Implementing a Verified FTP Client and Server

by

Jennifer Ramseyer

S.B., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Dr. Martin Rinard
Professor
Thesis Supervisor

Accepted by .....
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Implementing a Verified FTP Client and Server

by

Jennifer Ramseyer

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

I present my implementation of an FTP: File Transfer Protocol system with GRASShopper. GRASShopper is a program verifier that ensures that programs are memory safe. I wrote an FTP client and server in `SPL`, the GRASShopper programming language. `SPL` integrates the program logic's pre- and post- conditions, along with loop invariants, into the language, so that programs that compile in GRASShopper are proven correct. Because of that, my client and server are guaranteed to be secure and correct. I am supervised by Professor Martin Rinard and Dr. Damien Zufferey, a post-doctoral researcher in Professor Rinard's laboratory.

Thesis Supervisor: Dr. Martin Rinard
Title: Professor

Acknowledgments

I am grateful to Professor Rinard, for his help and advice throughout my project, along with his encouragement and enthusiasm. As Prof. Rinard would say, "Be brilliant!" I have tried.

Many thanks to Damien Zufferey, for his invaluable help with GRASShopper and my FTP implementation. His knowledge of the intricacies of GRASShopper is incredible and indispensable.

I would also like to thank David Stingley for his collaboration on the SNTP code. Thanks to all of Professor Rinard's group for their fun Wednesday lunches and advice on navigating the perils of graduate school.

I am grateful to Damien and Mark Ramseyer for their advice and edits on the many drafts of this thesis.

I thank also all my friends at MIT for making my five years here a wonderful experience.

And of course, four thousand, two hundred and ninety-nine thanks to my Pickle family, for their friendship and support all these years. Without them, I'd never have been inspired to explore computer science in the first place.

Finally, many thanks to my family–Norma, Mark, Geoff, Eli, and Pichu, for their love and support all these years. Thank you for fostering my enthusiasm for the world for as long as I can remember.

Contents

List of Figures

List of Tables

Chapter 1

Background

I am supervised by Professor Martin Rinard and Damien Zufferey, a post-doctoral researcher in Professor Rinard's lab. My project is to implement utility programs, such as the File Transfer Protocol (FTP), in GRASShopper. GRASShopper [PWZ14] is a program verifier which uses separation logic [Rey02] to analyze programs that access and change the heap [PWZ13]. It is written and maintained by Zufferey, Thomas Wies of NYU, and Ruzica Piskac of Yale University. GRASShopper uses Z3, a SMT solver provided by Microsoft Research [dMB08].

1.1 GRASShopper

I use GRASShopper to compile my programs[1] to C. To use GRASShopper, I wrote my programs in `SPL`, the GRASShopper language. I present an overview of `SPL`, the GRASShopper input and specification language, along with an explanation of how GRASShopper compiles to C.

1.1.1 SPL

`SPL` is a C-like language, but with simpler syntax. See Figure 1-1 for a sample `SPL` program. It treats memory-accesses like Java, in that there are no allocations to the

[1] Code accessible at: https://github.com/JaneEyre446/grasshopper

stack and no function pointers. All arrays and structs are stored on the heap, and are passed only by reference. All arrays have a length field, which is set on allocation and cannot be changed. Unlike Java, `SPL` integrates the function specifications into the core language, allowing the functions to be verified [Zuf16]. `SPL` allows the writer to specify the desired pre- and post-conditions and loop invariants for the program, which z3 uses to verify the program.

1.1.2 Specification Language

GRASShopper is a program verifier which uses Hoare logic [Hoa69] and separation logic [Rey02] to reason about a program's partial correctness. In the following, I give a short introduction to the mechanism used by GRASShopper. A more comprehensive explanation can be found in "The Calculus of Computation," by Bradley and Manna [BM07]. In GRASShopper, every method is annotated with pre- and post-conditions and loop invariants. A pre-condition is an assumption put on a program's inputs. Similarly, a post-condition is a property that is proven true of the outputs of the program, given the preconditions. A loop invariant is a property that each iteration of the loop fulfills. As an example: in various parts of my program I require that all arrays have the desired size and type, that different arrays point to different memory locations when copied, and that flags and other values are in the desired range. By having these conditions, I ensure that there are no invalid array accesses, that arrays contain values of all the same type, that all values are in the desired range (if one exists), and various other useful conditions. These conditions make the program secure and verifiable. I can guarantee conditions put on packets, as in § 4.1. If the program verifies, then GRASShopper will translate it into C, which one can compile with `gcc`.

Each function in GRASShopper has pre-conditions specifying constraints on the inputs to the functions, and post-conditions guaranteeing the state of the variables and the return value once the function terminates. Please see Figure 1-2 for an example of GRASShopper's pre- and post- conditions.

For every loop in GRASShopper, I also must provide loop invariants, as in Fig-

ure 1-3. A loop invariant is a condition that is true at the beginning of each iteration of the loop.

1.1.3 Compiling to C

GRASShopper can output C code. Once a program has passed GRASShopper's verification checks, I can specify the optional `-compile [C filename]` flag, and GRASShopper will translate my `SPL` code into C. I can compile that code and link in other C files to produce an executable. Alternatively, I can also write function specifications in `SPL`, verify them with GRASShopper, and output C stubs for those functions. At C-compilation time, I can link in the C implementation for those functions. That way, I can use functions in the standard C library without necessarily implementing them in `SPL`. Those linked-in functions are part of our "trusted code base" (see § 4.2).

1.2 TCP

TCP, the "Transmission Control Protocol," is a standard for sending data between two machines over the internet [ISI81]. Under TCP, the first computer sends a packet to the second computer. The second computer must acknowledge – "ack" – the receipt of each packet, and send this ack back to the first computer. If the first computer does not receive an ack for a given packet within the time limit, then it must resend that packet. With this "ack" system, TCP helps prevent data loss when sending over the internet. If a packet is lost, the first machine will notice it never received an ack, and will resend it.

TCP implements First-In, First-Out (FIFO) channels. My FTP client and server communicate using TCP. TCP ensures that no packets are dropped while sending files. A user wants to be sure that the whole file was sent, and TCP's ack system helps provide that security.

```
1  include "byte_array.spl";
2
3  procedure copyByte(a: Array<Byte>)
4    returns (b: Array<Byte>)
5    requires byte_array(a)
6    ensures byte_array(a) &*& byte_array(b)
7    ensures a.length == b.length
8    ensures forall i: Int:: i >= 0 && i < a.length ==> a[i] == b[i]
9  {
10   b := new Array<Byte>(a.length);
11
12   var i := 0;
13
14   while (i < a.length)
15     invariant i >= 0 && i <= a.length && a.length == b.length
16     invariant byte_array(a) &*& byte_array(b)
17     invariant forall j: Int:: j >= 0 && j < i ==> a[j] == b[j]
18   {
19     b[i] := a[i];
20     i := i + 1;
21   }
22
23   return b;
24 }
```

Figure 1-1: The code to copy a byte array, written by myself, David Stingley (another MEng student), and Damien Zufferey. Lines 5 and 6 provide memory safety guarantees, while lines 7 and 8 show functional correctness. Lines 15 to 17 are the loop invariants, which are the pre- and post- conditions of the loop. `&*&` is the separating conjunction, which states that array a and array b are allocated in disjoint areas of the heap.

1.3 FTP: File Transfer Protocol

FTP, the "File Transfer Protocol," allows users and programs to send files from one machine to another[PR85]. In an FTP system, there is one server and one (or many) clients. The client connects to the server, and once connected, can upload or download files from the server. There are two modes of connection to the server: anonymous and username/password. In an anonymous system, the client can login without a username or password (or with password "anonymous"). In a username and password system, the client needs to have an account on the server, and log in to the server with that account's username and password. I implement an anonymous login system for my FTP server. The client and server are connected over two channels, one being

```
1 procedure accept_incoming_file(dataFd: Int, filename: Array<Byte>,
    allo_size: Int) returns (res:Int)
2   requires dataFd >= 0
3   requires allo_size >= 0
4   requires allo_size <= 65535
5   requires byte_array(filename)
6   ensures byte_array(filename)
7   ensures res >= -1
```

Figure 1-2: The pre- and post- conditions required for the procedure `accept_incoming_` file.

```
1   while(cmdFd < 0)
2     invariant cmdFd >= -1
3     invariant socket_addr_4(cmdAddr)
4     invariant tempCmdFd >= 0
5   {
6     cmdFd := connectMeCommand(tempCmdFd, cmdAddr);
7   }
```

Figure 1-3: An example of a loop invariant in the server.spl code.

a data connection and the other a command connection. The client and server send file contents over the data connection, and commands over the command connection. The commands are sent as plaintext, while the data are sent as binary. The client can inquire about the files on the server, upload files from the client computer to the server computer, or download files from the server to the client computer.

FTP is often used to transfer files from one computer to another or to store files on a remote computer cluster. Anonymous login FTP is often used for downloading software updates.

Chapter 2

Problem

C, one of the industry standard languages, does not provide memory safety or program logic guarantees. Because of that, there have been many critical security bugs [Cod, Dat15, Mic03, Kno07] in the news which could have been avoided, were C to be a safer language.

2.1 Memory Safety

A language is memory safe if memory can only be accessed when valid. Memory is valid if it has been allocated and not yet deallocated. Memory safety also guarantees that no memory is accessed after it has been freed and that the program cannot write to memory outside the allocated memory. A memory safe program can only access an array's memory within the bounds that are explicitly allocated. In programs that are not memory safe, a program can access memory outside the bounds of allocated memory, and can leak memory if it is not freed. Memory leaks make programs run slower over time, as the memory is never freed. A hacker can exploit a memory unsafe program by reading memory outside the allocated bounds for important details, as in the Heartbleed bug [Cod]. GRASShopper is memory safe, so all memory that is allocated is freed, and all memory accesses must be within the specified bounds, and no memory or pointers are accessed after being freed.

Chapter 3

Previous Work

There have been many papers and programs written about program verification. I focus on GRASShopper, the program verifier I used for my thesis, and mention many other systems.

3.1 GRASShopper

GRASShopper is a program verifier written by Ruzica Piskac (of Yale University), Thomas Wies (of New York University), and Damien Zufferey (of MIT CSAIL). GRASShopper guarantees memory safety for all programs, and functional correctness for some data types [Zuf16]. For use with GRASShopper, Piskac et al. have developed `SPL`, a programming language that supports separation logic and first-order logic as specification in the language [PWZ14]. GRASShopper uses Microsoft's z3 [dMB08] as a verification backend. If the program fails to verify, GRASShopper points out where and why the constraints are unsatisfied, greatly simplifying the debugging process [PWZ14].

One feature I find invaluable in my thesis is that GRASShopper can compile to C. Given an `SPL` program, GRASShoppper can verify it, and convert it to C code. With this C code, I can compile it to an executable using `gcc` (or some other C compiler) and run it. I can also link in other C code, and even verify `SPL` methods that use C code (see § 4.2) [Zuf16]. Since my end goal of my thesis is to output a usable FTP

client and server, I relied on being able to produce an executable.

3.2 Other Work

A similar project by Microsoft Research, Dafny [Lei10], offers much of the same functionality as GRASShopper. Dafny is designed to work with Microsoft Visual Studio and outputs .NET executables, while GRASShopper can work on any system, and is able to output C code which can be used in any C project [Lei10]. GRASShopper can also link in other C files to be compatible with other code and libraries. Also from Microsoft Research, IronFleet is a verification system intended for larger systems than Dafny. IronFleet is used to prove correctness on large distributed systems [HHK+15]. Microsoft Research has also written IronClad, a system used to make verifiably secure applications, on an end-to-end verified and secure system [HHL+14].

Another system is Verifast [JSP+11], by Bart Jacobs et al. It is similar to GRASShopper, in that it is a verifier for Java and C programs. Facebook has also produced a static analysis program, called "Infer" [Fac]. Infer reads the specified code and points out possible bugs in the code. Unlike GRASShopper, it does not verify the correctness of the code, but helps to find problems in code before it is released to the general public. Jahob [BKW+07], is a program verifier for Java. Viper [MSS16], from ETH Zurich, is an intermediate verification language. Viper is an infrastructure which accepts programs written in several languages for verification. The user can also add backend verification and analysis tools, all of which can be run through Viper.

Chapter 4

My Work

I implemented an FTP client and server in GRASShopper. I detail the implementation process for the client and server, the guarantees they provide, and the additional helper wrappers I wrote. I also give instructions for how to run the programs, and describe some problems I solved throughout the process.

4.1 SNTP

Before implementing FTP, I needed a better grasp of how GRASShopper worked. To that end, I implemented SNTP: the Simple Network Time Protocol [Mil06]. I worked with David Stingley, another MEng student in Professor Rinard's lab, on the SNTP code for a client. Stingley changed projects after the SNTP project, so the FTP code is my work, with help from Damien Zufferey. I wrote the SNTP code with Stingley to better understand GRASShopper before tackling a larger program like FTP.

SNTP is used to synchronize system time on computers [Mil06]. Each computer queries a time server on the internet, which sends back the current time. The computer then uses the returned time to calculate and correct the system time.

My implementation of an SNTP client relies on file input/output instead of networking calls. Instead of reading packets from the time server, it reads dummy packets from a file on the computer. By porting the SNTP client to GRASShopper, I am able to say that my SNTP client is verifiable, i.e., that is has safe memory accesses,

fulfills the given pre- and post-conditions, and has no memory leaks [PWZ14].

My version of the SNTP client is modeled off of an implementation of SNTP written in Go [lub14]. I ported it to `SPL`, and added the appropriate pre- and post-conditions. As the main point of the SNTP project was not to figure out how to write an SNTP client, but instead to figure out how to use GRASShopper, porting it is a reasonable design decision. Please see the Appendix or my GitHub for our SNTP client implementation at time of writing.

When David and I wrote the SNTP client, we had not yet written the wrapper to the socket library. Therefore, our version reads mock packets from a file instead of querying the network time server on the internet. It is a simpler implementation for testing purposes. The next step would be to have it query the time server [Han15] and read actual packets. I would need to have it call UDP networking functions instead [Pos80]. However, since the purpose of the SNTP code was to learn GRASShopper, I switched to working on FTP instead, as it is more technically challenging and useful.

4.2 Wrapper for C Library

GRASShopper compiles to C. For utility functions that are not defined within the GRASShopper files but are part of the standard C library, Zufferey and I provide pre- and post- conditions for those function stubs in a GRASShopper file. GRASShopper uses the stub specifications to verify the functions. When compiling the generated C code, one can link in other C files containing implementations for the function stubs. Zufferey and I used this feature to write the file I/O, console, and networking code, which I then use in the FTP and SNTP code. When compiling the generated C code for the FTP client and server, I link in a file containing implementations of these calls. GCC links the files and produces an executable that one can run.

By linking in other C files, we assume that those C functions are correct. If this subset of "assumed correct" programs is small, then that is an acceptable assumption to make for the trusted code base. Most programming languages are built off of some

core subset of syscalls or functions in other languages, along with the compiler (in my case, GCC). I also include z3 and GRASShopper as part of the trusted code base. This trusted code base should be as small as possible.

For both SNTP and FTP, I needed file input and output (file I/O) functions like `open`, `close`, `read`, etc. Rather than implement those functions directly in GRASShopper, which would have have required adding many more capabilities to GRASShopper which is out of the scope of an MEng thesis, I used GRASShopper's linking capabilities. As part of the SNTP code, Zufferey and I implemented file I/O functions in GRASShopper. I wrote function stubs in SPL for the functions, with pre- and post- conditions, and then Zufferey and I implemented the wrappers to the functions in the standard C library, as in Figure 4-1. In most cases, we simply had the GRASShopper call the relevant C standard library function, but sometimes we had to handle edge cases. One such edge case was to check whether or not the file name string was null-terminated or not, before trying to open the file. GRASShopper carries the length of each array. Thus, I can use the length to mark the end of the string instead of the null character. As a result, I need to check how the filename is represented as a string before opening the file.

4.3 FTP Implementation

For FTP, I needed to write both a client and a server. This client and server connect over two channels: the command connection and the data connection.

4.3.1 Setting Up the Data and Command Connections

An FTP client and server connect on two different ports. One port is used for sending commands and command responses back and forth (the "command" or "control" connection), and the other is used for sending file data back and forth (the "data" connection). According to the FTP standard, the server command connection should be over port 21, and the server data connection should be over port 20. The client command and data connections are, by default, both over the same port, which can

```
1    procedure gread(fd: Int, buffer: Array<Byte>) returns (nbr: Int)
2        requires fd >= 0
3        requires byte_array(buffer)
4        ensures byte_array(buffer)
5        ensures buffer == old(buffer)
6        ensures buffer.length == old(buffer.length)
7        ensures nbr >= -1 && nbr <= buffer.length
```

(a) The GRASShopper function stub for `gread`, the function to read from a file. Here, GRASShopper requires that the `gread` function returns an integer. GRASShopper also requires that, at the start of the function, the file descriptor be greater than zero, and the buffer be an array of bytes. GRASShopper ensures that at every return point of `gread`, the buffer is still a byte array, the buffer has not been modified from the original buffer passed in, and the length of the buffer has not changed. GRASShopper alsos ensure that the return value, nbr, be greater than or equal to -1 and less than or equal to the length of the buffer.

```
1    int gread (int fd_2, SPLArray_char* buffer) {
2      return read(fd_2, buffer->arr, buffer->length);
3    }
```

(b) The C implementation for `gread`. Here, `gread` takes a file descriptor and a character array as inputs, and passes them to the C standard library function, `read`.

Figure 4-1: Linking in C files to GRASShopper-outputted code

be any port [PR85]. Since those ports are privileged ports, I use port 4444 as the command port for both the client and server, and port 4440 for the data port for both client and server.

