# Amateur Radio Satellite File Transfer Protocol (ARSFTP)

*Author:*
Andrew Morrison

*Advisor:*
Dr. John Bellardo

SENIOR PROJECT

California Polytechnic State University

San Luis Obispo

ELECTRICAL ENGINEERING DEPARTMENT

June 2012

# *Abstract*

During the course of a CubeSat's mission, a large amount of information is collected. It is necessary for this data to be transferred to ground stations over links characterized by low data rates, high packet corruption and drop rate, and high latency. Current file transfer protocols cannot handle these characteristics well, and delay the transfer of the file longer than acceptable. A new file transfer protocol, Amateur Radio Satellite File Transfer Protocol (ARSFTP), is designed specifically to transfer files across links mentioned previously and seeks to transfer data faster and more efficiently than current file transfer implementations.

ARSFTP was tested using a Linux utility to simulate network environments and show their effects on data transfer time. Using this, it was possible to determine whether or not ARSFTP would be suitable for satellite deployment.

# *Acknowledgements*

Firstly, I would like to thank my advisor, Dr. John Bellardo, for offering his continual support and guidance for the duration of the project, as well as amazing amounts of patience.

I would also like to thank my fellow lab mates in PolySat for their good spirits, company, and help along the way.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 CubeSats

CubeSats are nanosatellites, measuring 10cmx10cmx10cm (for a "1-U" CubeSat) and weighing at most 1.33kg. They are widely used in universities, but have more recently been adopted by government organizations and commercial entities. Their use as a platform for payloads is due to their low cost compared to larger satellites with similar capabilites. Many CubeSats communicate data across links with various characteristics detrimental to successful communication. This includes, but is not limited to, low data rates (typically sub-9600 bits per second), high packet corruption and drop rate (dependent on distance between transmitter and receiver), and high latency (also dependent on distance). In order to effectively communicate between a ground station and satellite, a more delay-tolerant transfer protocol is necessary. This protocol should transfer large files faster than existing file transfer protocols under conditions described. ARSFTP is developed for use on CubeSats running a Linux-based operating system with a UDP/IP stack. Previously, CubeSats were developed on systems that were not capable of running Linux with a UDP/IP stack, but recently advancements in technology have allowed for more storage space for flight units, resulting in the allowance for more complicated systems such as Linux.

## 1.2 File Transfer Protocols

A file transfer system allows for the basic transferring of files over a network. Protocols such as ARSFTP are built on a server-client model, where a client establishes a connection with a server, which then completes the client's request (most often GET or PUT). There are a multitude of existing protocols for transferring data, but most are not designed for poor connections and bad links. Many implementations (such as the default FTP program) are built on TCP, however TCP's congestion control algorithm is not adequate for the link problems; as packets are not acknowledged in adequate time, TCP waits an exponentially increasing amount of time before re-transmitting the packet. There do exist file transfer protocols built on UDP, but these still make assumptions on link connections and/or have more overhead than space communications can handle.

Two main methods of transferring data within the protocol are fixed length/format downlinks and block resolution downlinks. The former is useful when only certain information needs to be transmitted; the data is not dynamic and is always stored in the same number of bytes. In this form of communication, overhead is greatly reduced and throughput is increased. The latter is more useful for dynamic amounts and different types of data; i.e., files on a system. The amount of control over which block to send at certain points in time, as well as the size of the blocks, allows the client to request specific missing blocks of a file. This introduces much more overhead than fixed-length transmission, but is better for larger transfers.

# Chapter 2

# Requirements and Specifications

## 2.1 Specifications

### 2.1.1 Specification Considerations

These specifications touch upon important FTP implementation requirements, as defined in the RFC specification [6], as well as important engineering/programming practices. They are presented with a fully functional file transfer program accounting for amateur radio link characteristics in mind.

Firstly, it is important to approach the design of a program with good programming practices. To minimize impact on the system, the program should utilize dynamic memory allocation. This allows the program to use the least amount of memory necessary. Otherwise, the program may request more memory than it needs. While this may not noticably impact performance, it still allows the program to grow and shrink only as necessary, thereby being more efficient. As with any dynamic program designed to be running for long periods of time, ARSFTP must be implemented such that no memory leaks are present. The longer a program runs, the more memory leaks affect its performance; as more memory is allocated without freeing any to the operating system, the program begins to consume more resources than allowed (or than the operating system can allocate), and it crashes. Constantly consuming too much resources and crashing are not acceptable, as both cause excess strain on the hardware. As a result, the program should be tested with memory leakage analyzers such as Valgrind in order to ensure minimal impact on the operating system and flight hardware.

In addition to the absence of memory leaks, the program should be designed such that it should not wait at any one section of code. This is especially important for the server side of the application, where it is possible that multiple requests are being made. If the server stops accepting requests in order to wait for a packet (one that may never arrive), no other clients may make necessary requests. This situation is detrimental to handling mission-critical data, and so considerations should be made to ensure that the program does not halt for any reason. Similarly, the code should be modular. Because it is possible that any code may be re-used in another application, abstracted code allows for portability of the system while minimizing need to re-write code. Additionally, modularity allows for easier bug fixes – any issues may be isolated and fixed with minimal impact on the rest of the system.

At its core, a file transfer protocol should provide enough functionality to transfer a file onto a server as well as retrieve a file from a server. Especially with space communication, the transfer should complete quickly enough so enough transactions may be completed in an alloted time. The amateur radio links used in communication with CubeSats impose constraints on data rate as well as how quickly a packet may be determined to time out. The latter concern leads to problems in current file transfer protocols, which contain assumptions on the rount-trip time (RTT) of packets that are too low for space communication. Because of this, ARSFTP should allow for these much longer RTT without wrongfully assuming dropped packets, since doing so would waste available transfer time and lower overall link utilization through needless retransmission. Similarly, ARSFTP should complete transfers in noticably less time than FTP after the introduction of packet loss. This time varies by the length of file, chosen size of individual packets, and data rate of line, but the time difference should increase as file size increases.

