Amateur Radio Satellite File Transfer Protocol (ARSFTP)

Greg Eddington and Connor Lange
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA
{geddingt,rclange}@calpoly.edu

***Abstract*—With the recent advances in CubeSat technology, an increasing number of ambitious missions are being planned. Many of these missions will collect a large amount of data and will need to transmit it back to the ground. In order to retrieve the collected data, ground station operators must receive data over amateur radio links which impose several limiting characteristics, including slow baud rates, high drop and corruption rates, high latency, and half-duplex communication. To make matters worse, current file transfer protocols are not designed to handle these limitations. To rectify this issue, we propose a new file transfer protocol named the Amateur Radio Satellite File Transfer Protocol (ARSFTP). ARSFTP is a protocol specifically designed to overcome the limiting characteristics of satellite communications over amateur radio.**

ARSFTP allows for the efficient transfer of files over amateur radio with a link utilization of up to 71%, compared to the 3% link utilization of existing file transfer protocols over satellite amateur radio. In order to test our protocol, a simulator was constructed which induces user specified drop rates, error rates, latency, baud rates, and half-duplex communication. With ARSFTP, we were able to transfer a 750kB file from one computer, through the simulator server, to another computer in under 15 minutes (within the duration of a low Earth orbit satellite pass), using packet shaping to simulate realistic characteristics of nanosatellite communications.

***Index Terms*—Amateur radio, CubeSat, File transfer protocol, Nanosatellites**

I. Introduction

CubeSats are small 10cm X 10cm X 10cm nanosatellites with a mass of no more than 1.33kg and have become widely adopted as a viable vehicle for satellite payloads by government agencies [**?**], educational institutes [**?**], and commercial industry [**?**]. Because of the cheap cost of launching nanosatellites compared to larger satellites, CubeSats have become the target for a number of payloads which aim to collect relatively large amounts of data. Most CubeSats communicate via amateur radio bands which impose limitations that reduce the amount of data which can be collected from payloads onboard these satellites. These limitations include baud rates of 1200 to 9600 bits per second, high packet drop and corruption rates, high latency, and half-duplex communication. An efficient file transfer protocol designed specifically for satellite communications over amateur radio is necessary to overcome these limitations and to increase the amount of data which can be collected from payloads on future CubeSat missions.

The ability to upload files to CubeSats in space is another reason a file transfer protocol designed for nanosatellites is important for future missions. Being able to send files provides satellite operators the ability to fix software bugs, update or add software with new features, and create additional tests for payloads.

The `arsftp` and `arsftpd` programs were designed for CubeSats running Linux with a UDP/IP protocol stack. Specifically, they were designed for the new bus software architecture created by PolySat, a group of engineers whom design and operate CubeSats. While in the past many CubeSats were not capable of running Linux with a UDP/IP stack, the increase in the capabilities of embedded hardware has allowed for more CubeSats to run this type of architecture. Examples of such an architecture include PolySats new bus [**?**] and QuakeSat [**?**].

A. Paper Layout

The structure of the rest of the paper is as follows: Section 2 describes previous file transfer protocols, Section 3 explains the design of the Amateur Radio Satellite Communication Simulator, Section 4 discusses the implementation of the Amateur Radio Satellite Communication Simulator, Section 5 explains the design of ARSFTP, Section 6 discusses the implementation of ARSFTP, Section 7 analyzes the performance of ARSFTP, Section 8 suggests future work, and Section 9 concludes.

II. Related Work

A. CubeSat Data Transfer Protocols

Although it is difficult to provide a survey of all data transfer protocols on existing CubeSats due to the fact that most of these designs arent published, the characteristics of the protocols we have come in contact with are outlined here. Two common forms of data transfer protocols between ground station and satellite are fixed format transfers and block resolution downlinks.

Fixed length downlinks are a common form of sending CubeSat payload and satellite data, and involve sending data in a fixed format and length defined and hard-coded into the program sending and receiving data. For example, the satellite may send battery information in a form that puts the voltage in the first two bytes and the temperature in the second two

bytes. This form of communication is simple, but is limited to only a small amount of fixed formatted data.

Block resolution downlinks are often used when sending picture and file data, and involve a client requesting a specific file block and the server sending that block. This provides fine grained control of which packets are sent which is important for requesting packets which have been dropped. The major limitation of this method is that it induces a huge cost due to the low utilization resulting from the need to send a request for every block over a high latency, half duplex link.