4.3.2 Required Commands

The standard FTP implementation has many different commands, but I reduced the library down to the eight commands I deemed essential. In many cases, the client checks for one of several acceptable command responses from the server, even though the server only sends one predetermined response. I accept many replies so that my client will be usable after slight modification with servers other than my implementation (see § 5.3). For a diagram showing in what order these commands are used, please see Figure 4-2

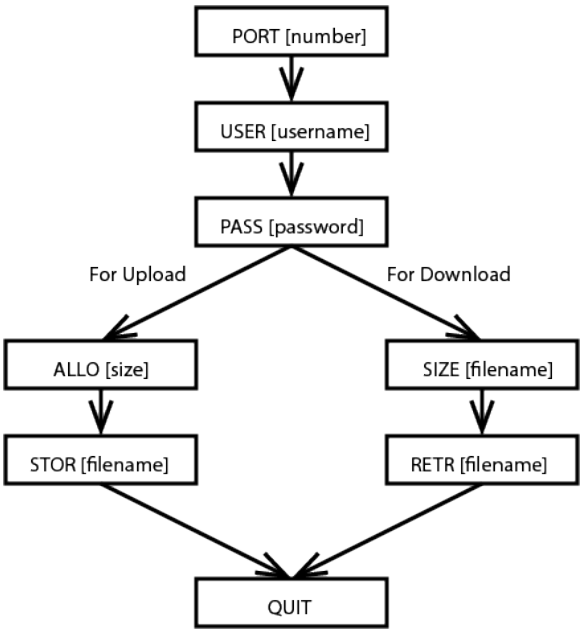

Figure 4-2: The order in which the FTP commands are sent by the client.

ALLO [size]

The client sends `ALLO [size]`, where `[size]` is a file size, to the server, so the server knows what size of file to allocate on the server. It is used during upload.

If the server receives a valid size, it replies `200`, meaning "Command okay" [PR85]. If the server receives an invalid size, it replies `552`, which means "Requested file action aborted" [PR85].

PORT [number]

The client sends `PORT [number]` to tell the server what port to open as the data connection. The server does not reply, but connects to the port specified by the client.

STOR [filename]

The client sends `STOR [filename]` to tell the server that it would like to upload `[filename]`. The server replies with either `200` or `150`. `200` means "Command okay", and `150` means "File status okay" [PR85]. In my implementation, the server sends

150, and the client checks for 200 or 150. Then, the server receives the file contents. If it succeeds, it replies with 250 ("Requested file action okay, completed" [PR85]). If it fails, it sends 550 ("Requested action not taken. File unavailable" [PR85]). The client checks for any of 200, 226 ("Closing data connection. Requested file action successful" [PR85]), or 250.

SIZE [filename]

The client sends `SIZE [filename]` to the server to ask for a the size of the file. The server replies with `213 [filesize]` if the file exists and is accessible, and 550 otherwise. The client checks for 213.

RETR [filename]

The client sends `RETR [filename]` to tell the server it would like to download `[filename]`. The server replies with 150. The client checks for 150 or 200. Then, the server sends the file contents. If it fails to send, the server replies with 550. If it succeeds, it replies with 250. The client checks for any of 200, 226, or 250.

USER [username]

The client sends `USER [username]` to the server to login. The client checks for any of 200, 230 ("User logged in, proceed" [PR85]), 234 ("Security data exchange complete" [Sup09]), or 331 ("User name okay, need password" [PR85]). The server replies with 331, as it never uses the username. My system supports anonymous login.

PASS [password]

The client sends `PASS [password]` to the server to login. The client checks for any of 200, 202 ("Command not implemented, superfluous at this site" [PR85]), 230, or 234. The server replies with 230 if the password is correct and 530 ("Not logged in") otherwise.

QUIT

The server does not reply to a `QUIT` message, as the client has already disconnected.

4.3.3 How the FTP Client Works

For all sockets used in the client and server, I open them with the flags `PF_INET`, `SOCK_STREAM`, and `IPPROTO_TCP`. I need `PF_INET` to indicate I am using IPv4 internet protocol, `SOCK_STREAM` since I am streaming data, and `IPPROTO_TCP` since I am using TCP.

Initially, the client asks the user via the command line whether she would like to upload or download a file. The client reads the user's response from the command line. Next, the client sets port 4444 as the port for the command connection to the server. The client asks the user for the IP address of the server using the command line, and opens a socket and connects to that IP. The client saves the file descriptor of the socket as the command connection.

The client then needs to set up the data connection over port 4440. With the client's IP address, the client opens a socket, binds to the socket, and listens for an incoming connection from the server. The client sends `PORT 4440` to the server over the command connection, and accepts the incoming connection from the server. Now that the server has connected, the client saves the file descriptor of the socket as the data connection.

Once connected, the client needs to authenticate to the server. It sends `USER Potato` over the command connection to the server. (My system supports anonymous login, so the client can put whatever it wants for the username.) The server sends back a response, and if it is okay (see § 4.3.2), the client sends `PASS anonymous` to the server over the command connection. My system is an anonymous login system, so the password is "anonymous." The server sends a reply, and if it is okay, then the client continues.

Next, the client asks the user for the filename of the file she would like to upload or download. The client prints `input the filename:` onto the command line, and

then accepts the user's response as the filename. If the filename is less than 100 characters long, and greater than one character long, the client null-terminates the filename and uses it. If it is not within those bounds, the client rejects it and shuts down.

Here the client takes one of two routes: upload a file or download a file.

Upload

In order to upload, the client measures the size of the file to upload. Then, it sends the size of the file to the server over the data connection, using the `ALLO [size]` command. Provided that the client receives an acceptable response, the client opens the file with the `O_CREAT` and `O_RDONLY` flags. `O_CREAT` ensures that if the file does not exist yet, a blank file is created. `O_RDONLY` opens the file in read-only mode. The client reads the file into an array `A`.

Next, the client sends the `STOR [filename]` command to the server over the command connection. If the response is okay, the client sends array `A` to the server over the data connection. The server responds, and if it is okay, the client closes the file. If the amount of data sent to the server over the data connection is the same as the size of the file, the client has successfully uploaded the file.

Download

In order to download a file, the client needs to know the size of the file. It asks the server for the size of the file using the `SIZE [filename]` command over the command connection. The server sends back the size of the file. If the size is valid (greater than 0, and less than or equal to the maximum size of an array, 65535), the client allocates an array `B` of the appropriate size and sends a request to download the file to the server: `RETR [filename]` over the command connection. If the server sends back the appropriate response code, the client receives the incoming file data into array `B` over the data connection. The client receives the server response over the command connection. If the client received at least zero bytes, then it opens file `[filename]` with the `O_TRUNC`, `O_CREAT`, and `O_WRONLY` flags. The `O_TRUNC` flag indicates that if

the file already existed, it will be wiped on opening so the client can overwrite the file. I use the `O_CREAT` flag to create the file if it does not already exist. And the `O_WRONLY` flag opens the file in write-only mode.

Finally, the client writes the data from array `B` into the file, and closes the file.

After upload or download, the client closes the data connection. Then, it sends the `QUIT` command to the server over the command connection, indicating its plan to disconnect. The client closes the command connection and returns 0.

4.3.4 How the FTP Server Works

The server uses "localhost" as its IP address, since to it, it is running locally. It computes the address of localhost on port 4444, the command connection. It then opens a socket and binds the address to it. Next, it waits for the client to connect to it. While waiting, it listens for a connection and accepts the connection once it arrives.

After the client connects, the server sets up the data connection. It receives the client's `PORT [number]` message, and opens a socket. It computes the client's address on port `[number]`, and connects to the client at that address.

The server then receives the client's username and password over the command connection. Since I am using anonymous login, the server ignores the client's username. If the client sends "anonymous" as the password, the server accepts the client connection, and sends back a "success" reply.

After confirming the authentication, the server loops, accepting responses until it receives a `QUIT` request or a bad packet. It can receive any (multiple) of the commands below:

ALLO

If the server receives the `ALLO [size]` command, it parses the size sent and saves it. If no size is specified, or if the size is less than one or greater than 65535, the server sets the saved size to be 65535, sends an "invalid size" response over the command

connection, and breaks out of the loop.

If the size is in bounds, it saves the size, sends a "size okay" response over the command connection, and keeps looping.

STOR

If the server receives the `STOR [filename]` command, it checks to see if it has a valid size stored from the `ALLO` command. If so, then it receives the file over the data connection. If it receives no data, the server exits the loop and replies "fail" on the command connection. Once it has the file data, the server opens a file, writes the data, and closes the file. If that succeeds, it replies "okay" on the command connection and continues looping. (If it fails, it replies "fail" on the command connection and exits the loop).

SIZE

If the server receives the `SIZE [filename]` command, the server opens the file, sends back its size over the command connection, and keeps looping. If that fails, it sends back an error on the command connection and exits the loop.

RETR

The server receives the `RETR [filename]` when the client asks to download the file. The server opens the file, reads it into and array, and sends that array back to the client on the data connection. If any of those steps fail, it sends a "fail" response back on the command connection and exits the loop. If they all succeed, the server sends back an "okay" response and continues looping.

QUIT

If the server receives a `QUIT` message, the server notes that the client disconnected with no errors and exits the loop.

Bad Packet

If the server receives any message that is not one of `ALLO`, `STOR`, `SIZE`, `RETR`, or `QUIT`, it replies with "bad packet" and breaks out of the loop. The server does not know what to do with this request, so it closes its connections and returns -1.

After the loop, the server closes the command and data connections. If the server received a `QUIT` message, it returns 0. If it exited the loop for any other reason, it returns -1.

4.4 Usage Instructions

I ran most of my code on my 2014 MacBook Pro running OS X El Capitan, which has 8 gigabytes of memory and a 2.8 GHz Intel Core i5 processor. When I temporarily exceeded the computational power of my laptop, I borrowed a 2015 ThinkPad running Ubuntu 14.4, with and 2.8 GHz Intel Core i7 processor and 20 gigabytes of memory. I only needed to use the ThinkPad for two weeks, until I resolved a bug that caused GRASShopper to require a large amount of my computer's memory (see § 4.7.1 for details on that bug).

All code is available for download at: https://github.com/JaneEyre446/grasshopper

To download it, open a Terminal window and run:

`git clone: https://github.com/JaneEyre446/grasshopper.git`

To run it, install Microsoft's z3, version 3.2 or later (available from [dMB08]), as well as OCaml, version 4.01 or later (available via the computer's package manager).

Once those are installed, `cd` to the grasshopper folder and run: `./build.sh`

To compile the FTP files, run

```
./grasshopper.native -v tests/spl/ftp/server.spl -bitvector -nostratify
    -nomodifiesopt -compile ftpCcode/ftpServer.c
```

to compile the server, and

```
./grasshopper.native -v tests/spl/ftp/client.spl -bitvector -nostratify
    -nomodifiesopt -compile ftpCcode/ftpClient.c
```

to compile the client[1]. The above commands verify the `SPL` code, and then compile it to C code. For the `SPL` and C code, please see the appendices.

To compile and run the client and server, `cd` to the "ftpCcode" directory and run the `build.sh` file, shown in Figure 4-3. When debugging, I passed in the optional `-g` flag to enable debug mode. On Linux machines, I needed to pass in the `-std=gnu99` in order to use the proper version of C.

```
1 gcc -c ${1} ../lib/console.c -o console.o
2 gcc -c ${1} ../lib/file.c -o file.o
3 gcc -c ${1} ../lib/socket.c -o socket.o
4 gcc -c ${1} ../lib/makeStr.c -o makeStr.o
5 gcc -c ${1} ftpClient.c -o ftpClient.o
6 gcc -c ${1} ftpServer.c -o ftpServer.o
7 gcc ${1} console.o file.o socket.o makeStr.o ftpServer.o -o ftpServer
8 gcc ${1} console.o file.o socket.o makeStr.o ftpClient.o -o ftpClient
```

Figure 4-3: The build script to compile the GRASShopper C files to object files

Open two terminal windows. There are two executables in the "ftpCcode" directory: `ftpServer` and `ftpClient`. Copy `ftpServer` to a different directory and run it

```
./ftpServer
```

Go back to the other window and run the client as:

```
./ftpClient
```

Follow the prompts to upload or download a file, as in Figure 4-4. To run it locally, pass in the local IP or pass in "localhost" (see Figure 4-5). If running the client on one computer and the server on another, pass in the IP of the server (as in Figure 4-5a. 127.0.0.1 is the IP of localhost, but the client will work for remote servers as well, given their IP.).

[1]I needed to increase the memory limit for my Terminal

```
➜  test git:(master) ✗ ./ftpServer
➜  test git:(master) ✗
```

(a) The server

```
➜  ftpCcode git:(master) ✗ ./ftpClient
upload (u) or download (d):u
Enter IP of server:127.0.0.1
input the file name:tomato.txt
➜  ftpCcode git:(master) ✗
```

(b) Uploading using the client

```
➜  ftpCcode git:(master) ✗ ./ftpClient
upload (u) or download (d):d
Enter IP of server:127.0.0.1
input the file name:tomato.txt
➜  ftpCcode git:(master) ✗
```

(c) Downloading using the client

Figure 4-4: How to upload and download a file

```
➜  ftpCcode git:(master) ✗ ./ftpClient
upload (u) or download (d):u
Enter IP of server:127.0.0.1
input the file name:tomato.txt
➜  ftpCcode git:(master) ✗
```

(a) Pass in the IP of the server

```
➜  ftpCcode git:(master) ✗ ./ftpClient
upload (u) or download (d):u
Enter IP of server:localhost
input the file name:tomato.txt
➜  ftpCcode git:(master) ✗
```

(b) Pass in "localhost"

Figure 4-5: Various ways to connect the client to the server

4.5 GRASShopper Guarantees

My FTP system provides several guarantees. Some guarantees come from GRASShopper itself, while others come from the pre- and post- conditions I added to the program.

GRASShopper itself guarantees memory safety for all data types. For each memory access, GRASShopper generates a formula that encodes the memory access and the state of the heap. z3 uses that statement to prove the memory access is valid [Zuf16]. For a definition of memory safety, see § 2.1. GRASShopper provides functional correctness for some data types, but not for arrays. Having functional correctness for arrays would be useful, as then GRASShopper could guarantee that the contents of an uploaded or downloaded file were not changed while sent. In order to prove functional correctness, GRASShopper would need to have an annotation preserving the state of the network, so GRASShopper could save the state of the file before and after upload. Currently, GRASShopper has no concept of the network, since annotations only extend to the bounds of the method. Thus, GRASShopper can only guarantee that the size of the file sent or received is the same as the original file's size, and cannot guarantee that the contents are the same. As arrays are already slow to process, Zufferey has decided to focus on speeding up arrays before adding functional correctness [Zuf16]. Please see section § 3.1 for more details.

The FTP client and server make use of many arrays, which I would like to be memory safe. C itself provides no memory safety, so even with only memory safety, GRASShopper gives a large advantage. Thus, GRASShopper promises that my FTP client and server are memory-safe.

4.6 Program Logic Guarantees

I prove FTP-specific guarantees with program logic (see section § 1.1.2 for an explanation of program logic).

All code is available in the Appendix. The code for the socket and console guarantees and functions were written by Damien Zufferey. As I use his code in my thesis,

I include them for completeness.

For the socket, console, and file code, Zufferey and I provide the guarantees in GRASShopper. Since GRASShopper can compile to C, we wrote the implementations of the functions in C, which I link in at compilation using the build script in Figure 4-3. Please see the Appendices for the guarantees on those functions.

4.6.1 Server Guarantees

At a high level, GRASShopper guarantees that the server returns an integer greater than or equal to -1. My program returns -1 upon failure at any point in the server process. If the upload or download is successful, my program returns 0.

Before any return, the server frees all memory that is currently allocated, and closes all file descriptors.

First, the server sets up the command and data file descriptors to communicate with the client. Please see Figure 4-6 for the loop invariants when setting up the command connection. If the server receives an invalid address or a bad file descriptor, it fails and returns -1. If the setup succeeds, it continues.

If the client has invalid credentials to log in to the server, the server fails and returns -1.

After the client has connected and has a valid username and password, the server loops, accepting commands from the client and sending back responses. See Figure 4-7 for the loop invariants on that loop. The invariant for this loop is that both the data and command file descriptors are greater than or equal to zero.

In the loop, the server receives many commands from the client, all of which are verified.

ALLO

The client sends the `ALLO` command to tell the server the size of the incoming file. If no size is specified, the server sends an "invalid size" response and exits the loop. If the size is less than 1 or greater than 65535 (the maximum size of an array in

```
1 while(cmdFd < 0)
2   invariant cmdFd >= -1
3   invariant socket_addr_4(cmdAddr)
4   invariant tempCmdFd >= 0
5 {
6   cmdFd := connectMeCommand(tempCmdFd, cmdAddr);
7 }
```

(a) The loop invariants for the first loop in the server. In this loop, the server sets up the command connection to the client. These loop invariants guarantee that, at the beginning of each loop iteration, the file descriptor for the command connection is greater than or equal to -1, the address of the command connection is of type `socket_addr_4`, and that the temporary command file descriptor is greater than or equal to zero. The only thing to change in the loop should be the command file descriptor, `cmdFd`. However, in order to compile, GRASShopper required invariants for `cmdAddr` and `tempCmdFd`. Inside the loop, the server calls the `connectMeCommand`, which has pre- and post- conditions on the temporary file descriptor and the command address. In order to satisfy those conditions, GRASShopper needed loop invariants for those variables.

```
1  procedure connectMeCommand(cmdFd: Int, cmdAddr: SocketAddressIP4)
       returns (res: Int)
2    requires socket_addr_4(cmdAddr)
3    requires cmdFd >= 0
4    ensures socket_addr_4(cmdAddr)
5    ensures res >= -1
6  {
7    var listening := glisten(cmdFd, 10);
8    if (!listening) {
9      return -1;
10   }
11
12   var connFd := accept4(cmdFd, cmdAddr);
13   return connFd;
14 }
```

(b) The `connectMeCommand` function. This function connects the server to the client over the command connection. It requires that the file descriptor passed in be greater than or equal to zero. It also requires and ensures that the address used for the connection be a socket address.