Since it is likely over the course of a mission to require more from a FTP server than a simple transfer of files, more capabilities need to be introduced. For instance, it is useful to find a current listing of the contents of specific directories in order to locate specific files. To accomodate to this want, directory listing functionality should be implemented. This should be done efficiently so that the directory listing does not consume all of the available bandwidth. Additionaly, if an unwanted file were to be accidentally placed on a server, or a file existed that was too large and needed to be removed, it would be useful to remotely issue one command to remove the file and, as such, file removal should be implemented. Lastly, it is important to group files into categories and download a

specified category, or to prioritize files within categories. This leads to the need to implement the functionality to retrieve statistics of files within such groups, as well as downlink these files in order of priority.

These considerations are summarized in Table 2.1.

### 2.1.2 Resulting Requirements

Table 2.1: ARSFTP Requirements and Specifications

| **Marketing Requirements** | **Engineering Specifications** | **Justification** |
|---|---|---|
| 1, 2 | 1. No memory leaks | Running out of memory causes the system to crash. |
| 1, 6 | 2. Program should not hang on any one section of code | Maximizes throughput when multiple packets arrive close together. |
| 3, 5 | 3. Code should be modular | Easier to debug; portable to other applications. |
| 1, 2, 3 | 4. Dynamic memory usage | Minimize footprint of system; low overhead of program. |
| 4, 6 | 5. Correctly lists, deletes, or retrieves files from a satellite | Necessary to retrieve data and monitor files on the remote satellite. |
| 4, 6 | 6. Correctly transfers files to a satellite | Necessary to store information needed in data collection. |
| 4, 6 | 7. Correctly implements priority queueing behavior when requested | Necessary for retrieving important information and grouping files in a meaningful manner. |
| 1, 2, 4, 6 | 8. Should transfer a file faster than FTP under higher packet loss conditions | Important to retrieve data under satellite visibility window and to have the highest link utilization possible. |
| 2, 4 | 9. System should not erroneously assume dropped packets due to increased RTT | Since increased orbit distance results in increased RTT, the system should support longer RTT on packets in order to not assume a dropped packet when it is not the case; results in a higher link utilization. |
| **Marketing Requirements**<br>1. Efficient<br>2. Stable<br>3. Flexible<br>4. Ability to store, retrieve, delete, queue files<br>5. Low maintenance<br>6. High throughput | | |

# Chapter 3

# Design and Implementation

## 3.1 System Overview

The first consideration for ARSFTP is the basic implementation of the protocol; which transport layer protocol to use, and whether to use fixed format downlinks for block resolution downlinks as a method of transferring data. Since, as discussed previously, TCP is not acceptable for the link characteristics, UDP was the next choice as a well-established datagram protocol. No other protocols were considered due to UDP's lightweight datagram transfer. Since UDP has no reliable data transfer, but it is still important to receive the entire file, acknowledgement (ACK) packets were implemented to ensure reliable delivery (however, unlike TCP, no backoff is implemented). Due to the possible variable length of packets being sent for different commands, block resolution downlink was chosen and implemented as a window protocol. This allows for a faster transmission with the choice of a correct block and window size for a given link. Finally, ARSFTP is designed on a server-client model, with the server on a CubeSat and client program used on a ground station. This basic level 1 block diagram interaction is presented in Figure 3.1. Individual components are explained in Table 3.1.

The Requirements and Specifications section leads to specific commands that should be implemented: transferring a file from and to a server (PUT and GET, respectively); removing a file on a server (RM); listing the contents of a directory (LS); adding a file to a priority queue (QUEUE); reporting one or multiple priority queue contents (REPORT). The details of the design decisions of each component are presented in the following sections.

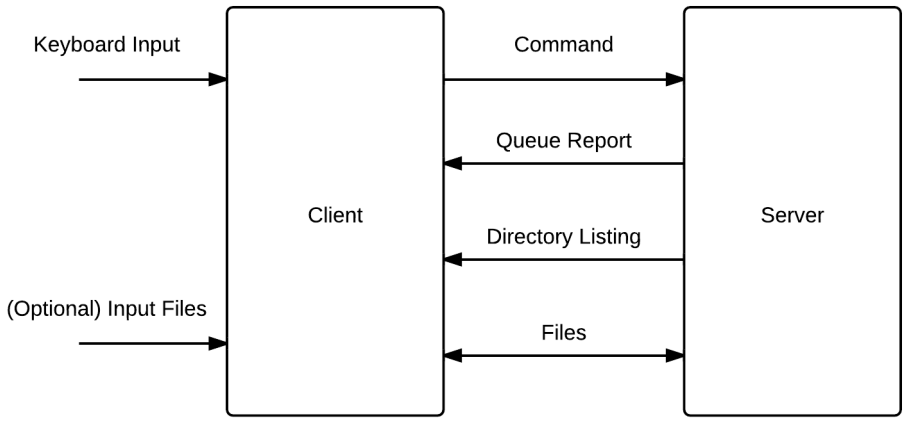


FIGURE 3.1: ARSFTP Level 1 Block/Flow Diagram.

TABLE 3.1: ARSFTP Level 1 Block Diagram Decomposition

| **Item** | **Functionality** |
|---|---|
| Keyboard Input | User input from keyboard to client. Describes which function should execute as well as defining additional parameters. |
| (Optional) Input Files | User designates which files, if any, should transmit to the server. |
| Client | Program running on a local machine (generally a ground station). Communicates with the server to execute functionality the user wants. |
| Command | Part of message from client to server describing which task the server should execute. |
| Queue Report | Priority queue listing information (described more thoroughly under QUEUE and REPORT). |
| Directory Listing | Information regarding a specific directory the user defined (described more thoroughly under LS). |
| Files | Files that are either sent to the server or received from the server. |
| Server | Program running on a remote machine (in this case a CubeSat) that response to commands sent from client. |

## 3.2 GET

### 3.2.1 Client

The client begins by sending its initial request to the server containing the GET command and destination file name it wishes to receive. The client then enters a loop waiting for data or until it times out by not receiving data for a configurable amount of time. After timing out, it re-sends its request containing the vector of ACKs. After sending a configurable number of requests, the client assumes communication has been dropped and removes the the local file. If the client loses connection with the server mid-transfer,

a state file is kept so that the transfer may resume and missing blocks of the file may be transferred. This allows a file that may be too large to downlink in a single pass to transfer over multiple passes, allowing complete transfer of large files such as picture or video. The basic GET procedure for the client is summarized in Figure 3.2.