B. File Transfer Protocols

Many file transfer protocols exist, but are often designed with link assumptions that are not true when using an amateur radio link for satellite communications. The two protocols used in comparison to ARSFTP in this paper are FTP (File Transfer Protocol), a well adopted TCP based file transfer protocol, and ATFTP (Advanced Trivial File Transfer Protocol), a UDP based file transfer protocol designed for simplicity and low overhead.

III. Simulator Design

In order to test ARSFTP, a simulator was developed which imposes the limitations of satellite communications over amateur radio on all packets. This simulator was added on top of a previously existing satellite communication architecture shown in Figure **??**.a. In this architecture, all IP packets destined to the satellite are sent to a ground station server (GSS). The GSS performs a set of functions (such as compression and packet fragmentation) to prepare the packet for the satellite, and then sends the packet to the satellite through the radio if the satellite is in space, or Ethernet if the satellite is on the ground. Likewise, the GSS intercepts all packets being sent from the satellite and forwards them to their respective destination after performing a set of functions (i.e. decompression). Our simulated packet shaping was added to the Ethernet communication path, as shown in Figure **??**.b.

Fig. 1. (a) The satellite communication path. Packets sent to and from the satellite pass through the ground station server. (b) The satellite communication path with satellite amateur radio packet shaping added to the Ethernet path.

The simulator introduced several limitations of satellite communications over amateur radio on packets being sent to and from the satellite over Ethernet. These limitations include the following: the baud delay induced by radio hardware, latency caused by the time it takes for a packet to travel between the ground station and the satellite, packet corruption and packet drops which are frequent in space communications, packet size limitations of radio link layer protocols, and the half-duplex limitations of radio hardware.

IV. Simulator Implementation

The simulator was implemented as a set of optional packet shaping callbacks written in C++ which were added to the ground station server (GSS). All packets to and from the satellite pass through the GSS, where they pass through the simulation packet shaping functions.

All packets passing through the GSS were placed in a queue. The attributes recorded for each packet placed in the queue are shown in Table **??**. Packet shaping is performed by manipulating packets in this queue. A timed event was added to the GSS which fires when the packet at the head of the queue is ready to be sent. The packet is then removed from the queue, sent to its target, and the timed event is again added for the new head of the queue.

TABLE I
Simulation Shaped Packet Attributes

Parameter	Function
`Data`	The data contained in the packet.
`Size`	The number of data bytes the packet contains.
`Direction`	The direction the packet is being sent (sent or received).
`Start Time`	The time the packet will start being transmitted.
`Baud Delay`	The delay on the packet caused by hardware baud rates.
`Latency Delay`	The delay on the packet caused by trip latency.

A list of parameters which control the simulation packet shaping are shown in Table **??**. Before being handled by our packet shaping code, the ground station server will fragment the IP packet according to the user defined MTU of the radio link layer. A packet being added to the buffer is first shaped with hardware delays by adding a baud delay equal the size of the packet divided by the parameter `BAUD_RATE`. A delay for the latency to travel between the satellite and ground station is added as a constant value equal to the parameter `LATENCY_LENGTH`. Equation **??** shows the total packet delay incurred by baud rate and latency, where n is the size of the packet in bits.

$$Delay = \frac{n}{BAUD\_RATE} + LATENCY\_LENGTH \quad (1)$$

Because packets being sent in the same direction have to wait for hardware delays, if the packet that is at the tail of the queue is in the same direction as the packet being added, the start time of the packet is set to the greater value of the following: the time the packet was received by the GSS, or the time the baud delay of the tail packet of the queue is finished.

To simulate half duplex, two algorithms are used depending on whether the `PARTITION_HALF_DUPLEX` or `DROP_HALF_DUPLEX` flags are set. If the `PARTITION_HALF_DUPLEX` flag is set, then if the direction of the new packet and tail packet are different, the new packet must wait for the link to switch directions, and thus must wait for the latency delay of the tail packet. The start time of the new packet is set to the maximum of either: the time the tail packets latency delay is finished, plus a `HALF_DUPLEX_WINDOW_ERROR` parameter which simulates any extra time created because of error in the link layers partitions of send and receive windows; or the time it was received by the GSS. If the `DROP_HALF_DUPLEX` flag