Figure 4-6: The first loop in the server, and its invariants

```
while (!iQuit)
  invariant cmdFd >= 0
  invariant dataFd >= 0
```

(a) The above loop processes all the commands sent by the client, and quits if it receives the `QUIT` command, or a bad request. The loop invariants guarantee that at the beginning of each iteration, the command and data file descriptors are valid (greater than zero). GRASShopper needs the file descriptors to be valid in order to send and receive data and commands.

```
  var iQuit := false;
  var properQuit := false;
  var allo_size := 65535;

  while (!iQuit)
    invariant cmdFd >= 0
    invariant dataFd >= 0
  {
    var request := new Array<Byte>(150); // it needs to be so big to hold the filename
    var recd := tcp_recv(cmdFd, request);
    var typeCom := copy_byte_slice(request, 0, 4);
    var final := process_string(typeCom);
    var filename := copy_byte_slice(request, 5, (request.length -1));
    free(request);

    if (is_allo(final)) {
      allo_size := allo_help(cmdFd, filename);
      if ((allo_size < 1) || (allo_size > 65535)) {
        allo_size := 65535;
        free(final);
        iQuit := true;
      }
    }
    else if (is_stor(final)) {
      if ((allo_size < 1) || (allo_size > 65535)){
        free(filename);
        free(final);
        iQuit := true;
      }
      else {
        var temp := store_help(cmdFd, dataFd, filename, allo_size);
        if (temp) {
          free(filename);
          free(final);
        }
        iQuit := temp;
      }
    }
    else if (is_size(final)) {
      var temp := size_help(cmdFd, filename);
      if (temp) {
        free(filename);
        free(final);
      }
      iQuit := temp;
    }
    else if (is_retr(final)) {
      var temp := retr_help(cmdFd, dataFd, filename);
      if (temp) {
        free(filename);
        free(final);
      }
      iQuit := temp;
    }
    else if (is_quit(final)) {
      properQuit := true;
      iQuit := true;
    } else {
      var badPacket := new Array<Byte>(4);
      badPacket := "500";
      var sent := tcp_send(cmdFd, badPacket, 4);
      free(badPacket);
      iQuit := true;
      //something we did not expect
    }
    free(filename);
    free(final);
  }
```

(b) The full context for the loop. The server receives commands from the client, and replies. It also sends and receives file data. To do so securely, GRASShopper needs the loop invariants to guarantee that the file descriptors will be valid at all stages of loop execution.

Figure 4-7: The loop invariants for the second loop in the server.

GRASShopper), the server sends an "invalid size" response and terminates execution. Figure 4-8 shows the program logic for the `ALLO` helper functions.

If the size is valid, the server sends a "request okay" response, saves the size, and continues the loop.

```
1 procedure is_allo(cmd: Array<Byte>) returns (is: Bool)
2   requires byte_array(cmd) &*& cmd.length == 5
3   ensures  byte_array(cmd) &*& cmd.length == 5
```

(a) The pre- and post- conditions for `is_allo`, a helper function for `ALLO`. This function checks whether or not the command is `ALLO`. It guarantees that the `cmd` array passed in is a byte array both at the start and end of the program, and that it is of length 5 at the start and end of the program. The function is guaranteed to return a boolean.

```
1 procedure allo_help(cmdFd:Int, sizeB: Array<Byte>) returns (allo_size:
    Int)
2   requires byte_array(sizeB);
3   ensures byte_array(sizeB);
4   requires cmdFd >= 0;
5   ensures allo_size >= -1;
6   ensures allo_size <= 65535;
```

(b) The pre- and post- conditions for `allo_help`, the main helper function for `ALLO`. It requires that the command file descriptor be valid (greater than or equal to zero), and that `sizeB`, the array containing the size of the file, is a byte array. The post conditions guarantee that `sizeB` is still a byte array at the end of the program, and that the returned size (the size of the file to be allocated) is an integer between -1 and 65535. 65535 is the maximum size of the file, and -1 indicates an invalid file size.

Figure 4-8: The program logic for the `ALLO` helper functions.

STOR

The client sends the `STOR` command to tell the server that it wants to upload a file. If the server has received a invalid size for this file from `ALLO`, or has not received an `ALLO` command, then it exits the loop.

If the server has a valid size for the file, then it tries to receive the file. If it receives no data or cannot open a new file to store the incoming file, it sends a "bad request" reply and exits the loop. Next, it tries to write the incoming data to a file. If it cannot write, it sends a "bad request" reply and terminates. If it succeeds in writing the file, it sends a "success" response and continues the loop.

```
1 procedure is_stor(cmd: Array<Byte>) returns (is: Bool)
2   requires byte_array(cmd) &*& cmd.length == 5
3   ensures  byte_array(cmd) &*& cmd.length == 5
```

(a) The pre- and post- conditions for `is_stor`, a helper function for `STOR`. This function checks whether or not the command is `STOR`. It guarantees that the `cmd` array passed in is a byte array both at the start and end of the program, and that it is of length 5 at the start and end of the program. The function is guaranteed to return a boolean.

```
1 procedure store_help(cmdFd: Int, dataFd: Int, filename: Array<Byte>,
    allo_size: Int) returns (fail: Bool)
2   requires cmdFd >= 0;
3   requires dataFd >= 0;
4   requires allo_size >= 1;
5   requires allo_size <= 65535;
6   requires byte_array(filename);
7   ensures byte_array(filename);
```

(b) With the pre-conditions set for `stor_help`, the major `STOR` helper function, GRASShopper guarantees that both the data and command file descriptors are valid (greater than or equal to zero), and that the size of the file to be allocated is a legitimate file size (greater than or equal to 1, and less than or equal to 65535). GRASShopper also ensures that the file name to be stored is a byte array at the beginning and the end of the program. GRASShopper also requires that the function returns a boolean value indicating the success of the program

Figure 4-9: The program logic for the `STOR` helper functions.

SIZE

The client sends the `SIZE` command to ask the server what is the size of the file the client wants to download. If the file does not exist on or cannot be opened by the server, the server sends an "error" response and exits the loop. If the server can open and calculate the size of the file, it sends back the size of the file in the response and continues the loop.

Please see Figure 4-10 for the pre- and post- conditions on the `SIZE` helper functions.

```
1 procedure is_size(cmd: Array<Byte>) returns (is: Bool)
2   requires byte_array(cmd) &*& cmd.length == 5
3   ensures  byte_array(cmd) &*& cmd.length == 5
```

(a) The pre- and post- conditions for `is_size`, a helper function for `SIZE`. This function checks whether or not the command is `SIZE`. It guarantees that the `cmd` array passed in is a byte array both at the start and end of the program, and that it is of length 5 at the start and end of the program. The function is guaranteed to return a boolean.

```
1 procedure size_help(cmdFd: Int, filename: Array<Byte>) returns (fail:
    Bool)
2   requires cmdFd >= 0;
3   requires byte_array(filename);
4   ensures byte_array(filename);
```

(b) The program logic statements for `size_help`, the main helper function for `SIZE`. The program requires that the filename be a byte array at the start and end of the program, and that the command file descriptor be valid (greater than or equal to zero) at the start of the program. `size_help` is guaranteed to return a boolean value reporting the success of the function.

Figure 4-10: The program logic for the `SIZE` helper functions.

RETR

The client sends the `RETR` command when it wants to download a file from the server. The server tries to send the file to the client. If it cannot open, compute the size of, or read the file, or cannot send back the file, it sends an "error" response and exits the loop. Otherwise, it sends the file to the client and continues the loop. See Figure 4-11 for an explanation of the pre- and post- conditions for the `RETR` helper functions.

```
1 procedure is_retr(cmd: Array<Byte>) returns (is: Bool)
2   requires byte_array(cmd) &*& cmd.length == 5
3   ensures  byte_array(cmd) &*& cmd.length == 5
```

(a) The pre- and post- conditions for `is_retr`, a helper function for `RETR`. This function checks whether or not the command is `RETR`. It guarantees that the `cmd` array passed in is a byte array both at the start and end of the program, and that it is of length 5 at the start and end of the program. The function is guaranteed to return a boolean.

```
1 procedure retr_help(cmdFd: Int, dataFd: Int, filename: Array<Byte>)
    returns (fail: Bool)
2   requires cmdFd >= 0;
3   requires dataFd >= 0;
4   requires byte_array(filename);
5   ensures byte_array(filename);
```

(b) The pre- and post- conditions for `retr_help`, the major helper function for `RETR`. It requires that both the command and data file descriptors be valid (greater than or equal to zero), and that the file name be a byte array at the beginning and end of the function. GRASShopper also requires that the function return a boolean, guaranteed by GRASShopper.

Figure 4-11: The program logic for the `RETR` helper functions.

QUIT

The client sends a `QUIT` response when it is done uploading or downloading a file and would like to disconnect. Upon receipt of a quit message, the server notes that it received the message and exits the loop. I illustrate the program logic for `QUIT` in Figure 4-12.

```
1 procedure is_quit(cmd: Array<Byte>) returns (is: Bool)
2   requires byte_array(cmd) &*& cmd.length == 5
3   ensures byte_array(cmd) &*& cmd.length == 5
```

Figure 4-12: The pre- and post- conditions for `is_quit`, the helper function for the `QUIT` command. This function checks whether or not the command is `QUIT`. It guarantees that the `cmd` array passed in is a byte array both at the start and end of the program, and that it is of length 5 at the start and end of the program. The program is guaranteed to return a boolean, indicating whether or not the command is `QUIT`.

Bad Packet

If the server receives any message while in the loop that is not one of the above messages, it sends a "bad packet" response and exits the loop. In this case, the server has received a message that it does not know how to handle, so it exits the loop and shuts down.

Upon exiting the loop, the server closes all file descriptors. If it has received a `QUIT` message, it returns 0. Otherwise, it returns -1.

4.6.2 Client Guarantees

GRASShopper guarantees that the client will return an integer greater than or equal to -1. It returns 0 on success. First, the client asks if the user would like to upload or download a file. If the user inputs something else, the client returns -1 and quits. Otherwise, the client asks for the IP address of the server. If the user inputs an invalid IP address, the client returns -1 and quits. If the user types a valid IP address, the client sets up the connections to the server. If these connections fail, the client quits. If the client successfully connects, it tries to authenticate to the server. If it fails to authenticate, it returns 4 and quits. If it is successful, the client asks the user to input the filename of the file to upload or download. If the filename is invalid, the client returns 5 and quits.

At this point, the client calls one of two helper functions: upload or download.

Upload

For uploading a file, the client requires that the filename byte array length be less than or equal to 65529, and that the command and data file descriptors be greater than or equal to 0. Upon completion, it guarantees that the filename array is still a byte array. It promises to return a boolean.

First, the upload program computes the size of the file. If the file size is less than zero or greater than 65535, it returns false.

Next, it sends an `ALLO` command to the server, along with the size of the file. If

it fails to send, or if the server sends back a failure response, it returns false.

If that succeeds, it opens and reads the file. If it fails to open or read the file, it returns false.

The upload program then sends the file. If the server response is invalid (not one of `200`, `226`, or `250`), the upload program returns false. Otherwise, it closes the file. If the amount of sent data equals the size of the file, the upload program returns true. If not, then it returns false. Please see Figure 4-13 for the program logic statements written for the major file upload helper function. The program logic statements for the other upload helper functions are in the Appendix.

```
1 procedure uploadFile(cmdFd: Int, dataFD: Int, filename: Array<Byte>)
2   returns (success: Bool)
3   requires cmdFd >= 0 && dataFD >= 0
4   requires byte_array(filename) &*& filename.length <= (65535 - 6)
5   ensures byte_array(filename)
```

Figure 4-13: The pre- and post- conditions for `uploadFile`, the main helper function used to upload a file from the client to the server. GRASShopper requires that both the command and data file descriptors be valid (greater than or equal to zero) at the start of the program. GRASShopper also insists that the filename be a byte array at the beginning and end of the function. The client adds the pre-condition that the filename's length must be less than or equal to $65535 - 6$. The client sends `STOR [filename]` over the command connection to the server. Since the maximum length of an array is 65535, and the client adds 6 characters in the `STOR` command, the client caps the filename length at $65535 - 6$. GRASShopper also requires that the function return a boolean indicating the success of the upload.

Download

The download function requires that the filename be a byte array and ensures that it is still a byte array upon completion. It also requires that both file descriptors be greater than or equal to zero. It promises to return a boolean.

If the filename array length is less than or equal to 0 or greater than or equal to 65529, it returns false.

Then, the download function calculates the size of the file to download from the server. If the size is less than zero, it returns false.

The download function asks the server to send it the file via a `RETR` request. If the server replies with `200` or `150` (success), then execution continues. If not, the download program returns false.

The download program receives the file data. If the received data is less than zero, the download function returns false.

After receiving the file data, the download program receives the server response. If the response is `200`, `226`, or `250`, (okay), the function continues. If not, it returns false.

Next, the download program opens and writes the file corresponding to the filename. If it cannot open or write the file, it returns false. But, if it succeeds in writing the file, it closes the file descriptor and returns true.

If upload or download returned false, the client closes the file descriptors and returns 6. Otherwise, it closes the data file descriptor. If that fails, it returns 22. If not, it continues and sends a `QUIT` message to the server.

See Figure 4-14 for the program logic statements in the client's download helper function. The program logic statements for the rest of the download helper functions are in the Appendix.

```
1 procedure downloadFile(cmdFd: Int, dataFD: Int, filename: Array<Byte>)
2   returns (success: Bool)
3   requires cmdFd >= 0 && dataFD >= 0
4   requires byte_array(filename)
5   ensures byte_array(filename)
```

Figure 4-14: The guarantees on the client's main download helper function. The client requires that the filename be a byte array at the start and end of the function, and that the function return a boolean. The client also requires that the command and data file descriptors be valid (greater than or equal to zero) at the start of the program.

After Upload or Download

The client does not check if the server receives the `QUIT` message. The client sends the `QUIT` message because it is good practice. Either way, the client closes the command file descriptor and shuts down. The server does not reply to a `QUIT` message. If

Number of Lines	Server	Client
Comments	10	32
Pre-conditions	30	22
Post-conditions	20	16
Return Requirements	17	12
Loop Invariants	5	0
Execution Instructions	373	308

Table 4.1: Distribution of lines of code in my client and server. Note that here I only count the code in client.spl and server.spl, and not in any of files included by client.spl and server.spl.

I allowed multiple clients to connect to a server, or for the client to send or receive multiple files, then each client would need to send a `QUIT` message, as the server would need a way to tell when the client has finished sending and receiving files. However, I only allow one client to connect to the server, and to only upload or download one file, so the `QUIT` command is not strictly necessary. Next, the client closes the command file descriptor. If it fails to close, the client returns 21. Otherwise, the client returns 0 on success.

Above, I give the end-to-end guarantees on the client and server. For detailed guarantees on each function used, please see the Appendices.

4.6.3 Code Setup

I spent the first semester of my thesis researching program verification and learning how to use GRASShopper via the SNTP code. I spent January through mid-April writing the FTP code, and mid-April through May writing my thesis.

In my FTP code, I wrote execution instructions, commends, and verification conditions. Please see Figure 4.1 for a distribution of how much of the code I wrote was conditions versus actual execution instructions.[2] For the purposes of analyzing how long each section took, I will treat time spent writing comments as trivial. Between the lines of actual code ("Execution Instructions") and verification conditions ("Pre-

[2]The "Comments" section is an estimate, as some comments are on the same line as a line of code, while others are on their own line

conditions," "Post-conditions," "Return Requirements,"[3] and "Loop Invariants"), the bulk of the initial work lay in solving the verification conditions. I needed to know exactly what guarantees each function in my code would provide. Next, I wrote the initial, un-debugged code. Writing the code took a substantial amount of time, as I had never written an FTP system before, and needed to figure out how to do so. I spent about January to mid February writing my code. I spent mid-February to mid-April debugging the code. For this debugging period, I would add or fix a feature in my execution code, and then need to update my verification conditions and execution code such that the verification conditions were satisfied. In most software projects, the bulk of the work is spent debugging execution code. However, since I also had to satisfy verification conditions, I estimate that a third of my time was spent debugging my conditions, and the other two-thirds fixing my actual code.

4.7 Problems and Solutions

I encountered three main categories of bugs while working on the project: bugs in GRASShopper, bugs in the C wrappers, and bugs not checked by GRASShopper. For bugs in GRASShopper, I found the bugs and referred them to Zufferey, who fixed them. For bugs in the C wrappers and bugs not checked by GRASShopper, I found the bugs and corrected them.

4.7.1 Bugs in GRASShopper

I found two main bugs in GRASShopper. The first was that when importing an `SPL` file into another `SPL` file, GRASShopper looked up the file via relative paths from the /tests directory. I found this when I tried to import a file from another directory not in the "/tests" directory tree, and GRASShopper could not find it. To fix this, Zufferey changed GRASShopper's import to use the absolute path of the file, as opposed to the relative path.

[3]Return requirements can be counted as post-conditions, but I count them as separate categories as they have different GRASShopper syntax

The other problem I found was that, when compiling a large `SPL` file (like the client), GRASShopper would run out of memory and freeze. A temporary workaround which Zufferey suggested was to increase the memory limit in my Terminal and split the client file into multiple, smaller functions. I did so, and switched to using a different computer with 20 GB of memory[4], which fixed the problem, although GRASShopper still ran slowly (it took around fifteen minutes to process the client file). Later, Thomas Wies fixed the underlying problem. GRASShopper was opening new file descriptor pipes to z3 for every verification condition in the program, and leaving them open until the whole file was processed. Wies fixed it so that GRASShopper closed each pipe after it was done checking that condition. Now, GRASShopper runs in a few minutes at most.