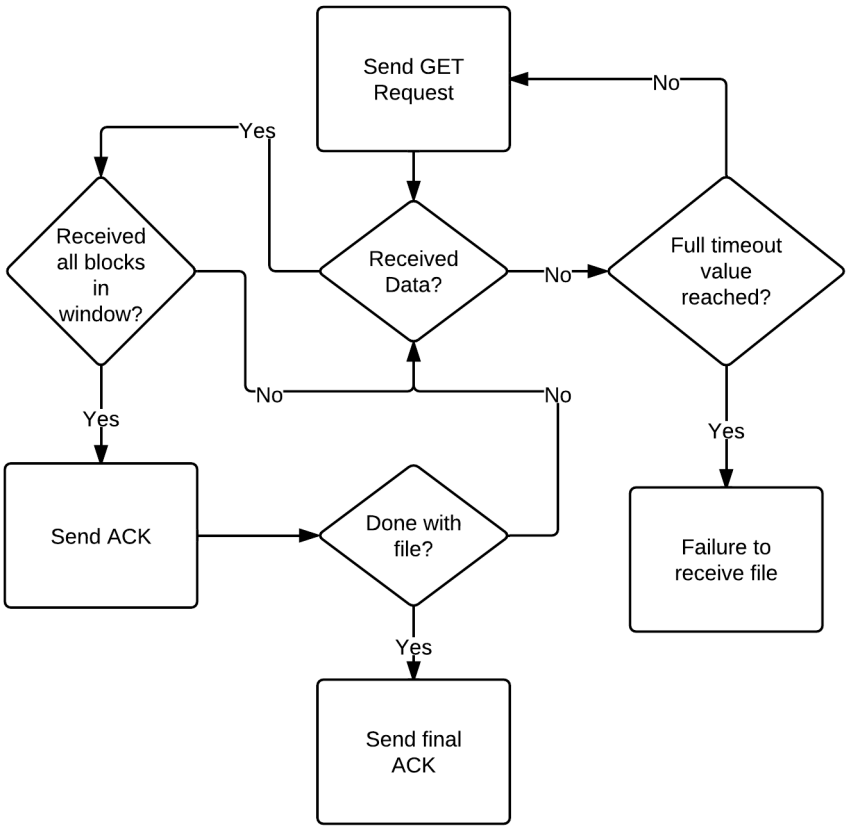


FIGURE 3.2: Client GET Flowchart.

### 3.2.2 Server

The server utilizes functions from the PolySat library in order to allocate time to different clients; the client address is registered with the server and the server passively waits in an event loop for any client to send it data that it will process before sending more data back to the client. The event loop was chosen over other forms of parallel programming due to UDP being a connectionless protocol, as well as the lowest amount of overhead in event-driven programming as opposed to threads (which would spawn too many processes to manage). This process of registering a client with the server and the basic event loop is consistent across all functions.

After registering with a client, the server verifies that the file exists – if it does not, it responds with a failure. If it can, it sends metadata of the file back to the client, and

then waits until the client responds before sending windows of the file. As the client ACKs vectors that it receives, the server tracks which blocks have been received and slides the window if enough blocks have been ACKed. Otherwise, the server will send the missing blocks until the client receives all of the blocks in a vector. A simplified version of this flow is shown in Figure 3.3, and is also used in PUT, LS, and REPORT, as described in their respective sections.

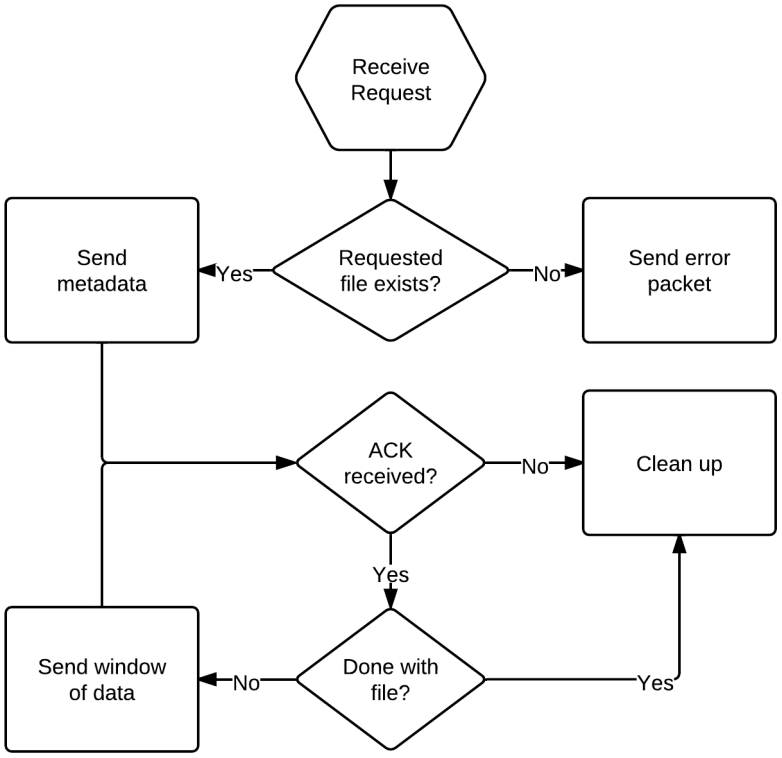


FIGURE 3.3: Server GET Flowchart.