TABLE II
PACKET SHAPING PARAMETERS

Parameter	Function
BAUD_RATE	The number of bits per second the hardware sends.
LATENCY_LENGTH	The number of milliseconds of latency between the ground station and satellite.
PARTITION_HALF_DUPLEX	A flag used to set the simulation of a link layer which partitions time for half duplex communications.
DROP_HALF_DUPLEX	A flag used to set the simulation of a link which simulations half duplex by dropping collisions.
HALF_DUPLEX_WINDOW_ERROR	The number of milliseconds of unused time added as error due to inaccuracies in half duplex partitions. Only used when PARTITION_HALF_DUPLEX is set.
DROP_RATE	The percentage of packets which are dropped.
CORRUPTION_RATE	The percentage of bytes which are corrupted.

is set, then a simulation is used which represents the case of having a link layer which does not partition send and receive time slots, such as AX.25 [**?**], a protocol commonly used on CubeSat amateur radio links. To simulate this, if the packet on the tail of the queue is a different direction than the new packet being added, and the new packet starts before the tail packets latency delay is finished, then the new packet is dropped.

Once a packet is removed from the head of the queue, it passes through packet shaping which applies packet drops and corruption. A random number generator is used to produce a floating point value between 0 and 1, and is compared against the DROP_RATE threshold. If the random number is below the threshold, then the packet is dropped and never sent to its destination. Similarly, a random number between 0 and 1 is generated for each byte of the packet and compared again the CORRUPTION_RATE parameter. If the number is below this threshold, then the byte is corrupted by replacing it with a bitwise inverse. The packet is then returned to the standard code path of the GSS which forwards the packet to its target destination.

V. ARSFTP DESIGN

A. Limitation Designs

Several design decisions were made specifically to increase the performance of ARSFTP in the presence of the various limitations of satellite communication over satellite radio.

1) High Drop and Corruption Rates: One major limitation of nanosatellite communication is the high packet drop and packet corruption rates. This is due to both the noisy interference caused by the signals passing through space, as well as the relatively low receive strength and transmit power of radios on CubeSats compared to radios on larger satellites.

The high drop and corruption rate prevents utilization of protocols which assume a clean link. One prime example is TCP stacks which implement exponential back-off on a packet loss. In order to see the effects of packet drops and packet corruption of a file transfer protocol over TCP, an experiment was run where a 4490 hyte file was sent from one machine to another through the ground station server. The ground station server applied a 9600 baud rate and 100ms latency delay to each packet using full duplex, and then performed packet dropping and packet corruption. Figure **??** shows the effects of different drop and corruption rates on the link utilization of the file transfer for transfers which did not time-out.

Based on this test, it is apparent that this exponential back-off severely affects the performance of any protocol running on top of it when going over a communication link with high drop rates. This is one reason UDP was chosen for the transport layer protocol, in addition to the fact that TCP's sliding window protocol can negatively impact performance due to high latency and half-duplex communication.

Fig. 2. The time of completion for the file transfer running on TCP vs. the packet drop and packet corruption rate.

Another impact high packet drop and corruption rates had on our protocol design was the consideration of sending redundant data in hopes that one packet would make it, in order to avoid dropping a packet and having to incur the high cost of switching from send to receive in a high latency, half duplex link. This is further described in the ARSFTP Implementation and Results sections.

2) High Latency: Due to the large distances of roughly 160-300km between CubeSats and ground stations [**?**], there is significant latency between when a packet is sent and when it is received. This can cause large delays and decrease link utilization when protocols are waiting for a reply (i.e. a sliding window protocol when a window is closed).

In order to illustrate this effect, two programs were written which ran on top of UDP. The first sent a 256 byte packet and waited for a 256 byte reply 500 times, and the second sent 500 packets and then waited for 500 reply packets. Both programs sent and received the same amount of data, but the former used a stop and wait protocol while the latter sent everything in one batch and received everything in another batch. These two programs were run through the packet shaping ground station server with packet shaping set up for 9600 baud with a 100ms latency delay using half-duplex. No packet drops or corruption were induced. Figure **??** shows the results, with the stop and wait styled program taking 70.238 seconds to complete and the batch program taking 19.041 seconds

Fig. 3. The time to program completion of the stop-and-wait program and the pipeline program which batched sends and receives separately.

ARSFTP was designed to pipeline data and avoid the penalty of waiting for a receive by sending every block of the file needed by the receiving end at once.