4.7.2 Bugs in C Wrappers

One bug I found in the C wrapper was that ints were converted to chars. In GRASShopper, we have int arrays, and byte arrays. Int arrays store ints, and byte arrays store bytes. Strings and characters are stored in byte arrays. When the client asks the server what size a file is via the `SIZE [filename]` command, the server needs to reply with a `213 [size]` response. `[size]` is an integer, but needs to be stored in the byte array containing `213`. In essence, I want to append an integer to a string. GRASShopper has a built-in `int2byte` method, which casts the integer to its byte form. In most cases, this conversion is fine. However, `int2byte` does not handle ASCII values. If one wants the integer to be the actual value of the integer in character form (as in 5 is the character '5'), one needs to append '0' to the integer, making it a character. Initially, I used `int2byte` to cast the file size to a byte, which gave its byte value, and then the client would allocate an array of size [byte value], which is not the same size as the actual file size. I switched to using `makeStr`, a function I wrote to append '0' to an integer, and sent the `makeStr`'d value (a character) to the client. Then, the client had the actual size of the file, and could allocate an array of the proper size. This bug was difficult to uncover, as the allocating of the file was

[4]My computer has 8 GB of memory

working correctly, except that the size was wrong. But, I uncovered the underlying problem and was able to fix it. Please see Figure 4-15 for the `makeStr` code.

```
1 procedure makeStrFromInt(myInt: Int) returns (myByte: Byte)
```

(a) The SPL code for `makeStr`

```
1 #include <stdlib.h>
2
3 int makeStrFromInt(int myInt){
4   char c = myInt + '0';
5   return c;
6 }
```

(b) The C code for `makeStr`

Figure 4-15: The `SPL` and C code to convert an integer into its corresponding character value. First I wrote the pre- and post- conditions for the function in `SPL`, and then wrote the corresponding C implementation. At compile time, I link in the C file with the build script.

Another problem I had with the C wrapper was that there was no way to read or write to the command line. In FTP, the user needs to input the filename and server IP address, along with whether to upload or download the file. The user writes all this information to the command line, which the client needs to read. To fix this, Zufferey added `ggets` (to read) and `gputs` (to write) to the trusted code base, which is linked in when compiling the C code. With them, I can write text to the command line and read in text from the command line.

4.7.3 Bugs Not Checked by GRASShopper

Most of my more persistent bugs fall into this category: bugs that are not checked by GRASShopper. The first bug I ran into was opening my files with the wrong flag. When I opened a file to write to it, I opened it with the flag `O_RDONLY`, the read-only flag. Thus, I could not write to the file. And, when I opened a file to read it, I opened it with the `O_WRONLY` flag, the write-only flag. I had confused the flags. I fixed it by switching the flags such that, when I wanted to write to a file, I opened it with `O_WRONLY`, and when I wanted to read a file, I opened it with `O_RDONLY`. It was an

easy bug to fix, but it was hard to catch. I caught it once I realized that I could not read files that I should have been able to read, and rechecked all my file permissions.

The other puzzle I needed to figure out was how to overwrite files. When opening a file to write, I want to overwrite any previous contents in the file. If I have a file "potato.txt" that contains "Hello, I am a potato," and I want to write the array "Tomato", I need to be sure to clear the previous contents of the file. If I do not, when I write, the "potato.txt" file will contain "Tomato I am a potato". I would like it to only contain "Tomato". In order to ensure that behavior, I needed to open the file for writing with the `O_TRUNC` file in addition to `O_WRONLY`. `O_TRUNC` truncates the file when opening, wiping its contents so I can overwrite them. Zufferey helped me find the `O_TRUNC` flag in the C open spec [IG04].

Another problem I had that was not checked by GRASShopper was that, when the client uploaded a file, the server would allocate the same size (a large size) array to receive the file data. The server did not know what size the incoming file was, so it allocated the largest possible size array to hold the data. It would receive the incoming file data, and then write the entire array–including the excess memory left unwritten by the client–into the file. Thus, my files would contain the sent file contents, along with a slew of nonsense characters written from the excess array space (see Figure 4-16).

The server needed to know the size of the file that was being uploaded. Thus, I had the client send the `ALLO [size]` command to the server before sending the actual file data, where `[size]` is the size of the file being uploaded. That way, the server could allocate an array of the appropriate size `[size]`, and receive the sent file data into that array. The server would then null-terminate the data, in case it was not already so, and write the file. Since the array is completely filled, the written file would contain only the desired file data, without the excess nonsense characters (as in Figure 4-17). At my first iteration of FTP, I tried to have only the bare minimum commands. I left out `ALLO [size]`, thinking it to be unnecessary, but later reread the FTP specification and added it back in once I realized I needed it [PR85].

The last bug that I found and solved was that I could not run the client and server

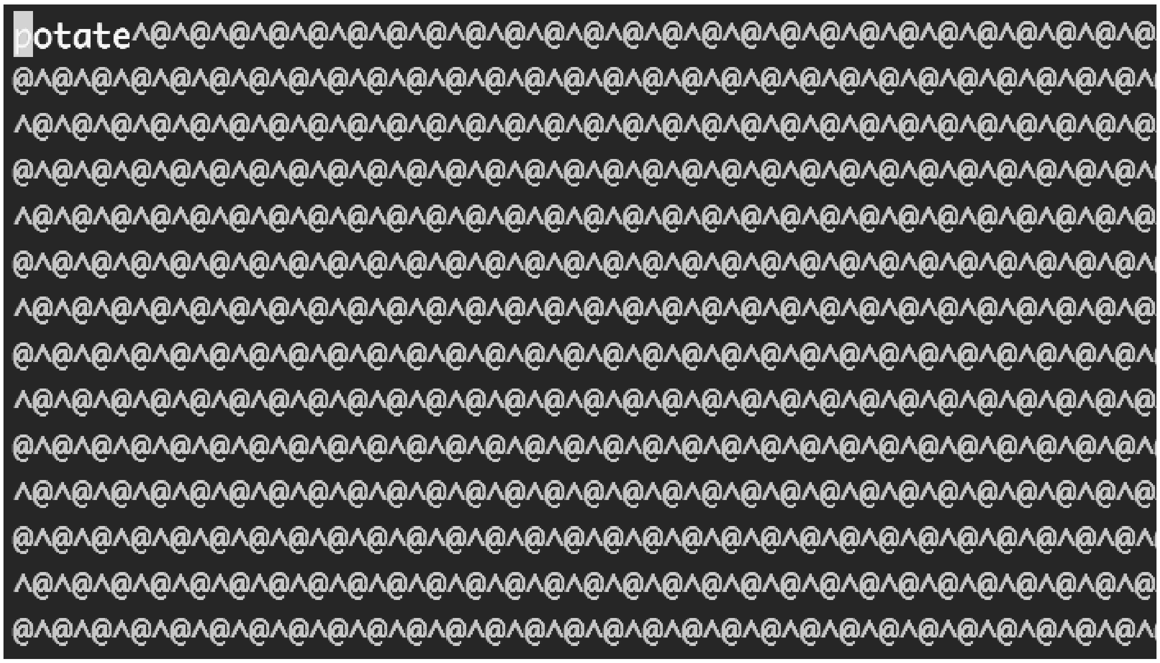

Figure 4-16: A screenshot of a file "potato.txt" with the file's intended contents, "potate", with the resulting unwritten memory nonsense written after it.

Figure 4-17: A screenshot of "potato.txt" with the file's intended contents, "potate". Since the array is allocated to the proper size before writing, the server does not write any nonsense.

on separate machines. Originally, I thought that I could not run them separately because my internet router had blocked port forwarding. However, it turned out that actually, I had made several errors when setting up the command and data connections between the client and server. My first mistake was to hardcode in localhost as the IP address for the client and server. I fixed my system to allow remote addresses, but it still did not work. At this point, I wrote it off as failing because of port forwarding. After speaking with Professor Rinard, he suggested I take a second look with Zufferey. With Zufferey, I figured out that I had also cast the connection ports to the incorrect format. We switched the format, but my system still only worked locally. After careful debugging, Zufferey and I realized that the command connection worked, but the data connection was set up incorrectly. When setting up the data connection, I had intended for the client to accept a connection from the server. However, I had written the client to wait for a connection from the server, but the server to wait for a connection from localhost. When both the client and server ran locally, both had the address "localhost." However, when I ran the server remotely, "localhost" gave the address of the server, and not the client. In essence, the server tried to connect to itself for the data connection, when it should connect to the client. I rewrote the server to connect to the client's IP address instead of"localhost" when setting up the data connection, and then the client and server could connect to each other when running on separate machines.

Once I rewrote the server, I was able to connect the client running on one computer to the server running on another computer. I could upload and download files from one computer to the other, with no errors.

Chapter 5

Future Work

If I were to continue my project, there are a few small changes I would make to the system to fix the remaining bugs. There are also a number of improvements and tests that I suggest anyone who decides to continue the project should implement. Those are outlined below, with some explanations and suggestions.

5.1 Extensions to my FTP System

While I solved most of the problems that I found in my system, there is still one that remains to be fixed.

Currently, the server can only connect to one client at a time. Which is fine, but in industry, a server needs to be able to handle multiple clients connecting at once. I ran out of time before I could modify the server to support multiple client connections. If I had more time, I would implement multiple client connections by forking off a new process from the server for each client connection. This new forked process would handle the upload and download of a file from the client, and then return to the main server process once the client disconnected. With this, the server could handle many more connections, limited only by the number of threads it would be allowed to spawn.

5.2 Possible GRASShopper Features

Two features that I feel would greatly improve GRASShopper are interrupt handling and file descriptor verification. While running the client and server, if I wanted to stop the client or server mid-execution, I would interrupt the binary while running on the command line. I used `CMD+C` on my Mac. (`CTRL+C` on a Linux machine). `CMD+C` stops the program, and if GRASShopper supported interrupt handling, would allow the program to execute its interrupt code before exiting. With the interrupt code, the program could close its sockets and file descriptors before quitting. However, GRASShopper does not support interrupt handling. Thus, when I or the user type `CMD+C`, the program immediately stops, leaving all of its sockets and file descriptors open. Since they are left open, when I try to run the client and server program again, they are still open, and cannot be reused. At this point, the only solution is to restart my computer and try again. Restarting my computer works fine, but it is time-consuming and frustrating. If GRASShopper supported interrupt handling, then, on receiving an interrupt, I could have the client and server close all sockets and file descriptors and exit.

GRASShopper has pre- and post- conditions for arrays and many of its other data types. These conditions allow the writer to check to make sure the array is not modified during execution, that the array length is the same throughout the program, and other such conditions. At the time of writing, there are no such checks for file descriptors. File descriptors have two states: open or closed. In my code, I close all file descriptors before exiting the program, and make sure all file descriptors are open before I write to them. But, if I miss a check, GRASShopper does not notice. I would only notice when the C executable breaks during run time. If file descriptors were made into a data type (right now, they are an integer) with an attribute "open," then one could add pre- and post- conditions to one's programs stating that the file descriptors used needed to be open before use and not open (closed) at the end of the function.

Currently, there is no GRASShopper cryptography library, be that in SPL or

linked in from C. If one were to add support for a crypto library, then I could encrypt the file data while sending it via FTP. I could also have a secure username and password setup for the client and server. Currently, I support anonymous login only, as it was simpler and I had no cryptography library in GRASShopper. With a secure way to store and lookup a user's password, one could implement a secure login FTP server. It would not be hard to add with a cryptography library. The server would store the users' passwords' hashes in a hash table, and a user would only be able to login if their password's hash were in the table. I considered adding a cryptography library while writing my thesis. However, that would require not only linking in a C implementation for a cryptography library and writing verification conditions for those functions, but also adding support for hash tables in GRASShopper. All these additions, while useful, seemed unnecessary for my project. Since my FTP program is more a proof of concept for FTP in GRASShopper, I omitted these additions.

One possible extension to my project would be to rewrite more of the trusted code base in `SPL`. By implementing C's `read` in GRASShopper, I would be able to put guarantees on its internals, and confirm that it verifies. While writing `read` in `SPL` would be a complex and interesting project, as the goal of the project is to write networking utility functions in GRASShopper, I focused my efforts on writing other networking programs. And, even if I rewrote `read` and other functions, I would still have to assume some trusted code base in C, so it is left for a later project. Technically, one could write an entire operating system in GRASShopper, and shrink or eliminate this subset of assumed functions. Alas, that is outside the scope of a Master's Thesis.

5.3 Additional Tests

I would like to test my client and server implementations with someone else's version of an FTP client and server. I am curious to know if my client will work with a server other than my own, and likewise, if my server will work with another's client. I suspect that my client will be able to communicate with another server, as it sends

the standard commands and handles file data as expected. I hope that my server will work, as it uses the standard commands. But, as it only supports one client connection at a time, it would not be able to substitute for an industry server at this time. I would need to add support for multiple client connections first.

I tried to have my client connect to several implementations of an FTP server. Unfortunately, my client could not connect. In my system, I have the client connect to a server over port 4444 for the command connection, and port 4440 for the data connection. By using two ports, I mimicked the server's setup, and also made it easier to place verification conditions on the connections. According to the FTP standard, the client by default will run both connections over the same port, although it is not required [PR85]. Most server implementations I found on the web expected the client to connect over the same port, so my implementation did not work. I would need to change the client's ports to connect. Changing them would be manageable but difficult, as I would have to restructure my verification conditions.

I also attempted to connect my server to other implementations of an FTP client. According to the FTP standard, the server's command connection should be over port 21, and its data connection over port 20 [PR85]. For ease of testing, I changed those ports to 4444 and 4440, respectively. However, most other clients expect the server to be using ports 21 and 20, and thus, could not connect to my server.

If I were to continue the project, I would write another version of the client that had both connections to the server over the same port. I would also have the server use the standard FTP server ports.

Chapter 6

Conclusion

Through my project, I gained a better understanding of program verification and the File Transfer Protocol. I wrote a memory-safe FTP client and server. The client and server are both guaranteed to be correct, due to the pre- and post- conditions that I incorporated in the `SPL` code. Because of these program logic guarantees, my FTP client and server are secure and correct, and with some extensions to improve scalability and usability, could be used on a large scale to provide an alternative to the current system.

Bibliography

[BKW+07] Charles Bouillaguet, Viktor Kuncak, Thomas Wies, Karen Zee, and Martin C. Rinard. Using first-order theorem provers in the jahob data structure verification system. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings*, volume 4349 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2007.

[BM07] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer Science & Business Media, 2007.

[Cod] Codenomicon. The heartbleed bug. `http://heartbleed.com/`. Accessed: 2016-05-01.

[Dat15] National Vulnerability Database. Vulnerability summary for cve-2014-7186. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7186`, 2015.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[Fac] Facebook. Infer. `http://fbinfer.com/`. Accessed: 2016-05-01.

[Han15] Ask Bjørn Hansen. NTP pool project. `http://www.pool.ntp.org/en/`, 2015.

[HHK+15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath T. V. Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17. ACM, 2015.

[HHL+14] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 165–181. USENIX Association, 2014.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[IG04] The IEEE and The Open Group. Open. `http://pubs.opengroup.org/onlinepubs/009695399/functions/open.html`, 2004.

[ISI81] Jon Postel (Editor) Information Sciences Institute. Rfc 793: Transmission control protocol. `https://www.ietf.org/rfc/rfc793.txt`, September 1981.

[JSP+11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[Kno07] Douglas Knowles. W32.sqlexp.worm. `https://www.symantec.com/security_response/writeup.jsp?docid=2003-012502-3306-99`, 2007.

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[lub14] lubia. sntp: A[sic] implementation of ntp server with golang. `https://github.com/lubia/sntp`, 2014.

[Mic03] Microsoft. Unchecked buffer in index server isapi extension could enable web server compromise. `https://technet.microsoft.com/library/security/ms01-033`, 2003.

[Mil06] D. Mills. Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI. `https://tools.ietf.org/html/rfc4330`, 2006.

[MSS16] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In B. Jobstmann and K. R. M.

Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[Pos80] J Postel. Rfc 768: User datagram protocol. `https://www.ietf.org/rfc/rfc768.txt`, August 1980.

[PR85] J. Postel and J. Reynolds. File transfer protocol (FTP). `https://www.ietf.org/rfc/rfc959.txt`, 1985.

[PWZ13] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using SMT. In *Computer Aided Verification*, pages 773–789. Springer, accessible: `http://cs.nyu.edu/wies/publ/automating_separation_logic_using_smt.pdf`, 2013.

[PWZ14] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Grasshopper: Complete heap verification with mixed specification. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–139. Springer, accessible: `http://cs.nyu.edu/wies/publ/grasshopper.pdf`, 2014.

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002.

[Sup09] Microsoft Support. Microsoft internet information services 7.0: The ftp 7.0 status codes in iis 7.0. `https://support.microsoft.com/en-us/kb/969061`, June 23 2009. Article ID: 969061.

[Zuf16] Damien Zufferey. Grasshopper guarantees personal communication. Email, April 2016.