## 3.3 PUT

### 3.3.1 Client

To initiate the PUT request, the client sends the PUT request containing the remote file name to be created. If this is acknowledged with a successful packet, the client initiates transmission of the file. ARSFTP is written in a modular way such that functions are able to be used in multiple subsystems. This is especially the case with GET and PUT. The client side of PUT functions similarly to the server side of GET; the client sends windows of missing blocks and waits for enough continual blocks to be ACKed, and resends windows as necessary. It utilizes the same timeout mechanism from GET to

determine if it needs to resend a window. This flow is similar to Figure 3.3, but includes a timeout mechanism for re-sending the window as mentioned above.

### 3.3.2 Server

Similar to the client, the server utilizes code that the client uses for GET, but still only passively receives portions of the file so it does not hang waiting for more blocks. At the beginning of the transaction, if it cannot create the file (e.g., because it does not have permission to) it sends a failure packet so the client does not waste link time sending blocks. The server does not re-send ACKs if no more data is received, but instead waits for the client to "time out" and re-send the window. The flow is similar to Figure 3.2, but does not include the timeout mechanism of the client, because doing so would cause a block in the program, which does not comply to the system requirements.

## 3.4 RM

### 3.4.1 Client

RM is the simplest function of ARSFTP for both the server and client. The client only sends a RM request packet with the name of the remote file name to delete. Whether or not this is a successful removal does not change what the client does, since it is doubtful if the removal was successful that a re-transmission would solve the problem. No authentication is added to the RM request; the reason and consequences of this are discussed in the following section.

### 3.4.2 Server

Once the server receives an RM request, it attemps to remove the file denoted by the file name field. This is in no way a secure transaction, and as a result is not the best decision for mission critical files. However, adding authentication introduces high amounts of overhead, which might not be the best in a limited transmission time situation. In the future, some form of authentication should be added so that the server does not accept RM requests from anybody, who might mailciously attempt to remove important files from the CubeSat.

## 3.5 LS

### 3.5.1 Client

The client begins the LS request by requesting a remote directory to find the contents of. To reduce overhead and to reduce the number of requests made by the client, recursive and hidden flags are implemented to view subdirectories as well as hidden files, if necessary. A major concern of transmitting this data is amount of data transferred due to the amount of files or subdirectories located on the server. To accomplish lowering the amount of data transferred (to reduce the amount of time necessary to receive this information), the directory listing results are compressed into a file, which is then transferred to the client. To compress and decompress the data, the zlib compression library is used due to its efficiency in compression as well as documentation for its use. The compressed file is received by the client in the same way that GET receives a file (see above). Once received, it is decompressed by an additional program that outputs the decompressed data to stdout. The decompression procedure is available on zlib's website. The idea behind this scheme was to allow a script to use the output to locate files a ground station wishes to download, or to download an entire directory.

### 3.5.2 Server

The server receives the client's LS request, and if it cannot access the directory, it responds with an error packet. Otherwise, it creates a directory listing file, compressing each file entry into the file as necessary. The method of creating a compressed file and adding entries to be compressed is available on zlib's website. The location of the created directory listing is in /tmp, so that the additional files do not clutter the directory the server is running in and are easily pruned from the system. If the recursive flag is set, LS will recursively enter each directory, adding the statistics for each of those files; likewise, if the hidden flag is set, LS includes all "hidden" files it finds. After the compressed file is completely finished, it sends the file's metadata to the client and the rest of the transaction uses the GET functions to transfer the file. The re-use of code allows for any further optimizations in transferring files to also be applied to LS, thereby increasing efficiency and conforming to the requirements.

## 3.6 QUEUE

### 3.6.1 Client

The QUEUE command is different from other commands in that the client is not located on a ground station; it is instead located on the CubeSat and used by other processes to add files to the appropriate priority queue. QUEUE uses IPC (and on a different port than other ARSFTP connections) to request that a file be added to to a specified queue and priority. Each queue corresponds to a different system and different processes use different queues to add their data. For example, sensor data may add its information to one queue, while satellite health may add its file to a different queue. After some time, the health system decides that it has information that is critical information (some part of the satellite crashed, for instance) that needs to be downlinked first, so it inserts the data at highest priority. This may be a problem if the application constantly attempts to add files at highest priority, but it is the application developer's responsibility to ensure this does not affect its intended operation. There is an optional remove argument that allows an application to remove a file specified by name and group.

### 3.6.2 Server

When a process wishes to add a file to the priority queues, the server receives the QUEUE request and attempts to insert it into the corresponding queue. If the file exists in the queue currently, it is removed from the queue and replaced at the desired priority. The insertion of the file into the queue is done in a common first in first out priority queue fashion, where files of highest priority at the beginning of the queue, with equal priority files being placed in chronological order (older insertions first). If the queues become too full (i.e., the number of files in a queue surpasses a configurable maximum per queue or the sum of the sizes of the files surpasses a configurable maximum), files are dropped in order of lowest priority. The server does not send confirmation that the file was successfully added to the queue, since the only reason the file does not successfully enter the queue is the case that it is too full and the file is too low of a priority, or the file statistics are unobtainable.

Since this queue data is vital for the ground station to retrieve groups of data, the queue data must be persistent. If the software or ARSFTP program must reboot, it is important that there be some method to recover the current items in the queue and

maintain the ability to report to the ground station files that were previously in the queue. The three main methods of persistence files that were considered were: SQL database, compressed file, and plain text file. The first method, using an SQL database, was discarded in interest of time and SQL's heavy I/O, which may be a much larger performance impact on the CubeSat than is necessary for the queue persistence (since it is not likely that there will be enough files in the queues to warrant a database. The second method, zlib compressed file, is promising until zlib's memory utilization is examined – if not closed properly, zlib allocates too much memory to the one process. In a process (such as ARSFTP) that runs for extended periods of time, this is a potential massive memory leak, which is not acceptable according to the requirements. When the zlib compression stream is closed, it adds its end-of-stream delimeter, which is not acceptable when the persistent file needs to be open and files need to be constantly added.