3) Half-Duplex Communication: The radio transceivers used on current CubeSats are half-duplex, and must be switched from a receive mode to a transmit mode, and vice versa. This is also the case for many amateur radios used by nanosatellite ground station operators. This configuration especially limits existing file transfer protocols as many assume a full-duplex communication link in order to continuously send file blocks one direction and acknowledgment packets the other direction.

The cost of switching between sending and receiving introduces three costs which can greatly reduce utilization on a link layer designed for half-duplex. The first cost is due to the fact that there is a relatively low granularity of time synchronization between the ground station and satellite, which can cause a receiver to switch early and miss packets, or switch late and reduce link utilization. Another cost is attributed to the high latency communication, and requires that the full latency delay be incurred for the final packet. The final cost can occur because of the time it takes to switch between the two modes on a given piece of hardware.

On a link layer which is designed to partition send and receive time between to nodes, this can cause a great number of lost packets due to hardware being in the incorrect mode. For this reason, we assume that the link layer ARSFTP will run on is designed to partition send and receive windows between the client and server.

In order to maximize link utility, ARSFTP was designed to reduce the total number of switches between sending and receiving. In order to accomplish this, communication during the file transfer is divided into two phases which send the largest amount of non-redundant information as possible. In one phase, all the needed blocks of the file are sent in one direction, and in the other phase a group of acknowledgment packets are sent in the other direction.

4) Limited Communication Window: Another limitation of satellite communication is the limited time window for communication. For a nanosatellite which is in low earth orbit, this equates to roughly 5 to 15 minute passes when the satellite is within line of sight of the ground station, and roughly 6 passes per day, depending on the orbit and ground station location [**?**]. For file transfer protocols which require that a transfer completes without timing out, this reduces both the file size that can be transferred, and creates wasted time at the end of a pass where there is not enough time for a transfer to complete.

In order to overcome this limitation, file transfer state is saved when a timeout occurs with an identifier so that a file transfer can be resumed at a later time.

VI. Protocol Implementation

Two programs were written in C: a client program `arsftp`, which is run by the mission operator on the ground, and a server program `arsftpd`, which is a daemon that runs on the satellite.

A. File Transfer

ARSFTP supports both the transfer of data from the server (satellite) to the client via the GET command and the transfer of the data from the client to the satellite via the PUT command. A file transfer using ARSFTP is broken down into three phases: initialization, file transfer, and completion.

The initialization packet is the first packet sent from the client to the server, and contains the fields listed in Table **??**. For the GET command, the file size field is zeroed out, and the server will respond with another initialization packet which contains the correct file size. Currently the only flag bits utilized in the initialization packet are a bit to indicate a resuming GET or PUT that will attempt to continue a timed out file transfer and a bit to indicate an access violation or file not found error error used by the server when returning the second initialization packet.

TABLE III
Initialization Packet Fields

Field	Size
Filename	128 Bytes
File Size	4 Bytes
Block Size	4 Bytes
Flags	1 Byte
Command	1 Byte
Checksum	2 Bytes

On a PUT command, the server will acknowledge the file transfer using an ack packet which contains the fields listed in Table **??**, and on a GET command the client will do the same. The acknowledgement packet contains two flags which can be used by the server to signal an access violation error. The ack packet also contains a bit vector with each bit representing whether or not a block is needed. The total size of the bit vector, in bytes, is shown by (**??**). The reason the first ack packet contains this bit vector is to allow for the ability to resume file transfers, in which the first ack packet contains information on the state of the transfer before timing out.

TABLE IV
Acknowledgement Packet Fields

Field	Size
Checksum	2 Bytes
Flags	1 Byte
Ack Vector	Variable

$$AckSize = \left\lceil \frac{\left(\frac{FileSize}{BlockSize}\right)}{8} \right\rceil \tag{2}$$

Once the header is received by the target and the request is accepted, the transfer moves to the file transfer stage of the protocol. The sender then sends the entire file to the receiver in a pipelined fashion without stopping to wait for an acknowledgement from the target until after the whole file has been sent. It does so in a block resolution, sending a block packet with fields shown in Table **??** for each block requested by the receiving end. After the file has been sent, the receiver sends an acknowledgment packet, containing which blocks of the file it is missing, to the sender. The sender then retransmits the missing blocks to the target. This cycle

of asking for missing blocks and resending only the missing ones continues until the receiver acknowledges that is has successfully received the entire file.