Appendix A

SPL code

A.1 Full SPL functions

Below is the code used to ask the user for the IP address of the server, which I wrote.

```
1  include "../../../lib/console.spl";
2  include "../array/copy_byte_slice.spl";
3  include "../array/string.spl";
4  include "../../../lib/socket.spl";
5
6  procedure askIP(port: Array<Byte>)
7    returns (ip: SocketAddressIP4)
8    requires byte_array(port)
9    ensures byte_array(port)
10   ensures port == old(port) && port.length == old(port.length)
11   ensures ip == null || socket_addr_4(ip)
12 {
13   var writtenPrompt := new Array<Byte>(20);
14   writtenPrompt := "Enter IP of server:";
15
16   var written := gputs(writtenPrompt);
17   free(writtenPrompt);
18   var getip := new Array<Byte>(50);
19
20   var numChars := ggets(getip);
```

```
21    var dataAddr: SocketAddressIP4 := null;
22    if (numChars < 1) {
23      free(getip);
24      return dataAddr;
25    }
26    getip[numChars-1] := int2byte(0);
27    if (numChars != 1){
28      dataAddr := get_address4(getip, port);
29    } else {
30      dataAddr := get_address4(null, port);
31    }
32    free(getip);
33    return dataAddr;
34  }
```

Below is the code used to ask the user whether she wants to upload or download a file, which I wrote.

```
1  include "../../../lib/console.spl";
2  include "../array/string.spl";
3
4  procedure doWeUpload()
5    returns (res: Int)
6    ensures (res == -1) || (res == 1) || (res == 0)
7  {
8
9    var updown := new Array<Byte>(30);
10
11   var text := new Array<Byte>(28);
12   text := "upload (u) or download (d):";
13   var written := gputs(text);
14   free(text);
15   if (!written) {
16     free(updown);
17     return -1;
18   } else {
19     var numChars := ggets(updown);
20
21     if (numChars == 2) {
22       var copy := new Array<Byte>(2);
23       copy[0] := updown[0];
24       copy[1] := int2byte(0);
25       free(updown);
26       var uChar := new Array<Byte>(2);
27       uChar := "u";
28
29       if (gstrcmp(uChar, copy) == 0) {
30         free(uChar);
31   free(copy);
32   return 1;
33       } else {
34   free(uChar);
```

```
35      free(copy);
36      return 0;
37          }
38        } else {
39          free(updown);
40          return -1;
41        }
42      }
43  }
```

Below is the code used by the client to process responses from the server, which Zufferey and I wrote.

```
1  include "../array/string.spl";
2  include "../array/int_array.spl";
3
4  procedure checkServerResponseUSER(response: Array<Byte>)
5    returns (success: Bool)
6    requires byte_array(response)
7    ensures byte_array(response)
8  {
9    var ack := atoiG(response);
10   if ((ack == 200) ||
11       (ack == 230) ||
12       (ack == 234) ||
13       (ack == 331) )
14     success := true;
15   else
16     success := false;
17   return success;
18 }
19
20 procedure checkServerResponsePASS(response: Array<Byte>)
21   returns (success: Bool)
22   requires byte_array(response)
23   ensures byte_array(response)
24 {
25   var ack := atoiG(response);
26   if ((ack == 200) ||
27       (ack == 202) ||
28       (ack == 230) ||
29       (ack == 234))
30     success := true;
31   else
32     success := false;
33   return success;
34 }
```

```
35
36 procedure checkServerResponse_200(response: Array<Byte>)
37   returns (success: Bool)
38   requires byte_array(response)
39   ensures byte_array(response)
40 {
41   var ack := atoiG(response);
42   if (ack == 200)
43     return true;
44   else
45     return false;
46 }
47
48 procedure checkServerResponse_213(response: Array<Byte>)
49   returns (success: Bool)
50   requires byte_array(response)
51   ensures byte_array(response)
52 {
53   var ack := atoiG(response);
54   if (ack == 213)
55     return true;
56   else
57     return false;
58 }
59
60 procedure checkServerResponse_200_150(response: Array<Byte>)
61   returns (success: Bool)
62   requires byte_array(response)
63   ensures byte_array(response)
64 {
65   var ack := atoiG(response);
66   if ((ack == 200) ||
67       (ack == 150))
68     success := true;
69   else
70     success := false;
```

```
71    return success;
72 }
73
74 procedure checkServerResponse_200_226_250(response: Array<Byte>)
75    returns (success: Bool)
76    requires byte_array(response)
77    ensures byte_array(response)
78 {
79    var ack := atoiG(response);
80    if ((ack == 200) ||
81        (ack == 226) ||
82        (ack == 250))
83      success := true;
84    else
85      success := false;
86    return success;
87 }
```

Below is the FTP client code, which I wrote.

```
1  include "../../../lib/socket.spl";
2  include "../../../lib/file.spl";
3  include "../../../lib/console.spl";
4  include "../array/string.spl";
5  include "response.spl";
6  include "doWeUpload.spl";
7  include "../array/copy_byte_slice.spl";
8  include "askIP.spl";
9
10 procedure connectTo(addr: SocketAddressIP4)
11   returns (fd: Int)
12   requires socket_addr_4(addr)
13   ensures socket_addr_4(addr)
14   ensures fd >= -1
15 {
16   fd := create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
17   if (fd == -1) {
18     return -1;
19   }
20
21   if (connect4(fd, addr)) {
22     return fd;
23   } else {
24     return -1;
25   }
26 }
27
28 procedure setupDataConnection(cmdFd: Int, address: SocketAddressIP4,
       port: Array<Byte>)
29   returns (connectedDataFD: Int)
30   requires cmdFd >= 0
31   requires port.length == 5;
32   requires byte_array(port)
33   requires socket_addr_4(address)
34   ensures byte_array(port)
```

```
35    ensures port == old(port) && port.length == old(port.length)
36    ensures connectedDataFD >= -1
37    ensures socket_addr_4(address)
38  {
39
40    var dataFD := create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
41    if (dataFD == -1){
42      return -1;
43    }
44    var bound := bind4(dataFD, address);
45    if (!bound) {
46      return -1;
47    }
48    var datalistening := glisten(dataFD, 10);
49    //we don't need a big backlog here, as the only connection should be
          to the server
50    if (!datalistening){
51      return -1;
52    }
53    //we tell the server what port we are setting the data connection on
54    if ((port.length < 0) || (port.length > (65535 - 6))){ //this is to
          cover for the max length of an array, as set in byte_array.spl  We
           subtract 6 to account for PORT
55      return -1;
56    }
57    var portMsg := new Array<Byte>(6 + port.length);
58    portMsg := "PORT ";
59    var copied := gstrcat(port, portMsg);
60    var sent := tcp_send(cmdFd, portMsg, portMsg.length);
61    if (sent != portMsg.length) {
62      free(portMsg);
63      return -1;
64    }
65    free(portMsg);
66
67    //now we wait for the server to connect to us on the data connection
```

```
68    connectedDataFD := accept4(dataFD, address);
69    var closeData := gclose(dataFD);
70    return connectedDataFD;
71  }
72
73  procedure authenticate(cmdFd: Int)
74    returns (success: Bool)
75    requires cmdFd >= 0
76  {
77    var userMsg := new Array<Byte>(12);
78    userMsg := "USER potato"; //it's anonymous login, so you can put
          whatever here.  It's not checked
79    var sent := tcp_send(cmdFd, userMsg, 12);
80    free(userMsg);
81    //receive confirmation by the server
82    var okMsg := new Array<Byte>(4);
83    var ok := tcp_recv(cmdFd, okMsg);
84    //valid responses are 200, 230, 234, 300, 331
85    var checked := checkServerResponseUSER(okMsg);
86    free(okMsg);
87    if (!checked) {
88      return false;
89    }
90
91    var passMsg := new Array<Byte>(15);
92    passMsg := "PASS anonymous";
93    sent := tcp_send(cmdFd, passMsg, 15);
94    free(passMsg);
95    //receive confirmation by the server.  Acceptable responses include
          200, 230, 202, 234
96    okMsg := new Array<Byte>(4);
97    ok := tcp_recv(cmdFd, okMsg);
98    checked := checkServerResponsePASS(okMsg);
99    free(okMsg);
100   if (!checked) {
101     return false;
```

```
102   }
103
104   return true;
105 }
106
107 procedure askFilename()
108   returns (fn: Array<Byte>)
109   ensures fn == null || (byte_array(fn) &*& fn.length == 100)
110 {
111   var filename := new Array<Byte>(100); //we decide max filename length
          is 100
112
113   var text := new Array<Byte>(24);
114   text := "input the file name:";
115   var putted := gputs(text);
116   free(text);
117   if (!putted) {
118     free(filename);
119     return null;
120   }
121   var numChars := ggets(filename);
122
123   if (numChars >= 100 || numChars <= 1){ //switched to 1 to catch the
          newline character at the end of the line
124     free(filename);
125     return null;
126   }
127   filename[numChars -1] := int2byte(0);
128   return filename;
129
130 }
131
132 procedure allo_help(size:Int, cmdFd:Int) returns (success:Bool)
133   requires cmdFd >= 0
134   requires ((size >= 0) && (size <= 65535))
135 {
```

```
136
137   var alloSize := new Array<Byte>(2);
138   alloSize[0] := int2byte(size);
139   alloSize[1] := int2byte(0);
140   var sendMsg := new Array<Byte>(6);
141   sendMsg := "ALLO ";
142   var copied := gstrcat(alloSize, sendMsg);
143   var sent := tcp_send(cmdFd, sendMsg, 7);
144   free(sendMsg);
145   free(alloSize);
146   //Get confirmation from the server
147   var okMsg := new Array<Byte>(4);
148   var ok := tcp_recv(cmdFd, okMsg);
149
150   var checked := checkServerResponse_200(okMsg);
151   free(okMsg);
152   if (!checked) {
153     return false;
154   }
155   return true;
156 }
157
158 procedure store_send_help(cmdFd: Int, filename: Array<Byte>) returns (
      success: Bool)
159   requires cmdFd >= 0
160   requires byte_array(filename) &*& filename.length <= (65535 - 6)
161   ensures byte_array(filename)
162 {
163   var commandSize := filename.length + 6;
164   var sendMsg := new Array<Byte>(commandSize);
165   sendMsg := "STOR ";
166   var copied := gstrcat(filename, sendMsg);
167   var sent := tcp_send(cmdFd, sendMsg, commandSize);
168   free(sendMsg);
169   //Get confirmation from the server
170   var okMsg := new Array<Byte>(4);
```

```
171   var ok := tcp_recv(cmdFd, okMsg);
172
173   var checked := checkServerResponse_200_150(okMsg);
174   free(okMsg);
175   if (!checked) {
176     return false;
177   }
178   return true;
179 }
180
181 procedure uploadFile(cmdFd: Int, dataFD: Int, filename: Array<Byte>)
182   returns (success: Bool)
183   requires cmdFd >= 0 && dataFD >= 0
184   requires byte_array(filename) &*& filename.length <= (65535 - 6)
185   ensures byte_array(filename)
186 {
187   var size := fileSize(filename);
188   if ((size < 0) || (size > 65535)){
189     return false;
190   }
191
192   var allo_check := allo_help(size, cmdFd);
193   if (!allo_check){
194     return false;
195   }
196
197   var opened := gopen(filename, O_CREAT | O_RDONLY);
198   if (opened < 0){
199     return false;
200   }
201   var buffer := new Array<Byte>(size);
202   var read := gread(opened, buffer);
203   if (read < 0){
204     free(buffer);
205     return false;
206   }
```

```
207
208   var stored := store_send_help(cmdFd, filename);
209   if (!stored){
210     free(buffer);
211     return false;
212   }
213
214   var sentData := tcp_send(dataFD, buffer, size);
215   //get confirmation from the server (200, 226, 250)
216   var okMsg := new Array<Byte>(4);
217   var ok := tcp_recv(cmdFd, okMsg);
218
219   var checked := checkServerResponse_200_226_250(okMsg);
220   free(okMsg);
221   free(buffer);
222   if (!checked) {
223     return false;
224   }
225
226   var close := gclose(opened);
227   if (sentData == size) {
228     return true;
229   } else {
230     return false;
231   }
232 }
233
234 procedure sizeHelp(cmdFd: Int, filename: Array<Byte>, cmdSize: Int)
      returns (success: Int)
235   requires cmdFd >= 0
236   requires byte_array(filename)
237   requires ((cmdSize >= 6) && (cmdSize <= 65535))
238   ensures byte_array(filename)
239   ensures ((success >= -1) && (success <= 65535))
240 {
241   var sizeMsg := new Array<Byte>(cmdSize);
```

```
242   sizeMsg := "SIZE ";
243   var copied := gstrcat(filename, sizeMsg);
244   var sent := tcp_send(cmdFd, sizeMsg, cmdSize);
245   //We get confirmation from the server when it sends us the size back.
246   free(sizeMsg);
247   var sizeBuff := new Array<Byte>(128); //this could probably be smaller
248   var recvData := tcp_recv(cmdFd, sizeBuff); //response gets sent on
          command buffer
249
250   var checked := checkServerResponse_213(sizeBuff);
251   if (!checked) {
252     free(sizeBuff);
253     return -1;
254   }
255   //this ensures we receive a valid size response
256   var size := atoiFrom(sizeBuff, 4);
257   free(sizeBuff);
258   if ((size <= 0) || (size > 65535)) { //65535-6 comes from byte_array.
          spl's max length.  We subtract 6 because we add 6 later on.
259     return -1;
260   }
261   return size;
262 }
263
264 procedure retrHelp(cmdFd: Int, filename: Array<Byte>, cmdSize: Int)
      returns (success: Bool)
265   requires cmdFd >= 0
266   requires byte_array(filename)
267   requires ((cmdSize >= 6) && (cmdSize <= 65535))
268   ensures byte_array(filename)
269 {
270   var recvMsg := new Array<Byte>(cmdSize);
271   recvMsg := "RETR ";
272   var copied := gstrcat(filename, recvMsg);
273   var sent := tcp_send(cmdFd, recvMsg, cmdSize);
274   free(recvMsg);
```

```
275
276   var okMsg := new Array<Byte>(4);
277   var ok := tcp_recv(cmdFd, okMsg);
278
279   var checked := checkServerResponse_200_150(okMsg);
280   free(okMsg);
281   if (!checked) {
282     return false;
283   }
284   return true;
285 }
286
287
288 procedure downloadFile(cmdFd: Int, dataFD: Int, filename: Array<Byte>)
289   returns (success: Bool)
290   requires cmdFd >= 0 && dataFD >= 0
291   requires byte_array(filename)
292   ensures byte_array(filename)
293 {
294   if ((filename.length <= 0) || (filename.length > (65535-6))) { //
        65535-6 comes from byte_array.spl's max length. We subtract 6
        because we add 6 later on.
295     return false;
296   }
297   var cmdSize := 6 + filename.length;
298
299   var size := sizeHelp(cmdFd, filename, cmdSize);
300   if (size < 0){
301     return false;
302   }
303
304   var buffer := new Array<Byte>(size);
305
306   var retrDone := retrHelp(cmdFd, filename, cmdSize);
307   if (!retrDone) {
308     free(buffer);
```

```
309       return false;
310     }
311
312     var recvData := tcp_recv(dataFD, buffer);
313     //get response
314     var okMsg := new Array<Byte>(4);
315     var ok := tcp_recv(cmdFd, okMsg);
316
317     var checked := checkServerResponse_200_226_250(okMsg);
318     free(okMsg);
319
320     if (!checked) {
321       free(buffer);
322       return false;
323     }
324
325     if (recvData < 0){
326       free(buffer);
327       return false;
328     }
329
330     var saveFD := gopen(filename, O_CREAT | O_WRONLY | O_TRUNC); //here we
          save the file under the same name as it is stored on the server.
          We use O_TRUNC to wipe the file before we overwrite it
331     if (saveFD < 0){
332       free(buffer);
333       return false;
334     }
335     var written := gwrite(saveFD, buffer);
336     free(buffer);
337     if (written < 0){
338       return false;
339     }
340     var close := gclose(saveFD);
341     return true;
342 }
```

```
343
344 procedure client(upload: Bool)
345   returns (res: Int)
346   requires emp;
347   ensures  emp;
348 {
349 //if store is 1, then we store a file STOR
350 // else we download it RETR
351
352   var port := new Array<Byte>(5);
353   port := "4444";
354
355   //we get the ip
356   var remoteAddr := askIP(port);
357   if (remoteAddr == null){
358     free(port);
359     return -1;
360   }
361
362   var fd := connectTo(remoteAddr);
363   free(remoteAddr);
364   if (fd == -1) {
365     free(port);
366     return -1;
367   }
368
369   //we set up the data connection
370   port := "4440";
371   var dataAddr := get_address4(null, port);
372   if (dataAddr == null){
373     free(port);
374     return -1;
375   }
376   var connectedDataFD := setupDataConnection(fd, dataAddr, port);
377   free(port);
378   free(dataAddr);
```

```
379   if (connectedDataFD == -1) {
380     var closed := gclose(fd);
381     return 3;
382   }
383   //now we can receive a bunch of data
384
385   //Note that at this point:
386   //-fd is the command file descriptor
387   //-connectedDataFD is the data file descriptor.
388
389   var authenticated := authenticate(fd);
390   if (!authenticated) {
391     var closed := gclose(fd);
392     closed := gclose(connectedDataFD);
393     return 4;
394   }
395
396   var filename := askFilename();
397
398   if (filename == null) {
399     var closed := gclose(fd);
400     closed := gclose(connectedDataFD);
401     return 5;
402   }
403
404   var success := false;
405   if (upload){
406     //we store the file
407     success := uploadFile(fd, connectedDataFD, filename);
408   } else {
409     //we retrieve the file
410     success := downloadFile(fd, connectedDataFD, filename);
411   }
412   free(filename);
413   if (!success) {
414     var closed := gclose(fd);
```

```
415       closed := gclose(connectedDataFD);
416       return 6;
417     }
418     var closeConn := gclose(connectedDataFD);
419     if (closeConn < 0){
420       return 22;
421     }
422     var quitMsg := new Array<Byte>(5);
423     quitMsg := "QUIT";
424     var sent := tcp_send(fd, quitMsg, 5);
425     free(quitMsg);
426     //we can check for the correct quitting response or not.
427     var closeFD := gclose(fd);
428     if (closeFD < 0){
429       return 21;
430     }
431     return 0;
432 }
433
434 procedure Main(args: Array<Byte>)
435     returns (res: Int)
436     requires byte_array(args)
437     ensures  byte_array(args)
438 {
439     var upload := doWeUpload();
440     if (upload == 1){
441       res := client(true);
442     } else if (upload == 0){
443       res := client(false);
444     } else {
445       res := -1;
446     }
447     return res;
448 }
```