To keep the queue data consistent, the plain text file method was chosen. When a new file is added to any queue, its information (file name, group, priority, flags) is added to the queue file. This file does not need to be kept in any priority or group order, since the initialization uses the functions used to insert a file into any queue; those functions handle insertion of files in any order. The highest cost of using this persistence system is in removing a file; to stay constantly persistent (with the least user interaction), it was decided that the persistence file would be updated completely after each removal. To accomplish this updating, the file is completely truncated and replaced with the current queue data. Even though this is not the most efficient method of updating the file (for instance, it is inefficient if the file to remove is the last of the file), it is simpler and requires less I/O than using a temporary file to copy partial contents to effectively remove the file from the queue.

## 3.7 REPORT

### 3.7.1 Client

Reporting contents of a certain queue is important for ground station operations to receive groups of files related to a current mission or system health information. The client may request the contents of some (up to a configurable maximum) or all queues. In addition, for each queue it wishes to downlink, the client may request a maximum

size (in bytes) of files to have in the report. If no maximum file size is specified (i.e., "0" is the input for the maximum size argument), then the server assumes maximum integer size (UINT32_MAX). If there is no argument for which queues to downlink, the server assumes all queues and maximum integer size for each queue. To initiate the REPORT request, the client stores the number of groups that exist in the request and, if specific groups are requested, the request packet is of variable length containing each group number and maximum file size for each queue. The report format is stored in the same format as LS, with the same compression library. The results are stored in numerical group order and files within groups in order of priority. This allows to the LS and REPORT mechanisms to be easily maintained and fixed as necessary. REPORT uses the same program flow as GET to receive the report file, and the same program as LS to decompress and read the queue data. The goal of report is to use a script to request specific queues and then use REPORT's output to request each file in the queue using GET.

### 3.7.2 Server

The server receives the queue request and creates a report file to store compressed queue data. REPORT uses the same compression tools and functions as LS to create the data file in /tmp. To determine which files to add to the data file, REPORT uses the limits sent by the client to loop through the queues and add specified groups up to the maximum total file size. If no limits are included in the request packet, REPORT assumes all queues and UINT32_MAX as the maximum total file size, and compresses all of them. After the compressed file is created, REPORT utilizes the GET functionality to transmit data to the client.

# Chapter 4

# Data and Analysis

## 4.1 Transfer Times

The portion of ARSFTP with the most impact is GET functionality – other functions make use of its transferring functions to send their respective data. This section is concerned with the impacts of link variables on ARSFTP. tc was used to simulate link characteristics; drop rate, link delay, and bandwidth. Unfortunately, the bandwidth setting did not correctly limit the traffic to a constant 9600 baud (a typical rate for satellite data communication). Instead, it allowed for a large burst of traffic before restricting sending of any data at all, in order to "simulate" 9600 baud. Because of this, current FTP implementations were able to burst 2MB of traffic before being limited, and are therefore not included as a comparison in this analysis.

Since other functions are dependent on GET, and any other computations and functions server-side are minimal (¡ 1 second for even large amounts of files for LS), transfer performance is the only characteristic analyzed.

### 4.1.1 Window Size

One of the major factors in transfer time is the window size. This determines the number of blocks that are sent at a time. Too large of a window increases overhead above the amount acceptable for satellite transfer. Too low of a window slows the transfer due to increased waiting time on ACKs. Figure 4.1 depicts the effects of window size on transfer performance. This test is with 100ms latency, 10% drop rate, and a 100KB file.

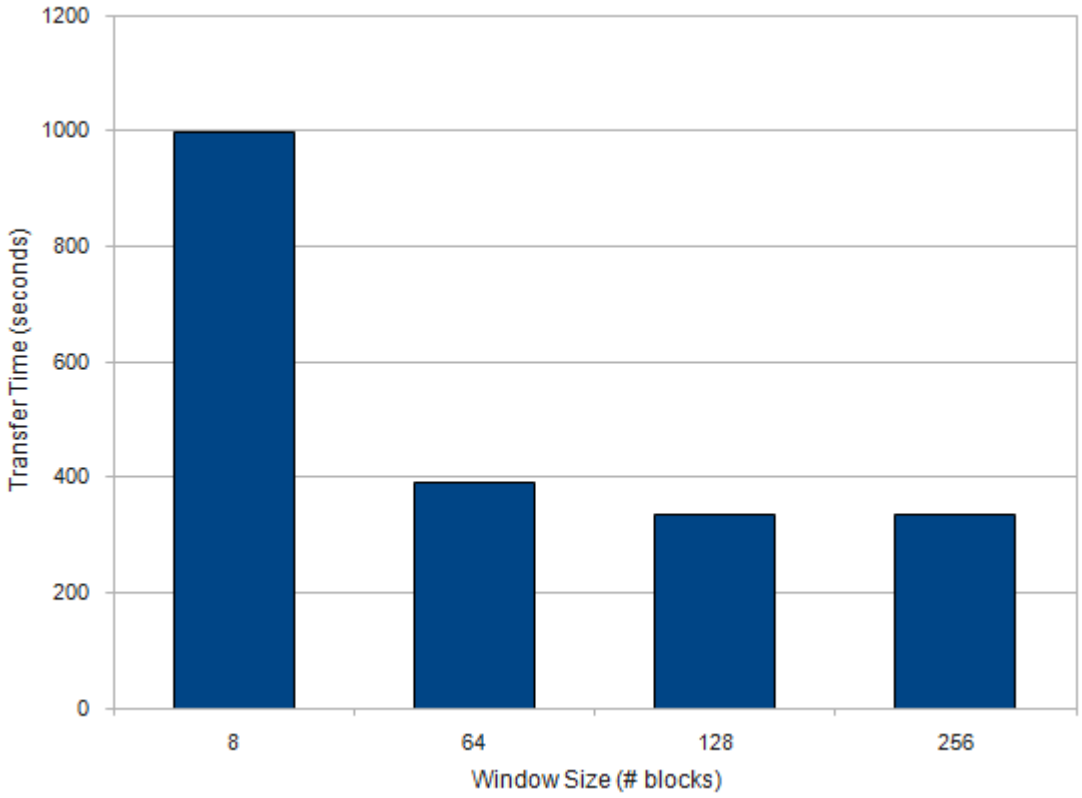


FIGURE 4.1: Effects of window size on transfer time. (100KB file)

The impact of the window size is clearly seen when the window is very small (around 8 blocks). The transfer time is much greater than the other sizes due to increased processing necessary. At 128 blocks per window, the transfer time due to larger window sizes becomes negligible, and a window size of around 128 may be best in this environment due to lower amounts of data on the line at once, as well as better for handling packet drops.