TABLE V
FILE BLOCK PACKET FIELDS

Field	Size
Block Number	4 Bytes
Checksum	2 Bytes
Block Data	Variable

After the sender has received an ack flag which acknowledges that all blocks have been received, it moves to the completion stage of the transfer. It will send a series of finish acknowledgement packets to the satellite to acknowledge that the transfer is complete. Once the finished acknowledgement is received, the file transfer is terminated successfully.

At each stage of the process, packet timeouts are present. On a timeout, the side which has timed out on a packet will resend the packet(s) it is waiting for a response from. Once a user defined threshold of packet timeouts occur, the file transfer times out and is terminated.

B. Resuming Timeouts

In order to allow the ability to resume a file transfer after a timeout, a directory was created to store timed out file information for both the `arsftp` and `arsftpd` programs. When a file transfer times out, instead of removing the file and all state information of the transfer, it is stored in this directory. The directory contains timed out files and a mapping file. The mapping file contains a dictionary which uses the filename and file checksum as a key and the ack flags, file size, block size, and timeout timestamp as values. When a timeout occurs the file being transferred is moved to this directory, and the mapping file is updated to include an entry for the file.

When a resuming PUT file transfer command is received by `arsftpd`, it first checks the mapping file in the timeout directory using the PUT files filename and checksum to see if an entry exists. If an entry does exist, it moves the partially completed file to the destination, loads the ack flags from disk, and continues the file transfer using the entry as its current state. The first ack packet sent will contain the flags loaded from disk, thus the `arsftp` program will only send file blocks needed by the server. If the file does not already exist in the timeout directory and mapping file, then the file transfer continues as normal. Similarly, on a resuming GET command, the `arsftp` client program will check the file checksum returned by the server and use that, along with the filename, as the key into the mapping file.

Timeout timestamp values are stored along with keys to allow for the timeout directory to be maintained. Files can be periodically removed from the timeout folder using the timeout timestamp to prioritize the removal of older files.

VII. EVALUATION

To test the ARSFTP protocol, the following setup was used. A Dell OptiPlex 755 was used as the client computer running `arsftp`, a virtual machine on a Dell PowerEdge 2950 was used as the ground station server outlined in Sections 3 and 4, and a Dell OptiPlex 780 was used as the satellite server computer running `arsftpd`. Each test data point is the mean of five successful file transfers timed using the Unix `time` command. The link utilization was calculated as shown in (**??**).

$$LinkUtilization = \frac{\left(\frac{(FileSize*8)}{BAUD\_RATE}\right)}{TransferTime} \quad (3)$$

For the following tests, a 4490 byte image was sent from the ground station client computer to the satellite server computer using the PUT command. Each set of transfers were parametrized with different drop and error rates. Both file transfer protocols were configured to use 512 byte block sizes. All packet drops and byte corruption rates indicate artificial drops and corruption added to network traffic by the simulator, and do not include normal network drops or corruption, drops due to half duplex collisions when a drop on collision link layer was being simulated, and tail drops due to a filled queue on the simulator.

Initially, our tests of the ARSFTP protocol showed sporadic file transfer times. Upon further investigation, it was noted that one major flaw of the proposed protocol was that only one acknowledgement packet was sent for each time every block needed by the receiving end was sent. Once packet drops and corruption were added to the communication, if the acknowledgement packet was dropped the sending side would react by resending all the blocks again based on the previously received ack bit vector. This resulted in a lot of redundant data being sent if one of the first few acks were dropped, and caused a large fluctuation in completion times depending on if, and which, acknowledgement packets were dropped.

In order to resolve this, the file transfer protocol was modified to send a number of duplicate acknowledgement packets at the end of a file transfer iteration, instead of just one. This reduces the chance that all ack packets would be lost and all received file blocks would be resent while minimizing the number of times the communication switches from sending to receiving. This comes at the cost of potentially sending redundant and unused ack packets. Figure **??** shows the link utilization using the naive one ack packet per file transfer iteration method, and a more consistent and robust four ack packet per file transfer iteration method.

Fig. 4. The link utilization of the ARSFTP protocol using one ack packet and four ack packets per file transfer iteration with varying (a) packet drop rates and (b) byte corruption rates.