Below is the FTP server code, which I wrote.

```
1  include "../../../lib/socket.spl";
2  include "../../../lib/file.spl";
3  include "../../../lib/console.spl";
4  include "../array/string.spl";
5  include "../array/copy_byte_slice.spl";
6  include "../../../lib/makeStr.spl";
7
8  procedure connectMeCommand(cmdFd: Int, cmdAddr: SocketAddressIP4)
       returns (res: Int)
9    requires socket_addr_4(cmdAddr)
10   requires cmdFd >= 0
11   ensures socket_addr_4(cmdAddr)
12   ensures res >= -1
13 {
14   var listening := glisten(cmdFd, 10);
15   if (!listening) {
16     return -1;
17   }
18
19   var connFd := accept4(cmdFd, cmdAddr);
20   return connFd;
21 }
22
23 procedure recvDataConnection(cmdFd: Int, addr: SocketAddressIP4) returns
       (res: Int)
24   requires cmdFd >= 0
25   requires socket_addr_4(addr)
26   ensures socket_addr_4(addr)
27 {
28   var resp := new Array<Byte>(11);
29   var response := tcp_recv(cmdFd, resp);
30
31   var fd := create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
32   if (fd == -1) {
33     free(resp);
```

```
34      return -1;
35    }
36    var portArray := copy_byte_slice(resp, 5, 10);
37    free(resp);
38
39    addr.sin4_port := atoiG(portArray);
40    free(portArray);
41    if (addr.sin4_port < 0 || addr.sin4_port >= 65536) {
42      addr.sin4_port := 0;
43      var closing := gclose(fd);
44      return -1;
45    }
46
47    if (connect4(fd, addr)) {
48      return fd;
49    } else {
50      var closing := gclose(fd);
51      return -1;
52    }
53 }
54
55 procedure handleAuth(cmdFd: Int)
56    returns (success: Bool)
57    requires cmdFd >= 0
58 {
59    var userName := new Array<Byte>(12); //we're capping the username
          message out at 12
60    var recvUser := tcp_recv(cmdFd, userName);
61    free(userName);
62    var response := new Array<Byte>(4);
63    response := "331"; //we don't actually do anything with the username
64    var sent := tcp_send(cmdFd, response, 4);
65    free(response);
66
67    var sentPassword := new Array<Byte>(15);
68    var recvpass := tcp_recv(cmdFd, sentPassword);
```

```
69
70    var pass := new Array<Byte>(15);
71    pass := "PASS anonymous";
72    var isOkay := false;
73    if (gstrcmp(pass, sentPassword) == 0) {
74      isOkay := true;
75    }
76    free(sentPassword);
77    free(pass);
78
79    response := new Array<Byte>(4);
80    if (isOkay) {
81      response := "230";
82      var sent := tcp_send(cmdFd, response, 4);
83      free(response);
84      return true;
85    } else {
86      response := "530";
87      var sent := tcp_send(cmdFd, response, 4);
88      free(response);
89      return false;
90    }
91 }
92
93 procedure accept_incoming_file(dataFd: Int, filename: Array<Byte>,
       allo_size: Int) returns (res:Int)
94    requires dataFd >= 0
95    requires allo_size >= 0
96    requires allo_size <= 65535
97    requires byte_array(filename)
98    ensures byte_array(filename)
99    ensures res >= -1
100 {
101   var buffer := new Array<Byte>(allo_size);
102   var recv := tcp_recv(dataFd, buffer);
103   if (recv < 0) {
```

```
104     free(buffer);
105     return -1;
106   }
107   var fileFd := gopen(filename, O_CREAT | O_TRUNC | O_WRONLY);
108   if (fileFd < 0) {
109     free(buffer);
110     return -1;
111   }
112   var written := gwrite(fileFd, buffer);
113   free(buffer);
114   var closed := gclose(fileFd);
115   if (written < 0 || closed < 0) {
116     return -1;
117   }
118   return 1;
119 }
120 procedure send_outgoing_file(dataFd: Int, filename: Array<Byte>) returns
      (res: Bool)
121   requires dataFd >= 0
122   requires byte_array(filename)
123   ensures byte_array(filename)
124 {
125   var fileFd := gopen(filename, O_CREAT | O_RDONLY);
126   var flag := false;
127   if (fileFd < 0){
128     flag := true;
129   }
130   var fileS := fileSize(filename);
131   if (fileS < 0 || fileS > 65535) {
132     fileS := 0; //we need it to be zero in order to allocate the array
133     flag := true;
134   }
135   var buffer := new Array<Byte>(fileS);
136
137   if (!flag) {
138     var read := gread(fileFd, buffer);
```

```
139     var closed := gclose(fileFd);
140     if (read < 0 || closed < 0){
141       flag := true;
142     }
143   }
144   if (!flag) {
145     var sent := tcp_send(dataFd, buffer, fileS);
146     if (sent < 0) {
147       flag := true;
148     }
149   }
150   free(buffer);
151   //we want to return !flag
152   if (flag) {
153     return false;
154   } else {
155     return true;
156   }
157 }
158
159 procedure store_help(cmdFd: Int, dataFd: Int, filename: Array<Byte>,
      allo_size: Int) returns (fail: Bool)
160   requires cmdFd >= 0;
161   requires dataFd >= 0;
162   requires allo_size >= 1;
163   requires allo_size <= 65535;
164   requires byte_array(filename);
165   ensures byte_array(filename);
166 {
167   var ok := new Array<Byte>(4);
168   ok := "150";
169   var sent := tcp_send(cmdFd, ok, 4);
170   free(ok);
171   var stored := accept_incoming_file(dataFd, filename, allo_size);
172   if (stored < 0) {
173     var notOk := new Array<Byte>(4);
```

```
174       notOk := "550";
175       sent := tcp_send(cmdFd, notOk, 4);
176       free(notOk);
177       return true;
178     }
179     var goodPacket := new Array<Byte>(4);
180     goodPacket := "250";
181     sent := tcp_send(cmdFd, goodPacket, 4);
182     free(goodPacket);
183     return false;
184 }
185
186
187 procedure allo_help(cmdFd:Int, sizeB: Array<Byte>) returns (allo_size:
        Int)
188     requires byte_array(sizeB);
189     ensures byte_array(sizeB);
190     requires cmdFd >= 0;
191     ensures allo_size >= -1;
192     ensures allo_size <= 65535;
193 {
194     if (sizeB.length < 2) {
195       var notOkay := new Array<Byte>(4);
196       notOkay := "552";
197       var sent := tcp_send(cmdFd, notOkay, 4);
198       free(notOkay);
199       return -1;
200     }
201     var allo_size_arr := copy_byte_slice(sizeB, 0, 1);
202     allo_size := byte2int(allo_size_arr[0]);
203     free(allo_size_arr);
204     if ((allo_size < 1) || (allo_size > 65535)) {
205       var notOkay := new Array<Byte>(4);
206       notOkay := "552";
207       var sent := tcp_send(cmdFd, notOkay, 4);
208       free(notOkay);
```

```
209     return -1;
210   }
211   var goodPacket := new Array<Byte>(4);
212   goodPacket := "200";
213   var sent := tcp_send(cmdFd, goodPacket, 4);
214   free(goodPacket);
215   return allo_size;
216 }
217
218 procedure size_help(cmdFd: Int, filename: Array<Byte>) returns (fail:
      Bool)
219   requires cmdFd >= 0;
220   requires byte_array(filename);
221   ensures byte_array(filename);
222 {
223   var sizeF := fileSize(filename); //an int
224   if (sizeF < 0) {
225     //file doesn't exist, or we can't get at it
226     var badPacket := new Array<Byte>(4);
227     badPacket := "550";
228     var sent := tcp_send(cmdFd, badPacket, 4);
229     free(badPacket);
230     return true;
231   }
232   var sizePacket := new Array<Byte>(2);
233   sizePacket[0] := makeStrFromInt(sizeF);
234   sizePacket[1] := int2byte(0);
235   var goodPacket := new Array<Byte>(6);
236   goodPacket := "213 ";
237   var finished := gstrcat(sizePacket, goodPacket);
238   var sent := tcp_send(cmdFd, goodPacket, 6);
239   free(sizePacket);
240   free(goodPacket);
241   return false;
242 }
243
```

```
244 procedure retr_help(cmdFd: Int, dataFd: Int, filename: Array<Byte>)
        returns (fail: Bool)
245   requires cmdFd >= 0;
246   requires dataFd >= 0;
247   requires byte_array(filename);
248   ensures byte_array(filename);
249 {
250   var ok := new Array<Byte>(4);
251   ok := "150";
252   var sent := tcp_send(cmdFd, ok, 4);
253   free(ok);
254   var done := send_outgoing_file(dataFd, filename);
255   if (!done){
256     //we failed
257     var notOk := new Array<Byte>(4);
258     notOk := "550";
259     sent := tcp_send(cmdFd, notOk, 4);
260     free(notOk);
261     return true;
262   }
263   var goodPacket := new Array<Byte>(4);
264   goodPacket := "250";
265   sent := tcp_send(cmdFd, goodPacket, 4);
266   free(goodPacket);
267   return false;
268 }
269
270 procedure process_string(thing:Array<Byte>) returns (out: Array<Byte>)
271   requires byte_array(thing) &*& thing.length == 4
272   ensures byte_array(out) &*& out.length == 5
273 {
274     out := new Array<Byte>(thing.length + 1);
275     out[0] := int2byte(0);
276     var copied := gstrcat(thing, out);
277     free(thing);
278     out[out.length -1] := int2byte(0);
```

```
279     return out;
280 }
281
282 procedure is_stor(cmd: Array<Byte>) returns (is: Bool)
283   requires byte_array(cmd) &*& cmd.length == 5
284   ensures  byte_array(cmd) &*& cmd.length == 5
285 {
286   var stor := new Array<Byte>(5);
287   stor := "STOR";
288   if (gstrcmp(cmd, stor) == 0) {
289     free(stor);
290     return true;
291   } else {
292     free(stor);
293     return false;
294   }
295 }
296
297 procedure is_size(cmd: Array<Byte>) returns (is: Bool)
298   requires byte_array(cmd) &*& cmd.length == 5
299   ensures  byte_array(cmd) &*& cmd.length == 5
300 {
301   var size := new Array<Byte>(5);
302   size := "SIZE";
303   if (gstrcmp(cmd, size) == 0) {
304     free(size);
305     return true;
306   } else {
307     free(size);
308     return false;
309   }
310 }
311
312 procedure is_allo(cmd: Array<Byte>) returns (is: Bool)
313   requires byte_array(cmd) &*& cmd.length == 5
314   ensures  byte_array(cmd) &*& cmd.length == 5
```

```
315 {
316   var allo := new Array<Byte>(5);
317   allo := "ALLO";
318   if (gstrcmp(cmd, allo) == 0) {
319     free(allo);
320     return true;
321   } else {
322     free(allo);
323     return false;
324   }
325 }
326
327 procedure is_retr(cmd: Array<Byte>) returns (is: Bool)
328   requires byte_array(cmd) &*& cmd.length == 5
329   ensures  byte_array(cmd) &*& cmd.length == 5
330 {
331   var retr := new Array<Byte>(5);
332   retr := "RETR";
333   if (gstrcmp(cmd, retr) == 0) {
334     free(retr);
335     return true;
336   } else {
337     free(retr);
338     return false;
339   }
340 }
341
342 procedure is_quit(cmd: Array<Byte>) returns (is: Bool)
343   requires byte_array(cmd) &*& cmd.length == 5
344   ensures byte_array(cmd) &*& cmd.length == 5
345 {
346   var quit := new Array<Byte>(5);
347   quit := "QUIT";
348   if (gstrcmp(cmd, quit) == 0) {
349     free(quit);
350     return true;
```

```
351     } else {
352       free(quit);
353       return false;
354     }
355 }
356
357 procedure server() returns (res:Int)
358     ensures res >= -1;
359 {
360     var port := new Array<Byte>(5);
361     port := "4444";
362
363     var cmdFd := -1;
364     if (port.length > 65535) {
365       return -1;
366     }
367
368     var cmdAddr := get_address4(null, port);
369     free(port);
370     if (cmdAddr == null) {
371       return -1;
372     }
373
374     var tempCmdFd := create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
375     if (tempCmdFd < 0) {
376       free(cmdAddr);
377       return -1;
378     }
379     var bound := bind4(tempCmdFd, cmdAddr);
380     if (!bound) {
381       free(cmdAddr);
382       var closed := gclose(tempCmdFd);
383       return -1;
384     }
385
386     while(cmdFd < 0)
```

```
387     invariant cmdFd >= -1
388     invariant socket_addr_4(cmdAddr)
389     invariant tempCmdFd >= 0
390   {
391     cmdFd := connectMeCommand(tempCmdFd, cmdAddr);
392   }
393   var closedTemp := gclose(tempCmdFd);
394
395   var dataFd := recvDataConnection(cmdFd, cmdAddr);
396   free(cmdAddr);
397   if (dataFd <= -1) {
398     var closed := gclose(cmdFd);
399     return -1;
400   }
401
402   //if handleAuth fails, abort.
403   var authenticated := handleAuth(cmdFd);
404   if (!authenticated){
405     var closed := gclose(cmdFd);
406     closed := gclose(dataFd);
407     return -1;
408   }
409
410
411   var iQuit := false;
412   var properQuit := false;
413   var allo_size := 65535;
414
415   while (!iQuit)
416     invariant cmdFd >= 0
417     invariant dataFd >= 0
418   {
419     var request := new Array<Byte>(150); // it needs to be so big to
          hold the filename
420     var recd := tcp_recv(cmdFd, request);
421     var typeCom := copy_byte_slice(request, 0, 4);
```

```
422     var final := process_string(typeCom);
423     var filename := copy_byte_slice(request, 5, (request.length -1));
424     free(request);
425
426     if (is_allo(final)) {
427       allo_size := allo_help(cmdFd, filename);
428       if ((allo_size < 1) || (allo_size > 65535)) {
429         allo_size := 65535;
430         free(final);
431         iQuit := true;
432       }
433     }
434     else if (is_stor(final)) {
435       if ((allo_size < 1) || (allo_size > 65535)){
436         free(filename);
437         free(final);
438         iQuit := true;
439       }
440       else {
441         var temp := store_help(cmdFd, dataFd, filename, allo_size);
442         if (temp) {
443           free(filename);
444           free(final);
445         }
446         iQuit := temp;
447       }
448     }
449     else if (is_size(final)) {
450       var temp := size_help(cmdFd, filename);
451       if (temp) {
452         free(filename);
453         free(final);
454       }
455       iQuit := temp;
456     }
457     else if (is_retr(final)) {
```

```
458         var temp := retr_help(cmdFd, dataFd, filename);
459         if (temp) {
460           free(filename);
461           free(final);
462         }
463         iQuit := temp;
464       }
465       else if (is_quit(final)) {
466         properQuit := true;
467         iQuit := true;
468       } else {
469         var badPacket := new Array<Byte>(4);
470         badPacket := "500";
471         var sent := tcp_send(cmdFd, badPacket, 4);
472         free(badPacket);
473         iQuit := true;
474         //something we did not expect
475       }
476       free(filename);
477       free(final);
478     }
479
480     var closed := gclose(cmdFd);
481     closed := gclose(dataFd);
482
483     if (properQuit){
484       return 0;
485     }
486     else {
487       return -1;
488     }
489 }
490
491 procedure Main(args: Array<Byte>)
492     returns (res:Int)
493     requires byte_array(args)
```

```
494     ensures byte_array(args)
495 {
496     res := server();
497     return res;
498 }
```

A.2 Function Verification Wrappers

Below are the function verification stubs for the console.c code, which Zufferey wrote.

```
1  include "../tests/spl/array/byte_array.spl";
2
3  procedure gputs(buffer: Array<Byte>) returns (success: Bool)
4      requires byte_array(buffer)
5      ensures byte_array(buffer)
6      ensures buffer == old(buffer)
7      ensures buffer.length == old(buffer.length)
8      ensures forall i: Int :: i >= 0 && i <= buffer.length ==> buffer[i]
           == old(buffer[i])
9
10 procedure ggets(buffer: Array<Byte>) returns (nChars: Int)
11     requires byte_array(buffer)
12     ensures byte_array(buffer)
13     ensures buffer == old(buffer)
14     ensures buffer.length == old(buffer.length)
15     ensures nChars >= -1 && nChars <= buffer.length
```

Below are the function verification stubs for the file.c and file.h code, which Zufferey and I wrote.

```
1  include "../tests/spl/array/byte_array.spl";
2
3  //using this requires adding "#include <fcntl.h>" to the generated C
       file
4
5  O_CREAT: Int
6  O_APPEND: Int
7  O_TRUNC: Int
8
9  O_RDONLY: Int
10 O_WRONLY: Int
11 O_RDWR: Int
12
13 procedure gopen(pathname: Array<Byte>, flags: Int) returns (fd: Int)
14     requires byte_array(pathname)
15     //requires string(pathname)
16     requires (flags & ~(O_CREAT | O_APPEND | O_TRUNC | O_RDONLY |
           O_WRONLY | O_RDWR)) == 0
17     ensures byte_array(pathname)
18     //ensures string(pathname)
19     ensures pathname == old(pathname)
20     ensures pathname.length == old(pathname.length)
21     ensures forall i: Int :: i >= 0 && i <= pathname.length ==> pathname
           [i] == old(pathname[i])
22     ensures fd >= -1
23
24 procedure gread(fd: Int, buffer: Array<Byte>) returns (nbr: Int)
25     requires fd >= 0
26     requires byte_array(buffer)
27     ensures byte_array(buffer)
28     ensures buffer == old(buffer)
29     ensures buffer.length == old(buffer.length)
30     ensures nbr >= -1 && nbr <= buffer.length
31
```

```
32 procedure greadOffset(fd: Int, buffer: Array<Byte>, offset: Int) returns
       (nbr: Int)
33     requires fd >= 0
34     requires byte_array(buffer)
35     requires offset >= 0
36     requires offset <= buffer.length
37     ensures byte_array(buffer)
38     ensures buffer == old(buffer)
39     ensures buffer.length == old(buffer.length)
40     ensures nbr >= -1 && nbr <= buffer.length
41
42 procedure gwrite(fd: Int, buffer: Array<Byte>) returns (nbr: Int)
43     requires fd >= 0
44     requires byte_array(buffer)
45     ensures byte_array(buffer)
46     ensures buffer == old(buffer)
47     ensures buffer.length == old(buffer.length)
48     ensures forall i: Int :: i >= 0 && i <= buffer.length ==> buffer[i]
          == old(buffer[i])
49     ensures nbr >= -1 && nbr <= buffer.length
50
51 procedure gwrite2(fd: Int, buffer: Array<Byte>, size: Int) returns (nbr:
       Int)
52     requires fd >= 0
53     requires byte_array(buffer)
54     requires size >= 0 && size <= buffer.length
55     ensures byte_array(buffer)
56     ensures buffer == old(buffer)
57     ensures buffer.length == old(buffer.length)
58     ensures forall i: Int :: i >= 0 && i <= buffer.length ==> buffer[i]
          == old(buffer[i])
59     ensures nbr >= -1 && nbr <= size
60
61 procedure gclose(fd: Int) returns (nbr: Int)
62     requires fd >= 0
63     ensures nbr >= -1
```

```
64
65 procedure fileSize(pathname: Array<Byte>) returns (size: Int)
66     requires byte_array(pathname)
67     ensures byte_array(pathname)
68     ensures pathname == old(pathname)
69     ensures pathname.length == old(pathname.length)
70     ensures forall i: Int :: i >= 0 && i <= pathname.length ==> pathname
           [i] == old(pathname[i])
```