### 4.1.2 File Size

This test was to determine if an increased file size impacted transfer time above the expected linear increase. The expected trend is a linear transfer time increase with respect to file size. This test is conducted with 100ms latency, 10% drop rate, and window size of 256 blocks.

As expected, transfer time increases linearly with file size. With other factors remaining constant, file size does not impact transfer time more than expected. In a normal 15-20 minute transfer window, according to this test, a 250KB file may be transferred.

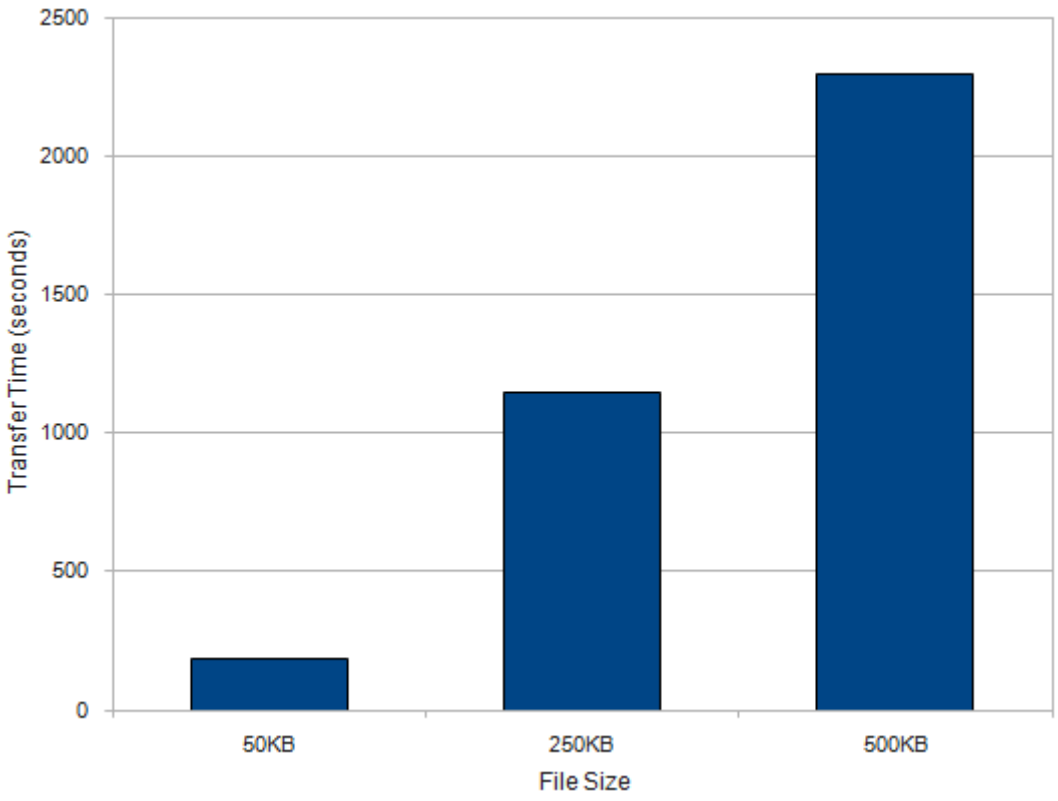


FIGURE 4.2: Effects of file size on transfer time.

This may be different in practice, however, since the tc utility did not limit bandwidth correctly, as mentioned in the introduction to this section.

### 4.1.3 Drop Rate

Drop rate should have a substantial impact on transfer time – at such a low data rate, packet drops require re-transmission of data, therefore using more link time. At higher drop rates, more re-transmissions are required, and the effect of window size is more substantial. This test shows the effects of drop rate on two different file sizes (50KB and 250KB), with 100ms latency, and window size of 256 blocks.

