in Amsterdam, Netherlands. The rented servers ran Ubuntu 16.04 LTS and had the following configuration: 4 CPUs, 8GB of RAM, 80GB SSD, and 5TB of transfer. The trsnsfer specification is with respect to DigitalOcean tracking how much data passes through their network and imposing restrictions on rented servers. Unfortunately, DigitalOcean does not disclose any server specifications beyond this.

5.2 Testing

The executed tests used specific configurations. The implementation of MCDTPi was configured with single-, dual-, quad-, and octa-channel counts. Tests were also configured to load data from a 1GB data source. The role of client and server were arbitrarily assigned to the rented servers since both servers are perceivably identical. The Maximum Transmission Unit (MTU) for both servers were 1500 bytes. Fragmentation is not

handled by MCDTP, so to avoid unpredictable behavior, MCDTPi was compiled to use a UDP packet size of 1400 bytes, this accounts for any packet headers.

5.2.1 Test Results

Throughput and packet loss were the metrics of focus while executing tests. The built-in network logger was used to track packet loss and tcpdump was used to capture time and number of bytes as packets were transmitted and received. Both metrics were captured to file in real-time and later a custom built program was used to process the raw data. For instance, there were time gaps between transmitted packets that needed to be filled in as times where 0 bytes were sent. The raw data was also aggregated into one second intervals so that throughput could be calculated. For the following figures and tables, the aggregated data was further aggregated into 60 second intervals so that the behavior could be better observed.

Figures 5.1 and 5.2 show the throughput in Kilobits per second for single-, dual-, quad-, and octa-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	17.9Kbps	18.1Kbps	2.7Kbps	0Kbps
Max Throughput	61.8Kbps	108.1Kbps	146.3Kbps	0Kbps
Packet Loss	14.06%	19.64%	87.97%	100%

TABLE 5.2: Client Performance

	Single	Dual	Quad	Octa
Average Throughput	15.9Kbps	14.9Kbps	2.6Kbps	0Kbps
Max Throughput	55.2Kbps	90.0Kbps	143.3Kbps	0Kbps

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.

At the surface, this behavior seems to be inexplicable. When looking at the implementation of MCDTP, there is no bottleneck in the architecture that would cause packet transmission to be under-performant. Disk I/O operations occur asynchronously in such a way that does not interrupt the flow of data. If the issue does not exist within MCDTPi, then it must exist elsewhere.

As stated in Section 4.2, the .NET Core Framework is the foundation for MCDTPi. It has a significant role in the performance of MCDTPi and warrants investigation. The TPL library was first to be inspected as it is key to asynchrony in MCDTPi and is a major component of the .NET Core Framework [27][35]. The Common Language Runtime (CLR), the underlying runtime system that executes code written atop the .NET Framework and .NET Core Framework, reveals how TPL handles I/O operations [36].

When the TPL library handles an I/O operation, it requests that the CLR use I/O Completion Ports (IOCP) to finish the I/O operation asynchronously [36]. The unique feature about IOCP is that it is a pool of threads (ThreadPool-IOCP) designated for waiting on I/O operations to complete so that the pool of threads (ThreadPool-TPL) used by TPL are used only for computation tasks. According to [36], IOCP is part of the Win32 Kernel API. This can be confirmed when looking at [35].

The source code for the .NET Core Framework indicates that technology equivalent to IOCP for Unix based operating systems is not being used for non-Windows variants [35]. This means that I/O operations are handled by the TPL library and that computation

tasks need to compete for resources against I/O operations. In scenarios where there are a significant number of asynchronous tasks, of both computation and I/O operations, the TPL library will under-perform. There is also the possibility that the TPL will deadlock if there are insufficient resources to handle the completion of an I/O operation as it must queue another task when the hardware signals the operation finished [36].