Below is the function verification stub for the makeStr.c code, which I wrote. The makeStr code is used to convert integers to strings.

```
1 procedure makeStrFromInt(myInt: Int) returns (myByte: Byte)
```

Below are the function verification stubs for the socket.c code, which Zufferey wrote.

```
1  include "../tests/spl/array/byte_array.spl";
2
3  ///////////////
4  // Constants //
5  ///////////////
6
7  //using this requires adding "#include <netinet/in.h>" to the generated
       C file
8
9  PF_INET: Int
10 PF_INET6: Int
11
12 SOCK_DGRAM: Int
13 SOCK_STREAM: Int
14
15 IPPROTO_TCP: Int
16 IPPROTO_UDP: Int
17
18
19 ////////////
20 // Struct //
21 ////////////
22
23 struct SocketAddressIP4 {
24   //var sin4_family: Int; = AF_INET
25   var sin4_port: Int;
26   var sin4_addr: Int;
27   //var sin4_addr_lower: Int;
28   //var sin4_addr_upper: Int;
29 }
30
31 struct SocketAddressIP6 {
32   //var sin6_family: Int; = AF_INET6
33   var sin6_port: Int;
```

```
34    var sin6_flowinfo: Int;
35    var sin6_addr: Array<Byte>; //should be 16 Bytes
36    var sin6_scope_id: Int;
37 }
38
39 ////////////////
40 // Predicates //
41 ////////////////
42
43 predicate socket_addr_4( address: SocketAddressIP4)(FP: Set<
       SocketAddressIP4>) {
44   FP == Set<SocketAddressIP4>(address) &&
45    address.sin4_port >= 0 &&
46    address.sin4_port < 65536
47 }
48
49 predicate socket_addr_6( address: SocketAddressIP6)(FP0: Set<
       SocketAddressIP6>, FP1: Set<Array<Byte>>, FP2: Set<ArrayCell<Byte>>)
        {
50    FP0 == Set<SocketAddressIP6>(address) &&
51    FP1 == Set<Array<Byte>>(address.sin6_addr) &&
52    byte_arrayseg(address.sin6_addr, 0, address.sin6_addr.length, FP2) &&
53    address.sin6_addr.length == 16 &&
54    address.sin6_port >= 0 &&
55    address.sin6_port < 65536
56 }
57
58 ////////////////
59 // Procedures //
60 ////////////////
61
62 procedure get_address4(node: Array<Byte>, service: Array<Byte>) returns
       (address: SocketAddressIP4)
63    requires (node == null || byte_array(node)) &*& byte_array(service)
64    ensures  (node == null || byte_array(node)) &*& byte_array(service)
65    ensures  node == old(node)
```

```
66    ensures  address == null || socket_addr_4(address)
67
68  procedure get_address6(node: Array<Byte>, service: Array<Byte>) returns
        (address: SocketAddressIP6)
69    requires (node == null || byte_array(node)) &*& byte_array(service)
70    ensures  (node == null || byte_array(node)) &*& byte_array(service)
71    ensures  node == old(node)
72    ensures  address == null || socket_addr_6(address)
73
74  procedure create_socket(inet_type: Int, socket_type: Int, protocol: Int)
        returns (fd: Int)
75    requires inet_type == PF_INET || inet_type == PF_INET6
76    requires (socket_type == SOCK_STREAM && protocol == IPPROTO_TCP) || (
        socket_type == SOCK_DGRAM && protocol == IPPROTO_UDP)
77    ensures fd >= -1
78
79  procedure bind4(fd: Int, address: SocketAddressIP4) returns (success:
        Bool)
80    requires fd >= 0
81    requires socket_addr_4(address)
82    ensures  socket_addr_4(address)
83    ensures  address == old(address) && address.sin4_port == old(address.
        sin4_port)
84    ensures  address.sin4_addr == old(address.sin4_addr)
85
86  procedure bind6(fd: Int, address: SocketAddressIP6) returns (success:
        Bool)
87    requires fd >= 0
88    requires socket_addr_6(address)
89    ensures  socket_addr_6(address)
90    ensures  address == old(address)
91    ensures  address.sin6_port == old(address.sin6_port)
92    ensures  address.sin6_flowinfo == old(address.sin6_flowinfo)
93    ensures  address.sin6_scope_id == old(address.sin6_scope_id)
94    ensures  forall i: Int:: i >= 0 && i < 16 ==> address.sin6_addr[i] ==
        old(address.sin6_addr[i])
```

```
95
96
97 ////////////////
98 // UDP methods //
99 ////////////////
100
101 procedure udp_send4(fd: Int, msg: Array<Byte>, len: Int, address:
      SocketAddressIP4) returns (byteCount: Int)
102   requires fd >= 0
103   requires socket_addr_4(address)
104   requires byte_array(msg)
105   requires msg.length <= len
106   ensures  socket_addr_4(address)
107   ensures  address == old(address) && address.sin4_port == old(address.
        sin4_port)
108   ensures  address.sin4_addr == old(address.sin4_addr)
109   ensures  byte_array(msg)
110   ensures  msg == old(msg)
111   ensures  msg.length == old(msg.length)
112   ensures  byteCount <= msg.length
113   ensures  byteCount >= -1
114
115 procedure udp_send6(fd: Int, msg: Array<Byte>, len: Int, address:
      SocketAddressIP6) returns (byteCount: Int)
116   requires fd >= 0
117   requires socket_addr_6(address)
118   requires byte_array(msg)
119   requires msg.length <= len
120   ensures  socket_addr_6(address)
121   ensures  address == old(address)
122   ensures  address.sin6_port == old(address.sin6_port)
123   ensures  address.sin6_flowinfo == old(address.sin6_flowinfo)
124   ensures  address.sin6_scope_id == old(address.sin6_scope_id)
125   ensures  forall i: Int:: i >= 0 && i < 16 ==> address.sin6_addr[i] ==
        old(address.sin6_addr[i])
126   ensures  byte_array(msg)
```

```
127   ensures  msg == old(msg)
128   ensures  msg.length == old(msg.length)
129   ensures  byteCount <= msg.length
130   ensures  byteCount >= -1
131
132
133 procedure udp_recv4(fd: Int, msg: Array<Byte>, from: SocketAddressIP4)
        returns (byteCount: Int)
134   requires fd >= 0
135   requires socket_addr_4(from)
136   requires byte_array(msg)
137   ensures  socket_addr_4(from)
138   ensures  byte_array(msg)
139   ensures  msg == old(msg)
140   ensures  msg.length == old(msg.length)
141   ensures  byteCount <= msg.length
142   ensures  byteCount >= -1
143
144 procedure udp_recv6(fd: Int, msg: Array<Byte>, from: SocketAddressIP6)
        returns (byteCount: Int)
145   requires fd >= 0
146   requires socket_addr_6(from)
147   requires byte_array(msg)
148   ensures  socket_addr_6(from)
149   ensures  byte_array(msg)
150   ensures  msg == old(msg)
151   ensures  msg.length == old(msg.length)
152   ensures  byteCount <= msg.length
153   ensures  byteCount >= -1
154
155
156 ////////////////
157 // TCP methods //
158 ////////////////
159
160 procedure glisten(fd: Int, backlog: Int) returns (success: Bool)
```

```
161   requires fd >= 0
162   requires backlog >= 0
163
164
165 procedure accept4(fd: Int, address: SocketAddressIP4) returns (
        acceptedFd: Int)
166   requires fd >= 0
167   requires socket_addr_4(address)
168   ensures  acceptedFd >= -1
169   ensures  socket_addr_4(address)
170
171 procedure accept6(fd: Int, address: SocketAddressIP6) returns (
        acceptedFd: Int)
172   requires fd >= 0
173   requires socket_addr_6(address)
174   ensures  acceptedFd >= -1
175   ensures  socket_addr_6(address)
176
177
178 procedure connect4(fd: Int, address: SocketAddressIP4) returns (success:
        Bool)
179   requires fd >= 0
180   requires socket_addr_4(address)
181   ensures  socket_addr_4(address)
182   ensures  address == old(address) && address.sin4_port == old(address.
        sin4_port)
183   ensures  address.sin4_addr == old(address.sin4_addr)
184
185 procedure connect6(fd: Int, address: SocketAddressIP6) returns (success:
        Bool)
186   requires fd >= 0
187   requires socket_addr_6(address)
188   ensures  socket_addr_6(address)
189   ensures  address == old(address)
190   ensures  address.sin6_port == old(address.sin6_port)
191   ensures  address.sin6_flowinfo == old(address.sin6_flowinfo)
```

```
192   ensures  address.sin6_scope_id == old(address.sin6_scope_id)
193   ensures  forall i: Int:: i >= 0 && i < 16 ==> address.sin6_addr[i] ==
          old(address.sin6_addr[i])
194
195
196 procedure tcp_send(fd: Int, msg: Array<Byte>, len: Int) returns (
      byteCount: Int)
197   requires fd >= 0
198   requires byte_array(msg)
199   requires msg.length <= len
200   ensures  byte_array(msg)
201   ensures  msg == old(msg)
202   ensures  msg.length == old(msg.length)
203   ensures  byteCount <= msg.length
204   ensures  byteCount >= -1
205
206
207 procedure tcp_recv(fd: Int, msg: Array<Byte>) returns (byteCount: Int)
208   requires fd >= 0
209   requires byte_array(msg)
210   ensures  byte_array(msg)
211   ensures  msg == old(msg)
212   ensures  msg.length == old(msg.length)
213   ensures  byteCount <= msg.length
214   ensures  byteCount >= -1
```