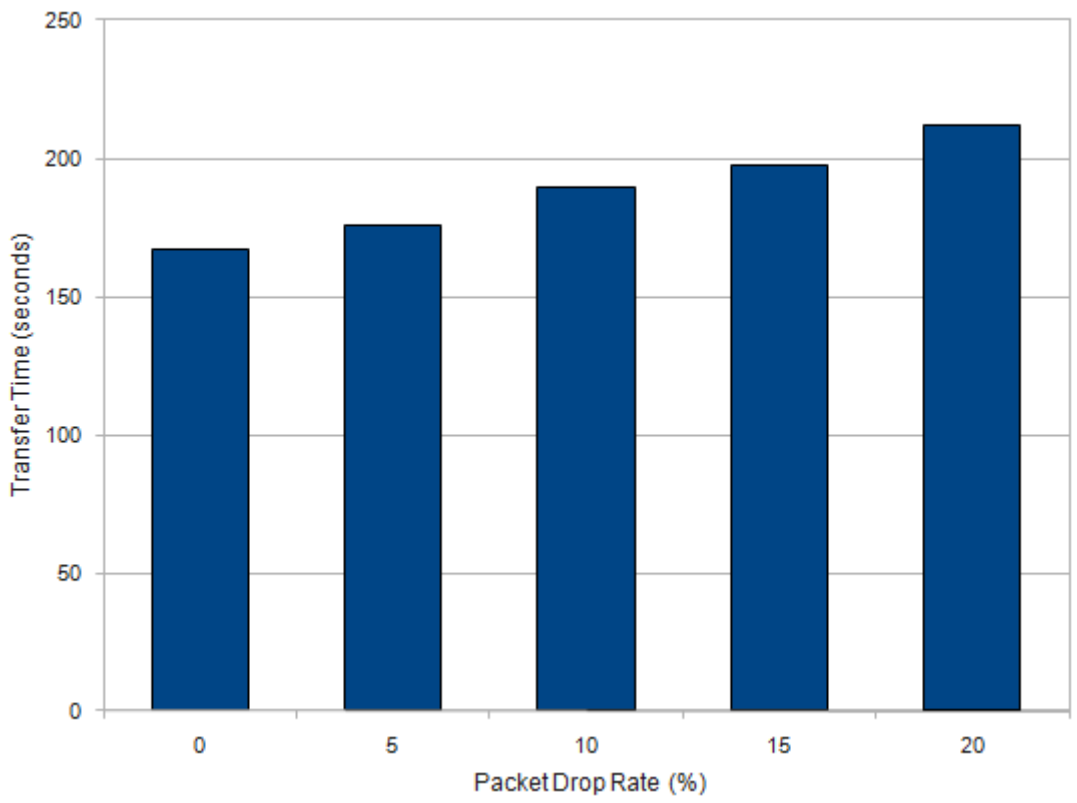


FIGURE 4.3: Effects of drop rate on transfer time. (50KB File)

With a 50KB file, transfer time increases with drop rate. The time increased by about 33

The data for the 250KB file is strange – there seems to be no correlation between drop rate and transfer time. This may be due to the tc utility not implementing drops and bandwidth in a certain order, therefore allowing packets to drop while not limiting that bandwidth, and having a random effect on the total transfer time, which is shown more clearly when a larger file size is transferred.

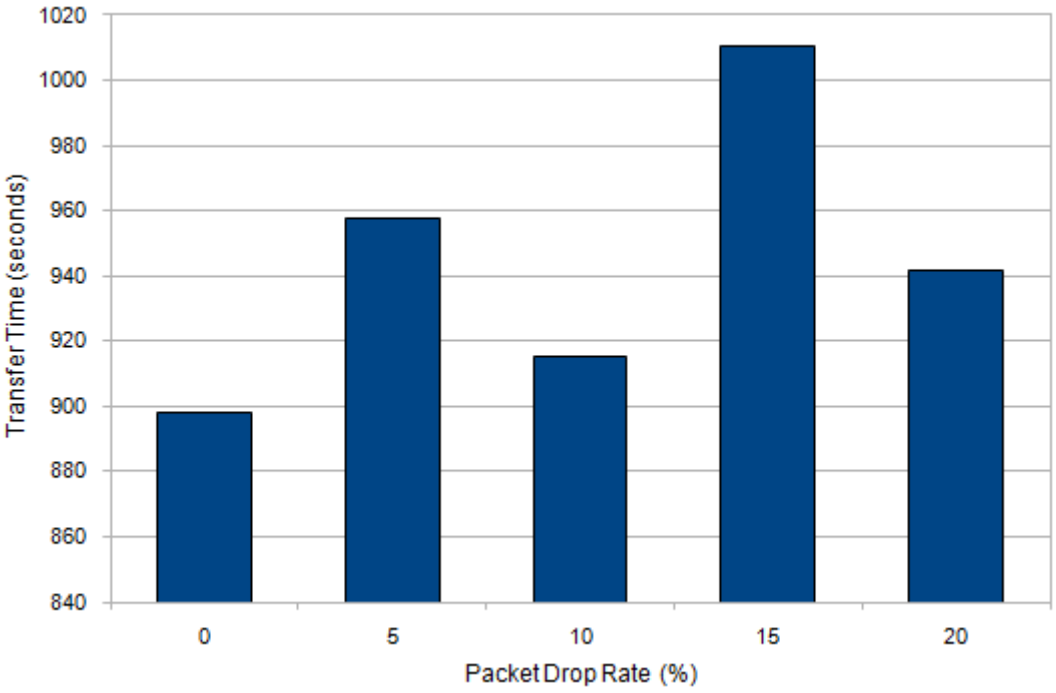


FIGURE 4.4: Effects of drop rate on transfer time. (250KB File)

# Chapter 5

# Conclusions

## 5.1 Possible Design Changes

In any system design, there are trade-offs necessary to allow for different functionality, ease of use, or efficiency. These decisions can greatly effect the program flow later in the design, which might not result in the most effective program.

If I were to begin the senior project anew, I would firstly make an effort to fully understand the existing code before developing anything new. Not doing so resulted in wasted time and code that did not conform to specifications (namely, in modularity). It would also have allowed me to clean up any errors that previously existed without also debugging my code.

As discussed in Chapter 3, the persistence of the priority queues was subject to a lot of thought, but the result could have been better. For instance, the type of storage file could be compressed so that it takes less space. Since space was deemed to not be much of an issue, this was not a primary concern. The file itself should be updated at a different rate than it currently is, however. Since the file is updated every time a file is removed from a queue (which also happens if a file's priority is updated), the file causes unnecessary churn on the storage device. This could be fixed with an update flag in removal packets or timed updates.

Another major change I would have made was to allow all parameters to easily be changed external to the program. Currently, most parameters (block and window size, for instance) are only configurable through #define statements. It is possible that this

may be changed eventually so that a user may easily change these parameters from a ground station, but it currently is not.

## 5.2 Future Work

The Design Change section leads to some future work that could be implemented. Most notably, future work may include parameter change without needing to re-boot the server program. This faster change in parameters allows for more time to transfer data and less time configuring the server. Similarly, in the future it would benefit users if there were a more user-friendly interface to send requests. Since many users of the program may not have strong backgrounds in computers, a more friendly interface than command line arguments would be helpful.

More future work includes optimization of the priority queue past what exists currently; mostly in the persistence of disk. This increases the efficiency of the queueing system and consumes less resources on the CubeSat.