This characteristic was not disclosed by [33][34][35] and thus this concern was not factored into the architecture of MCDTPi. Consequently, MCDTPi liberally used asynchrony to handle every packet that was transmitted and received. To transfer a 1GB data source requires more than 700,000 packets. Each packet is handled within the *Socket* module is handled by 4 nested asynchronous computation tasks before calling the inner most nested asynchronous I/O operation, making it 5 asynchronous tasks per packet. Transferring a 1GB requires 3.5 million asynchronous tasks for the *Socket* module alone.

With so many task being scheduled on a system not fully optimized for asynchrony, the observed behavior of MCDTPi is now clear. The single-, dual-, and quad-channel configurations were able to operate since scheduled asynchronous tasks were more distributed. However, with each increase in channel count, the TPL library was put under strain more rapidly, negatively impacting performance. The octa-channel configuration overwhelmed the TPL library to the point of deadlock.

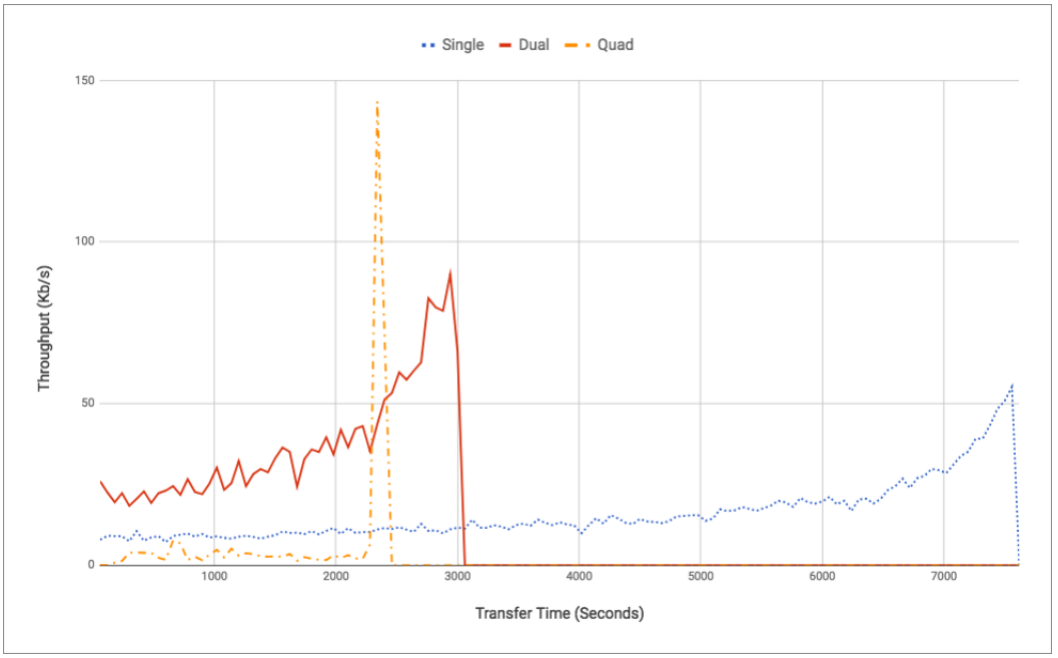

FIGURE 5.1: Client Throughput

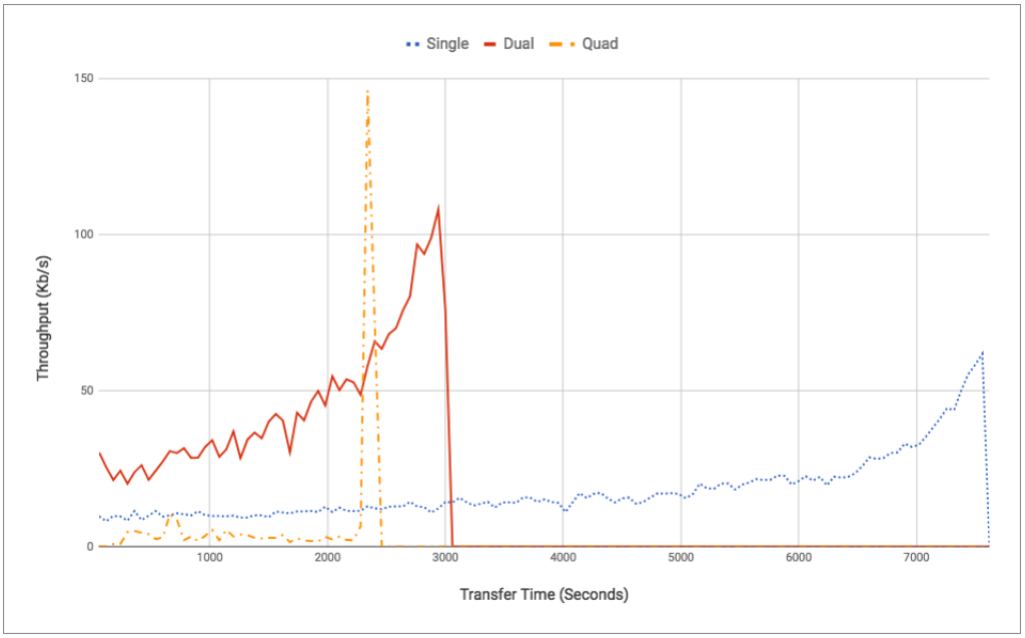

FIGURE 5.2: Server Throughput

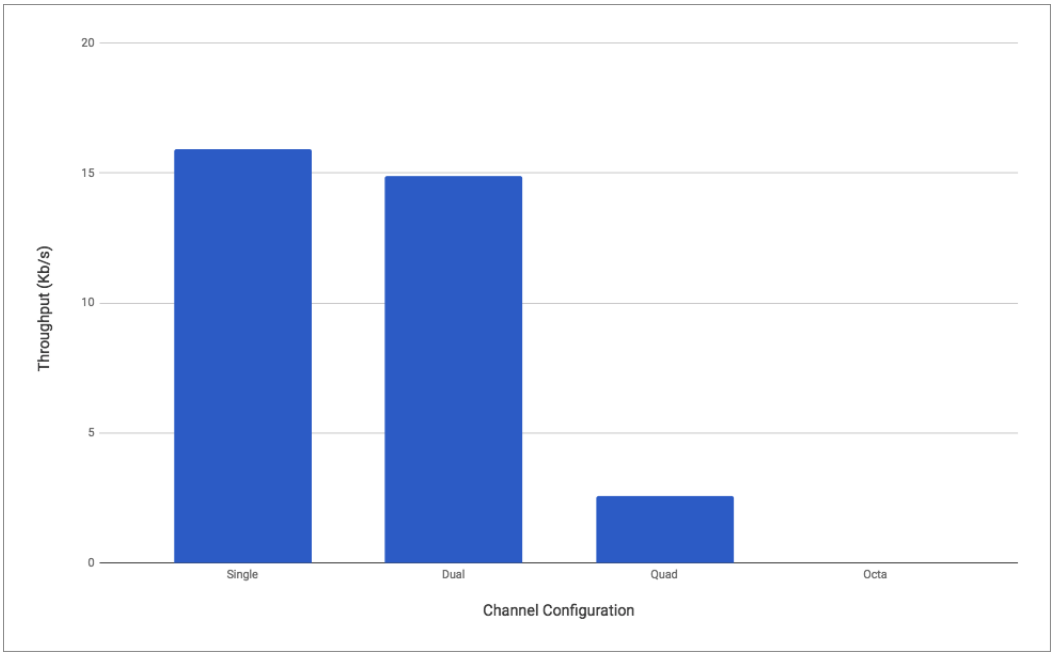

FIGURE 5.3: Average Throughput

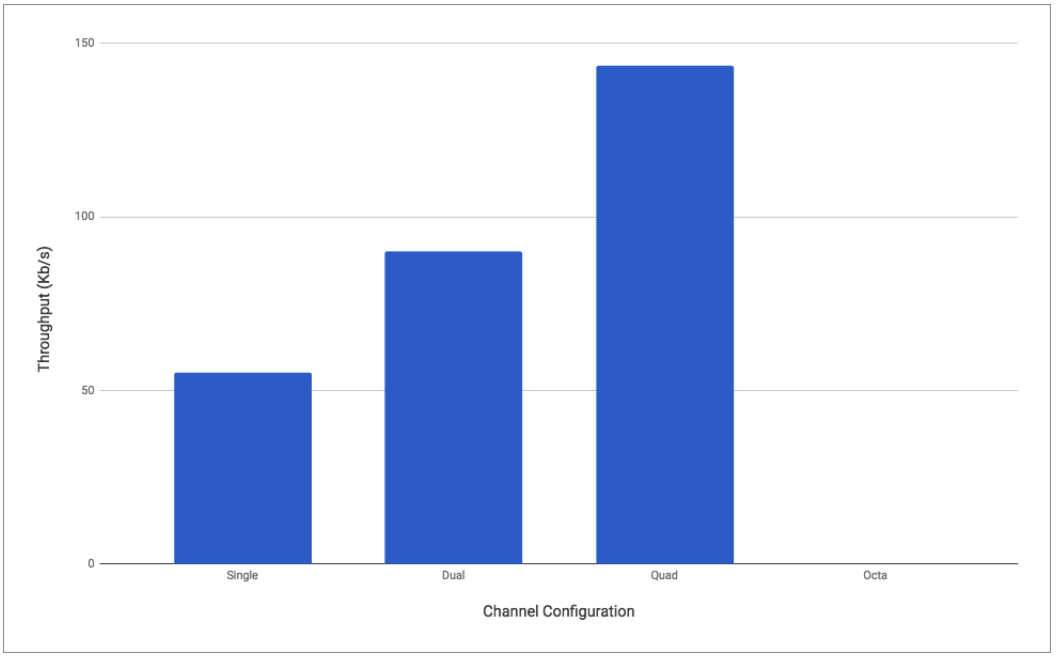

FIGURE 5.4: Max Throughput

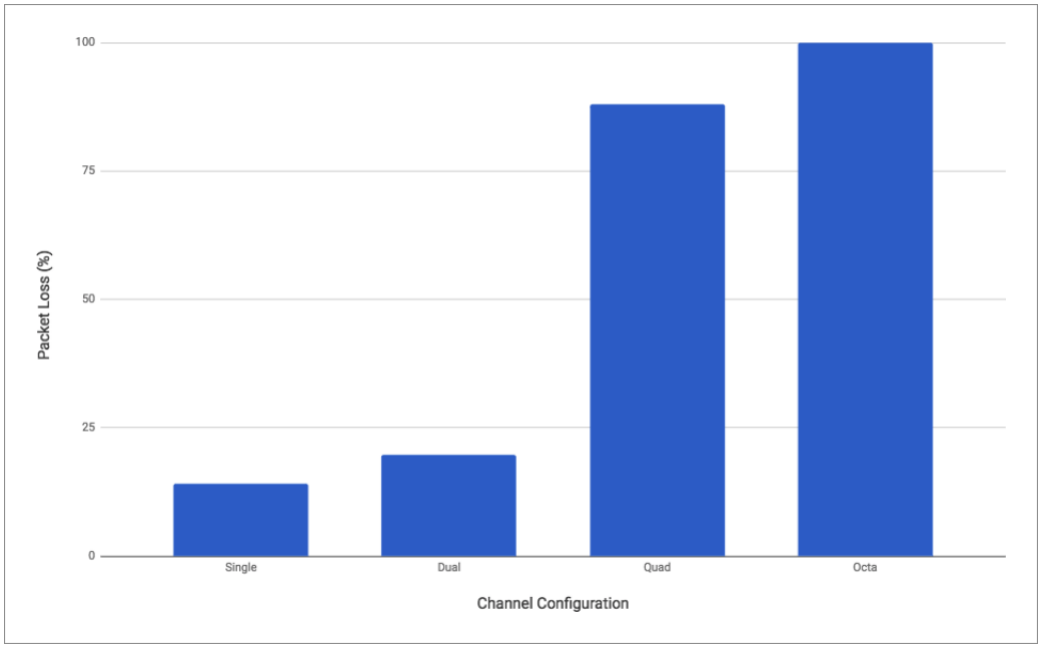

FIGURE 5.5: Packet Loss