The same test suite was run on both ARSFTP and ATFTP, a UDP based file transfer protocol used for comparison against a protocol not designed with the limitations of amateur radio satellite communications in mind. This test was designed to show the effect of different packet drop rates and byte corruption rates on link utilization. The test was also run on several different half duplex configurations. The first is an ideal half-duplex configuration, where there is perfect partitioning

of send and receive windows between the server and the client. This is shown in Figure **??**. The second test was run using a half duplex configuration in which packets were dropped on a collision. This is shown in Figure **??**. Finally, a test was run using a non-ideal half-duplex configuration, with a one second buffer added between send and receive windows during which no packets are sent. This was designed to simulate a practical link layer which accommodates synchronization differences between the ground computer and satellite. The results are shown in Figure **??**.

Fig. 5. The link utilization of the ARSFTP and ATFTP protocol on a half-duplex link layer which ideally partitions send and receive windows, with varying (a) packet drop rates and (b) byte corruption rates.

Fig. 6. The link utilization of the ARSFTP and ATFTP protocol on a half-duplex link layer which drops on collisions, with varying (a) packet drop rates and (b) byte corruption rates.

Fig. 7. The link utilization of the ARSFTP and ATFTP protocol on a half-duplex link layer which adds a one second buffer between send and receive windows, with varying (a) packet drop rates and (b) byte corruption rates.

In all tests, ATFTP outperformed ARSFTP when there were no packet drops or byte corruption, but once the two were added to the simulation the link utilization of ATFTP fell significantly. On the other hand, the link utilization of ARSFTP did not decrease much when artificial packet drops and byte corruption were added to the simulation.

The 10% packet drop rate and 0.0008% byte corruption rate were used as a test case for real satellite communication conditions. Figure **??** shows a comparison of the link utilization for the three half duplex configurations for both ARSFTP and ATFTP. Both protocols performed relatively poorly when using a drop on collision half-duplex compared to an ideal half-duplex. ARSFTP is affected more because if all acknowledgement packets collide or are dropped, a high cost is incurred. ARSFTP performs very well with a practical half duplex link layer which partitions time windows, dropping only 1% in link utilization when compared to the ideal case. This is because ARSFTP is designed specifically to avoid switching between sending and receiving and the link utilization penalty associated with such switches. In contrast, ATFTP performs relatively poorly with a practical partitioning scheme as it is designed with a full duplex link layer in mind, and expects to be able to send acks at the same time blocks are being sent.

Fig. 8. Comparison of ARSFTP and ATFTP using three different half-duplex configurations, with a 10% packet drop rate and 0.0008% byte corruption rate.

Using the non-ideal half-duplex time partitioning scheme with a 10% drop rate and 0.0008% byte corruption rate being simulated, files of varying sizes were transferred using ARSFTP. The test was performed with ARSFTP configured to 512 byte blocks. The results are shown in Figure **??**. Increasing the size of the file being transferred increases link utilization, suggesting that the initialization and completion portions of the file transfer are a source of significant overhead. The file transfer portion of the protocol, on the other hand, produces high link utilization. This is likely because the initialization and completion portions of the protocol require communicating back and forth, which is costly on the half-duplex link.

Fig. 9. The link utilization of ARSFTP when transferring files of various sizes over a non-ideal half duplex link using a 10% packet drop rate and 0.0008% byte corruption.

A test was then performed using the non-ideal half-duplex time partitioning scheme using ARSFTP with various block sizes. The test was performed with a 10% drop rate and 0.0008% byte corruption rate being simulated, with a 44.9 kB file being transferred. The results are shown in Figure **??**. As a result of analyzing the protocol, two limiting factors were found regarding block sizes. Increasing the block size will reduce the number of file block headers sent, but will incur larger penalties when a byte is corrupted for a packet, as the whole block is dropped. However, if the number of acknowledgement packets is fixed, then decreasing the block size will present the problem of increasing the acknowledgement packets size. With a larger acknowledgement packet, there is a greater chance of corrupting a byte in the packet and losing the whole acknowledgement. In order to alleviate the latter limitation, the protocol was changed so that a variable number of acknowledgement packets are sent depending on the ack size. The optimal number of acknowledgement packets to send is presented in (**??**), with sigma being the standard deviation of the packet error rate from (**??**).

$$PacketErrorRate = (\%PacketsDropped) * (1 - (\%BytesCorrupted))^{AckSize} \quad (4)$$

$$NumberOfAcks = \frac{(1 + 3\sigma)}{PacketErrorRate} \quad (5)$$

Fig. 10. The link utilization of ARSFTP when transferring a 44.9 kB file over a non-ideal half duplex link using a 10% packet drop rate and 0.0008% byte corruption rate using several different block sizes.