Another interesting improvement is adjusting transmission parameters automatically to maximize efficiency. This would only be used on sufficiently large files where this matters, but the optimization could potentially increase amount of data transferred greatly. The server would analyze how many blocks were dropped and adjust the window size accordingly to not overload the link.

## 5.3 Conclusion

ARSFTP is able to handle latency and drop rate well. Unfortunately, the bandwidth is not able to be tested, and that is a large component of whether or not ARSFTP is better than current FTP implementations. Due to the bandwidth limiter not behaving correctly, FTP is able to burst enough traffic to get relatively large files (anything less than 2MB) before the traffic is throttled. Also, the bandwidth throttling had odd effects on transfers when packets are dropped (as shown with the 250KB file). If there is a file that is too large to be transferred over one pass, ARSFTP makes it possible to resumee file transfer on a future pass, allowing very large (e.g., picture) files to be transferred, a necessary optimization for satellite transfer. Once a program to simulate satellite

communication is fixed and ready to use, ARSFTP may be tested more thoroughly to determine its usefuleness in CubeSat file transfer.

# Appendix A

# ABET Senior Project Analysis

**Project Title:** Amateur Radio Satellite File Transfer Protocol

**Students Name:** Andrew Morrison

**Students Signature:**

**Advisors Name:** Dr. John Bellardo

**Advisors Initials:** **Date:**

### 1. Summary of Functional Requirements

The project provides an implementation of a file transfer system for cube satellites. The program behaves as a normal implementation would; i.e., transferring, listing, and receiving files.

### 2. Primary Constraints

The largest constraint in any software project is time. With increased complexity of the project comes much more increased development time, both in planning and finding errors in code. Also, since others develop programs on the same hardware, hardware availability becomes another limiting factor that is, time becomes even more limited due to sharing testing equipment and hardware. Another constraint includes the ability to send and receive packets through the system at certain points during the day. At the edges of visibility, packet drop rate increases greatly, and the ability to conduct testing or data transfer decreases.

### 3. Economic

The economic impacts lie solely in the human capital invested during software development. Afterwards, there will be more human capital involved with incorporating the software into the other systems. There are no physical parts to purchase, but human capital should cost $4800 (see table A.1). Since the project is for scientific pursuits, there should be no profits. The project should take no longer than 8 months to complete, and may be maintained from the ground after satellite launch.

**4. If manufactured on a commercial basis:**

Commercial production of this senior project will not occur. Would production occur, the main difficulty would be in incorporating the system into other satellites, and changing the code so that the file transfer system continues to operate smoothly on another satellite.

**5. Environmental**

Since the project entirely resides in software, there are no direct environmental impacts; there are no parts to acquire that would impact earths resources, and there are no ecosystems that face destruction as a result of the software. However, the satellites purpose may be one to help the environment (through data or other means), and in that way this system will benefit the environment.

**6. Manufacturability**

A lack of physical parts limits this products manufacturability. In terms of mass production, copying files is not difficult. The only possible issue would be distribution to a customer (distribution via CDs or other means), and in the setup of the file transfer system on a customers product.

**7. Sustainability**

Issues exist regarding maintaining and updating the system when communication to the satellite is only possible during certain periods of a day. If the communication system is updating and visibility is lost, there may not be a way to communicate with the satellite thereafter. The project itself does not promote sustainable resource use, but other portions of the overall satellite may be efficiently engineered in order to maximize sustainability. Increasing throughput and packet success rate improves the system greatly. Another possible improvement is including more commands that the system supports as needed.

### 8. Ethical

Since the software exists on satellites and may transfer files for use within government, it is important to consider security of the system. The system must be safe to prevent data interception, as in doing so somebody may see information that is important to the success of a mission. Without proper use, files may transmit that could compromise the mission and/or the satellite. Since this is an important consideration, security of the system should be strong.

### 9. Health and Safety

The main concern with the design and manufacturing of this device is carpal tunnel while writing software. It will be important to take frequent breaks so as not to develop the condition.

### 10. Social and Political

Since the project is related to space and for a private company, there are various procedures that must be followed in order to conform to sets of rules necessary for operations in space. The project impacts PolySat, as well as other agencies and companies associated with the mission. The stakeholders benefit from being able to transfer necessary files to and from the satellite system in order to analyze data. Failure or success directly reflects PolySat and customers, and the projects success impacts the availability of future missions to be developed by PolySat.

### 11. Development

Multiple developmental tools were used in the development of this project. The network analysis tool "tc" is useful in emulating network conditions, namely latency, packet drop, and data rate. Valgrind was introduced to locate memory leaks and potential segmentation faults. The articles researched and used can be found in the bibliography.

Table A.1: Estimated Costs

| **Item** | **Cost** | **Justification** |
|---|---|---|
| Labor | \$4800 | $((4\frac{hours}{week} \times 10weeks) + (10\frac{hours}{week} \times 20weeks)) \times \frac{\$20}{hour}$ |

# Bibliography


[1] AXSEM, "Ax5042 data sheet and programming manual," *Axsem*. [Online]. Available: http://www.axsem.com/

[2] B. Elbert, *Introduction to Satellite Communication*. Boston, MA: Artech House, 1999.

[3] C. Y. et al., "Evaluation of tcp and internet traffic via low earth orbit satellites," *IEEE*, 2001. [Online]. Available: http://www.ieee.org

[4] W. C. et al., "Design of a secure ftp system," *IEEE*, 2010. [Online]. Available: http://www.ieee.org

[5] C. L. Greg E., "Amateur radio satellite file transfer protocol," California Polytechnic State University, San Luis Obispo, Tech. Rep., 2011.

[6] B. Lathi and Z. Ding, *Modern Digital and Analog Communication Systems*. New York, NY: Oxford University Press, 2009.

[7] J. Postel and J. Reynolds, "File transfer protocol (rfc 959)," 1985. [Online]. Available: http://www.ietf.org/rfc/

[8] R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1994.

[9] Wikipedia, "Ftp (wikipedia)." [Online]. Available: http://en.wikipedia.org/wiki/FTP