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 stemmed more from the .NET Framework. The discovered asynchrony issue in the cross-platform variant of the .NET Framework was rather obscure. With little documentation on the matter, a significant portion of the investigation involved examining the source code of both the .NET Core Framework and the Common Language Runtime. This was the only way to confirm that I/O Completion Ports were only being used on the Win32 Kernel API.

With respect to MCDTP and MCDTPi design challenges, early iterations had challenges with respect to disk I/O operations blocking socket I/O operations. The early architecture was not modular like MCDTPi is now. Consequently, MCDTPi needed to be restructured to be modular as discussed in Section 4.2 and incorporate asynchrony at an architectural level, which is discussed in Section 4.3. Additionally, 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

As mentioned in Chapter 3.1.1.2, file selection was not included in the design of MCDTP. This feature was left out because it was outside the scope of the project and was left as future work for MCDTP and MCDTPi.

Tests have revealed that the .NET Core Framework does not use I/O Completion Ports on Unix-based operating systems. Testing IOCP is beyond the scope of this project. In order to understand how IOCP will impact performance, a separate study may need to be conducted. Since MCDTP and its implementation are open source, they can be used as candidate software to test the Windows environment. This would provide a direct comparison to Unix-based operating systems.

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 Core 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 a separate study.

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. A path worth exploring to circymvent this would be to prefragment a $64KB$ chunk of data, it would likely need to be less than $64KB$ to account for any headers added to fragments, so that when fragmentation occurred, it would not fragment the data being transferred.

Chapter 7

Conclusion

The design of MCDTP attempts to minimize overhead by keeping control messages simple and employing a phase system for structured data transfers over each channel. Asynchrony was a major component to MCDTPi both architecturally and in implementation. In efforts to provide a continuous chain of asynchronous work flows from disk I/O to socket I/O, and vice versa, MCDTPi librally utilized the asynchronous technology built into the F# language. This, coupled with the incomplete cross-platform support of the .NET Core Framework, produced behavior contrary to what was expected. As shown in this paper, using asynchrony excessively can have negative effects on performance in Computer Networking. While asynchrony has many benefits, in the context of Computer Networking, it should be used methodically.

Bibliography

[1] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–HTTP/1.1. 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] Harimath Sivakumar, Stuart Bailey, and Robert L Grossman. Psockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 37. IEEE Computer Society, 2000.

[14] Aspera. Aspera FASP High Speed Transport - A Critical Technology Comparison. page 12, 2016.

[15] M. Meiss. Tsunami: A high-speed rate-controlled protocol for file transfer. pages 1–10, 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. 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] Josephs, Mark B and Hoare, C A R and Jifeng, He. A Theory of Asynchronous Processes. pages 1–26, 1989.

[25] J. He, M. B. Josephs, and C. A. R. Hoare. A Theory of Synchrony and Asynchrony. pages 459–478, 1990.

[26] 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.

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

[28] 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.

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

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

[31] J Postel. Internet Protocol. Internet Request for Comments, September 1981.

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

[33] R. Lander. Announcing .NET Core 1.0, 2016.

[34] K. Havens. December 2016 Update for .NET Core 1.0, 2016.

[35] .NET Foundation. *.NET Core 1.0.3 Source Code*, 2016.

[36] Jeffrey Richter. *CLR via c#*. Pearson Education, 2012.