After making this change, a block size which would fit into MTU of the link layer being simulated was hand calculated. The block size of 218 bytes was calculated by taking the 256 byte MTU, subtracting the 24 bytes for the IP header, the 8 bytes for the UDP header, and the 6 byte header of ARSFTP's file block packets. Using this block size minimizes overhead of headers from the link, network, and transport layers on the transfer.

Using the new 218 byte block size, a series of tests were run to determine the maximum amount of data which could be sent in a single fifteen minute CubeSat pass. The simulator was set to a 10% packet drop rate and 0.0008% byte corruption rate simulation over a non-ideal half-duplex link. Table **??** shows the results of the test. On average, a 750kB file was able to be transferred within a fifteen minute satellite pass, as opposed to 26kB using ATFTP, resulting in 24 fold improvement of link utilization when using ARSFTP as opposed to ATFTP.

TABLE VI
MAXIMUM FILESIZE FOR A SINGLE PASS FILE TRANSFER

Protocol	File Size	Transfer Time	Link Utilization
ARSFTP	750kB	14m 54s	0.72
ATFTP	26kB	14m 33s	0.03

VIII. FUTURE WORK

The majority of the work on ARSFTP presented in this paper was directed towards designing an efficient file transfer protocol with the focus being on the actual file transfer, via the GET and PUT commands. For this reason, some other commands, such as list and remove commands, have not yet been implemented, as there is much less room for optimization. We plan to implement these abilities in the near future, with a specialized form of file listing which recursively lists directories. This is important because if a user wishes to find a file in a directory tree which they are unfamiliar with, the act of listing each file individually will drastically decrease performance due to large latencies in communication.

Another area where future work is possible is in the user interface of the `arsftp` program. Because the program will be used by mission operators who may not have computer-related backgrounds, it is important that the interface be relatively simple and intuitive so that it can be used by many people to collect payload data.

Another important area of future work is making improvements to different aspects of the ARSFTP implementation. Currently projected improvements are the compression of the protocol header and ACKs to reduce overhead, implementing a form of user permissions in the protocol, and determining the optimum values for configuration attributes in ARSFTP for an actual missions communication characteristics.

The final addition to ARSFTP which would be desired is the ability to parametrize the file transfer protocol (i.e. change the block size or timeout durations) remotely. This is especially important as communication characteristics of CubeSats have not been well-studied, and being able to efficiently adjust the properties of the file transfer protocol on a satellite in space would allow mission operators to maximize file transfer efficiency by reconfiguring ARSFTP to suit their missions communication characteristics, thus allowing them to collect more payload data.

IX. CONCLUSION

Although many protocols exist for the transfer of files between two machines, very few are designed to account for the drop rates, error rates, half-duplex link layer, and latency introduced by amateur radio communication. To solve these problems, we propose the use of ARSFTP, a protocol designed specifically for satellite communication over amateur radio. ARSFTP overcomes the various limitations of satellite communication through the use of pipelining and minimizing switches between sending and receiving. ARSFTP sends the entire file at once and then waits for a single acknowledgement packet which contains which blocks the recipeint is missing. ARSFTP then repeats the process, sending only those missing blocks. This behavior allows ARSFTP to maximize its link utilization and throughput. Under realistic conditions, ARSFTP is able to transfer a 750kB file in just under 15 minutes with a link utilization of 71%. In comparison to ARSFTP, the existing UDP file transfer protocol ATFTP was only able to transfer 26kB in just under 15 minutes, with a link utilization of 3%, under the same conditions, resulting in a 24 fold improvement of link utilization by ARSFTP.

Although we have not completed all optimizations at the writing of this paper, the initial version of ARSFTP is already several orders of magnitude more efficient than the existing fixed length and block resolution downlinks present on CubeSat nanosatellites, and an order of magnitude more efficient than using a traditional file transfer protocol.

In addition to the implementation of ARSFTP, we have also implemented a simulator for satellite communication over amateur radio packet shaping on our existing ground station server. Through the use of the different parameters in our packet shaping, we are able to run tests to determine the effects of different amateur radio satellite communication characteristics on programs communicating over Ethernet.

X. ACKNOWLEDGMENT

We would like to thank Dr. John Bellardo for helpful discussions and feedback. We would also like to thank PolySat for the equipment used during testing.