Appendix B

C code

B.1 GRASShopper Compiled C code

Below is the C implementation of my client.spl code. GRASShopper generated the C code.

```
1  /*
2   * Includes
3   */
4  #include <stdbool.h>
5  #include <stdlib.h>
6  #include <assert.h>
7  #include <netinet/in.h>
8  #include <fcntl.h>
9
10 /*
11  * Preloaded Code
12  */
13
14 typedef struct {
15   int length;
16   int arr[];
17 } SPLArray_int;
18
19 SPLArray_int* newSPLArray_int(int size) {
```

```
20    SPLArray_int* a = (SPLArray_int*)malloc(sizeof(SPLArray_int) + size *
          sizeof(int));
21    assert(a != NULL);
22    a->length = size;
23    return a;
24  }
25
26  typedef struct {
27    int length;
28    bool arr[];
29  } SPLArray_bool;
30
31  SPLArray_bool* newSPLArray_bool(int size) {
32    SPLArray_bool* a = (SPLArray_bool*)malloc(sizeof(SPLArray_bool) + size
          * sizeof(bool));
33    assert(a != NULL);
34    a->length = size;
35    return a;
36  }
37
38  typedef struct {
39    int length;
40    char arr[];
41  } SPLArray_char;
42
43  SPLArray_char* newSPLArray_char(int size) {
44    SPLArray_char* a = (SPLArray_char*)malloc(sizeof(SPLArray_char) + size
          * sizeof(char));
45    assert(a != NULL);
46    a->length = size;
47    return a;
48  }
49
50  typedef struct {
51    int length;
52    void* arr[];
```

```
53 } SPLArray_generic;
54
55 SPLArray_generic* newSPLArray_generic(int size) {
56   SPLArray_generic* a = (SPLArray_generic*)malloc(sizeof(
         SPLArray_generic) + size * sizeof(void*));
57   assert(a != NULL);
58   a->length = size;
59   return a;
60 }
61
62 /*
63  * Structs
64  */
65 struct SocketAddressIP4;
66 struct SocketAddressIP6;
67
68 typedef struct SocketAddressIP4 {
69   int sin4_addr;
70   int sin4_port;
71 } SocketAddressIP4;
72
73 typedef struct SocketAddressIP6 {
74   SPLArray_char* sin6_addr;
75   int sin6_flowinfo;
76   int sin6_port;
77   int sin6_scope_id;
78 } SocketAddressIP6;
79
80 /*
81  * Procedures
82  */
83 int Main (SPLArray_char* args);
84 int accept4 (int fd, struct SocketAddressIP4* address);
85 int accept6 (int fd_1, struct SocketAddressIP6* address_1);
86 bool allo_help (int size, int cmdFd);
87 SPLArray_char* askFilename ();
```

```
88 struct SocketAddressIP4* askIP (SPLArray_char* port);
89 int atoiFrom (SPLArray_char* str, int startIdx);
90 int atoiG (SPLArray_char* str_1);
91 bool authenticate (int cmdFd_1);
92 bool bind4 (int fd_2, struct SocketAddressIP4* address_2);
93 bool bind6 (int fd_3, struct SocketAddressIP6* address_3);
94 bool checkServerResponsePASS (SPLArray_char* response);
95 bool checkServerResponseUSER (SPLArray_char* response_1);
96 bool checkServerResponse_200 (SPLArray_char* response_2);
97 bool checkServerResponse_200_150 (SPLArray_char* response_3);
98 bool checkServerResponse_200_226_250 (SPLArray_char* response_4);
99 bool checkServerResponse_213 (SPLArray_char* response_5);
100 int client (bool upload_1);
101 SPLArray_char* concat (SPLArray_char* str1, SPLArray_char* str2);
102 bool connect4 (int fd_5, struct SocketAddressIP4* address_4);
103 bool connect6 (int fd_6, struct SocketAddressIP6* address_5);
104 int connectTo (struct SocketAddressIP4* addr);
105 SPLArray_char* copy_byte_slice (SPLArray_char* a, int start, int end);
106 int create_socket (int inet_type, int socket_type, int protocol);
107 int doWeUpload ();
108 bool downloadFile (int cmdFd_2, int dataFD, SPLArray_char* filename_2);
109 bool equals (SPLArray_char* first, SPLArray_char* second);
110 int fileSize (SPLArray_char* pathname);
111 int gclose (int fd_9);
112 struct SocketAddressIP4* get_address4 (SPLArray_char* node,
      SPLArray_char* service);
113 struct SocketAddressIP6* get_address6 (SPLArray_char* node_1,
      SPLArray_char* service_1);
114 int ggets (SPLArray_char* buffer_1);
115 bool glisten (int fd_10, int backlog);
116 int gopen (SPLArray_char* pathname_1, int flags);
117 bool gputs (SPLArray_char* buffer_2);
118 int gread (int fd_12, SPLArray_char* buffer_3);
119 int greadOffset (int fd_13, SPLArray_char* buffer_4, int offset);
120 int gstrcat (SPLArray_char* str1_1, SPLArray_char* str2_1);
121 int gstrcmp (SPLArray_char* s1, SPLArray_char* s2);
```

```
122 SPLArray_char* gstrdup (SPLArray_char* str_2);
123 int gstrlen (SPLArray_char* str_3);
124 int gwrite (int fd_14, SPLArray_char* buffer_5);
125 int gwrite2 (int fd_15, SPLArray_char* buffer_6, int size_3);
126 bool retrHelp (int cmdFd_3, SPLArray_char* filename_3, int cmdSize_1);
127 int setupDataConnection (int cmdFd_4, struct SocketAddressIP4* address_8
    , SPLArray_char* port_2);
128 int sizeHelp (int cmdFd_5, SPLArray_char* filename_4, int cmdSize_2);
129 bool store_send_help (int cmdFd_6, SPLArray_char* filename_5);
130 SPLArray_char* strconcat (SPLArray_char* str1_2, SPLArray_char* str2_2);
131 int tcp_recv (int fd_16, SPLArray_char* msg);
132 int tcp_send (int fd_17, SPLArray_char* msg_1, int len);
133 int udp_recv4 (int fd_18, SPLArray_char* msg_2, struct SocketAddressIP4*
     from);
134 int udp_recv6 (int fd_19, SPLArray_char* msg_3, struct SocketAddressIP6*
     from_1);
135 int udp_send4 (int fd_20, SPLArray_char* msg_4, int len_1, struct
    SocketAddressIP4* address_9);
136 int udp_send6 (int fd_21, SPLArray_char* msg_5, int len_2, struct
    SocketAddressIP6* address_10);
137 bool uploadFile (int cmdFd_7, int dataFD_2, SPLArray_char* filename_6);
138
139 int Main (SPLArray_char* args) {
140   int res;
141   int upload;
142
143   upload = doWeUpload();
144   if ((upload == 1)) {
145     res = client(true);
146   } else {
147     if ((upload == 0)) {
148       res = client(false);
149     } else {
150       res = (-1);
151     }
152   }
```

```
153   return res;
154 }
155
156 bool allo_help (int size, int cmdFd) {
157   bool success;
158   SPLArray_char* tmp_2;
159   SPLArray_char* tmp_1;
160   SPLArray_char* tmp;
161   int sent;
162   SPLArray_char* sendMsg;
163   SPLArray_char* okMsg;
164   int ok;
165   int copied;
166   bool checked;
167   SPLArray_char* alloSize;
168
169   tmp = newSPLArray_char( 2);
170   alloSize = tmp;
171   (alloSize->arr[0]) = ((char) size);
172   (alloSize->arr[1]) = ((char) 0);
173   tmp_1 = newSPLArray_char( 6);
174   sendMsg = tmp_1;
175   (sendMsg->arr[0]) = ((char) 65);
176   (sendMsg->arr[1]) = ((char) 76);
177   (sendMsg->arr[2]) = ((char) 76);
178   (sendMsg->arr[3]) = ((char) 79);
179   (sendMsg->arr[4]) = ((char) 32);
180   (sendMsg->arr[5]) = ((char) 0);
181   copied = gstrcat(alloSize, sendMsg);
182   sent = tcp_send(cmdFd, sendMsg, 7);
183   free(sendMsg);
184
185   free(alloSize);
186
187   tmp_2 = newSPLArray_char( 4);
188   okMsg = tmp_2;
```

```
189   ok = tcp_recv(cmdFd, okMsg);
190   checked = checkServerResponse_200(okMsg);
191   free(okMsg);
192
193   if ((!checked)) {
194     return false;
195   }
196   return true;
197 }
198
199 SPLArray_char* askFilename () {
200   SPLArray_char* fn;
201   SPLArray_char* tmp_4;
202   SPLArray_char* tmp_3;
203   SPLArray_char* text;
204   bool putted;
205   int numChars;
206   SPLArray_char* filename;
207
208   tmp_3 = newSPLArray_char( 100);
209   filename = tmp_3;
210   tmp_4 = newSPLArray_char( 24);
211   text = tmp_4;
212   (text->arr[0]) = ((char) 105);
213   (text->arr[1]) = ((char) 110);
214   (text->arr[2]) = ((char) 112);
215   (text->arr[3]) = ((char) 117);
216   (text->arr[4]) = ((char) 116);
217   (text->arr[5]) = ((char) 32);
218   (text->arr[6]) = ((char) 116);
219   (text->arr[7]) = ((char) 104);
220   (text->arr[8]) = ((char) 101);
221   (text->arr[9]) = ((char) 32);
222   (text->arr[10]) = ((char) 102);
223   (text->arr[11]) = ((char) 105);
224   (text->arr[12]) = ((char) 108);
```

```
225   (text->arr[13]) = ((char) 101);
226   (text->arr[14]) = ((char) 32);
227   (text->arr[15]) = ((char) 110);
228   (text->arr[16]) = ((char) 97);
229   (text->arr[17]) = ((char) 109);
230   (text->arr[18]) = ((char) 101);
231   (text->arr[19]) = ((char) 58);
232   (text->arr[20]) = ((char) 0);
233   putted = gputs(text);
234   free(text);
235
236   if ((!putted)) {
237     free(filename);
238
239     return NULL;
240   }
241   numChars = ggets(filename);
242   if (((numChars >= 100) || (numChars <= 1))) {
243     free(filename);
244
245     return NULL;
246   }
247   (filename->arr[(numChars - 1)]) = ((char) 0);
248   return filename;
249 }
250
251 struct SocketAddressIP4* askIP (SPLArray_char* port) {
252   struct SocketAddressIP4* ip;
253   SPLArray_char* writtenPrompt;
254   bool written;
255   SPLArray_char* tmp_6;
256   SPLArray_char* tmp_5;
257   int numChars_1;
258   SPLArray_char* getip;
259   struct SocketAddressIP4* dataAddr;
260
```

```
261   tmp_5 = newSPLArray_char( 20);
262   writtenPrompt = tmp_5;
263   (writtenPrompt->arr[0]) = ((char) 69);
264   (writtenPrompt->arr[1]) = ((char) 110);
265   (writtenPrompt->arr[2]) = ((char) 116);
266   (writtenPrompt->arr[3]) = ((char) 101);
267   (writtenPrompt->arr[4]) = ((char) 114);
268   (writtenPrompt->arr[5]) = ((char) 32);
269   (writtenPrompt->arr[6]) = ((char) 73);
270   (writtenPrompt->arr[7]) = ((char) 80);
271   (writtenPrompt->arr[8]) = ((char) 32);
272   (writtenPrompt->arr[9]) = ((char) 111);
273   (writtenPrompt->arr[10]) = ((char) 102);
274   (writtenPrompt->arr[11]) = ((char) 32);
275   (writtenPrompt->arr[12]) = ((char) 115);
276   (writtenPrompt->arr[13]) = ((char) 101);
277   (writtenPrompt->arr[14]) = ((char) 114);
278   (writtenPrompt->arr[15]) = ((char) 118);
279   (writtenPrompt->arr[16]) = ((char) 101);
280   (writtenPrompt->arr[17]) = ((char) 114);
281   (writtenPrompt->arr[18]) = ((char) 58);
282   (writtenPrompt->arr[19]) = ((char) 0);
283   written = gputs(writtenPrompt);
284   free(writtenPrompt);
285
286   tmp_6 = newSPLArray_char( 50);
287   getip = tmp_6;
288   numChars_1 = ggets(getip);
289   dataAddr = NULL;
290   if ((numChars_1 < 1)) {
291     free(getip);
292
293     return dataAddr;
294   }
295   (getip->arr[(numChars_1 - 1)]) = ((char) 0);
296   if ((!(numChars_1 == 1))) {
```

```
297     dataAddr = get_address4(getip, port);
298   } else {
299     dataAddr = get_address4(NULL, port);
300   }
301   free(getip);
302 
303   return dataAddr;
304 }
305 
306 int atoiFrom (SPLArray_char* str, int startIdx) {
307   int res_1;
308   bool isPositive;
309   int i;
310   bool foundStart;
311   bool foundEnd;
312   int digit;
313 
314   res_1 = 0;
315   i = startIdx;
316   if ((i > (str->length))) {
317     i = (str->length);
318   }
319   foundStart = false;
320   foundEnd = false;
321   isPositive = true;
322   while (true) {
323     if (!(((i < (str->length)) && (!foundStart)))) {
324       break;
325     }
326     if (((((((((str->arr[i]) == ((char) 9)) || ((str->arr[i]) == ((char)
          10))) || ((str->arr[i]) == ((char) 11))) || ((str->arr[i]) == ((
          char) 12))) || ((str->arr[i]) == ((char) 13))) || ((str->arr[i])
           == ((char) 32)))) {
327       i = (i + 1);
328     } else {
329       foundStart = true;
```

```
330     }
331   }
332   if ((i < (str->length))) {
333     if (((str->arr[i]) == ((char) 45))) {
334       isPositive = false;
335       i = (i + 1);
336     }
337   }
338   while (true) {
339     if (!(((i < (str->length)) && (!foundEnd)))) {
340       break;
341     }
342     if ((((str->arr[i]) >= ((char) 48)) && ((str->arr[i]) <= ((char) 57)
          ))) {
343       digit = ((int) ((str->arr[i]) - ((char) 48)));
344       res_1 = (res_1 * 10);
345       res_1 = (res_1 + digit);
346       i = (i + 1);
347     } else {
348       foundEnd = true;
349     }
350   }
351   if ((!isPositive)) {
352     res_1 = ((-1) * res_1);
353   }
354   return res_1;
355 }
356
357 int atoiG (SPLArray_char* str_1) {
358   int res_2;
359   res_2 = atoiFrom(str_1, 0);
360   return res_2;
361   return res_2;
362 }
363
364 bool authenticate (int cmdFd_1) {
```

```
365   bool success_1;
366   SPLArray_char* userMsg;
367   SPLArray_char* tmp_10;
368   SPLArray_char* tmp_9;
369   SPLArray_char* tmp_8;
370   SPLArray_char* tmp_7;
371   int sent_1;
372   SPLArray_char* passMsg;
373   SPLArray_char* okMsg_1;
374   int ok_1;
375   bool checked_1;
376
377   tmp_7 = newSPLArray_char( 12);
378   userMsg = tmp_7;
379   (userMsg->arr[0]) = ((char) 85);
380   (userMsg->arr[1]) = ((char) 83);
381   (userMsg->arr[2]) = ((char) 69);
382   (userMsg->arr[3]) = ((char) 82);
383   (userMsg->arr[4]) = ((char) 32);
384   (userMsg->arr[5]) = ((char) 112);
385   (userMsg->arr[6]) = ((char) 111);
386   (userMsg->arr[7]) = ((char) 116);
387   (userMsg->arr[8]) = ((char) 97);
388   (userMsg->arr[9]) = ((char) 116);
389   (userMsg->arr[10]) = ((char) 111);
390   (userMsg->arr[11]) = ((char) 0);
391   sent_1 = tcp_send(cmdFd_1, userMsg, 12);
392   free(userMsg);
393
394   tmp_8 = newSPLArray_char( 4);
395   okMsg_1 = tmp_8;
396   ok_1 = tcp_recv(cmdFd_1, okMsg_1);
397   checked_1 = checkServerResponseUSER(okMsg_1);
398   free(okMsg_1);
399
400   if ((!checked_1)) {
```

```
401       return false;
402     }
403     tmp_9 = newSPLArray_char( 15);
404     passMsg = tmp_9;
405     (passMsg->arr[0]) = ((char) 80);
406     (passMsg->arr[1]) = ((char) 65);
407     (passMsg->arr[2]) = ((char) 83);
408     (passMsg->arr[3]) = ((char) 83);
409     (passMsg->arr[4]) = ((char) 32);
410     (passMsg->arr[5]) = ((char) 97);
411     (passMsg->arr[6]) = ((char) 110);
412     (passMsg->arr[7]) = ((char) 111);
413     (passMsg->arr[8]) = ((char) 110);
414     (passMsg->arr[9]) = ((char) 121);
415     (passMsg->arr[10]) = ((char) 109);
416     (passMsg->arr[11]) = ((char) 111);
417     (passMsg->arr[12]) = ((char) 117);
418     (passMsg->arr[13]) = ((char) 115);
419     (passMsg->arr[14]) = ((char) 0);
420     sent_1 = tcp_send(cmdFd_1, passMsg, 15);
421     free(passMsg);
422
423     tmp_10 = newSPLArray_char( 4);
424     okMsg_1 = tmp_10;
425     ok_1 = tcp_recv(cmdFd_1, okMsg_1);
426     checked_1 = checkServerResponsePASS(okMsg_1);
427     free(okMsg_1);
428
429     if ((!checked_1)) {
430       return false;
431     }
432     return true;
433 }
434
435 bool checkServerResponsePASS (SPLArray_char* response) {
436     bool success_4;
```

```
437   int ack;
438
439   ack = atoiG(response);
440   if (((((ack == 200) || (ack == 202)) || (ack == 230)) || (ack == 234))
          ) {
441     success_4 = true;
442   } else {
443     success_4 = false;
444   }
445   return success_4;
446 }
447
448 bool checkServerResponseUSER (SPLArray_char* response_1) {
449   bool success_5;
450   int ack_1;
451
452   ack_1 = atoiG(response_1);
453   if (((((ack_1 == 200) || (ack_1 == 230)) || (ack_1 == 234)) || (ack_1
          == 331))) {
454     success_5 = true;
455   } else {
456     success_5 = false;
457   }
458   return success_5;
459 }
460
461 bool checkServerResponse_200 (SPLArray_char* response_2) {
462   bool success_6;
463   int ack_2;
464
465   ack_2 = atoiG(response_2);
466   if ((ack_2 == 200)) {
467     return true;
468   } else {
469     return false;
470   }
```

```
471   return success_6;
472 }
473
474 bool checkServerResponse_200_150 (SPLArray_char* response_3) {
475   bool success_7;
476   int ack_3;
477
478   ack_3 = atoiG(response_3);
479   if (((ack_3 == 200) || (ack_3 == 150))) {
480     success_7 = true;
481   } else {
482     success_7 = false;
483   }
484   return success_7;
485 }
486
487 bool checkServerResponse_200_226_250 (SPLArray_char* response_4) {
488   bool success_8;
489   int ack_4;
490
491   ack_4 = atoiG(response_4);
492   if ((((ack_4 == 200) || (ack_4 == 226)) || (ack_4 == 250))) {
493     success_8 = true;
494   } else {
495     success_8 = false;
496   }
497   return success_8;
498 }
499
500 bool checkServerResponse_213 (SPLArray_char* response_5) {
501   bool success_9;
502   int ack_5;
503
504   ack_5 = atoiG(response_5);
505   if ((ack_5 == 213)) {
506     return true;
```

```
507   } else {
508     return false;
509   }
510   return success_9;
511 }
512
513 int client (bool upload_1) {
514   int res_3;
515   SPLArray_char* tmp_12;
516   SPLArray_char* tmp_11;
517   bool success_10;
518   int sent_2;
519   struct SocketAddressIP4* remoteAddr;
520   SPLArray_char* quitMsg;
521   SPLArray_char* port_1;
522   SPLArray_char* filename_1;
523   int fd_4;
524   struct SocketAddressIP4* dataAddr_1;
525   int connectedDataFD;
526   int closed_3;
527   int closed_2;
528   int closed_1;
529   int closed;
530   int closeFD;
531   int closeConn;
532   bool authenticated;
533
534   tmp_11 = newSPLArray_char( 5);
535   port_1 = tmp_11;
536   (port_1->arr[0]) = ((char) 52);
537   (port_1->arr[1]) = ((char) 52);
538   (port_1->arr[2]) = ((char) 52);
539   (port_1->arr[3]) = ((char) 52);
540   (port_1->arr[4]) = ((char) 0);
541   remoteAddr = askIP(port_1);
542   if ((remoteAddr == NULL)) {
```

```
543       free(port_1);
544
545       return (-1);
546     }
547     fd_4 = connectTo(remoteAddr);
548     free(remoteAddr);
549
550     if ((fd_4 == (-1))) {
551       free(port_1);
552
553       return (-1);
554     }
555     (port_1->arr[0]) = ((char) 52);
556     (port_1->arr[1]) = ((char) 52);
557     (port_1->arr[2]) = ((char) 52);
558     (port_1->arr[3]) = ((char) 48);
559     (port_1->arr[4]) = ((char) 0);
560     dataAddr_1 = get_address4(NULL, port_1);
561     if ((dataAddr_1 == NULL)) {
562       free(port_1);
563
564       return (-1);
565     }
566     connectedDataFD = setupDataConnection(fd_4, dataAddr_1, port_1);
567     free(port_1);
568
569     free(dataAddr_1);
570
571     if ((connectedDataFD == (-1))) {
572       closed = gclose(fd_4);
573       return 3;
574     }
575     authenticated = authenticate(fd_4);
576     if ((!authenticated)) {
577       closed_1 = gclose(fd_4);
578       closed_1 = gclose(connectedDataFD);
```

```
579       return 4;
580     }
581     filename_1 = askFilename();
582     if ((filename_1 == NULL)) {
583       closed_2 = gclose(fd_4);
584       closed_2 = gclose(connectedDataFD);
585       return 5;
586     }
587     success_10 = false;
588     if (upload_1) {
589       success_10 = uploadFile(fd_4, connectedDataFD, filename_1);
590     } else {
591       success_10 = downloadFile(fd_4, connectedDataFD, filename_1);
592     }
593     free(filename_1);
594
595     if ((!success_10)) {
596       closed_3 = gclose(fd_4);
597       closed_3 = gclose(connectedDataFD);
598       return 6;
599     }
600     closeConn = gclose(connectedDataFD);
601     if ((closeConn < 0)) {
602       return 22;
603     }
604     tmp_12 = newSPLArray_char( 5);
605     quitMsg = tmp_12;
606     (quitMsg->arr[0]) = ((char) 81);
607     (quitMsg->arr[1]) = ((char) 85);
608     (quitMsg->arr[2]) = ((char) 73);
609     (quitMsg->arr[3]) = ((char) 84);
610     (quitMsg->arr[4]) = ((char) 0);
611     sent_2 = tcp_send(fd_4, quitMsg, 5);
612     free(quitMsg);
613
614     closeFD = gclose(fd_4);
```

```
615   if ((closeFD < 0)) {
616     return 21;
617   }
618   return 0;
619 }
620
621 SPLArray_char* concat (SPLArray_char* str1, SPLArray_char* str2) {
622   SPLArray_char* res_4;
623   SPLArray_char* tmp_13;
624   int i_2;
625   SPLArray_char* copy;
626
627
628   tmp_13 = newSPLArray_char( ((str1->length) + (str2->length)));
629   copy = tmp_13;
630   i_2 = 0;
631   while (true) {
632     if (!((i_2 < (str1->length)))) {
633       break;
634     }
635     (copy->arr[i_2]) = (str1->arr[i_2]);
636     i_2 = (i_2 + 1);
637   }
638   while (true) {
639     if (!((i_2 < ((str1->length) + (str2->length))))) {
640       break;
641     }
642     (copy->arr[i_2]) = (str2->arr[(i_2 - (str1->length))]);
643     i_2 = (i_2 + 1);
644   }
645   return copy;
646 }
647
648 int connectTo (struct SocketAddressIP4* addr) {
649   int fd_7;
650   bool tmp_14;
```

```
651
652   fd_7 = create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
653   if ((fd_7 == (-1))) {
654     return (-1);
655   }
656   tmp_14 = connect4(fd_7, addr);
657   if (tmp_14) {
658     return fd_7;
659   } else {
660     return (-1);
661   }
662   return fd_7;
663 }
664
665 SPLArray_char* copy_byte_slice (SPLArray_char* a, int start, int end) {
666   SPLArray_char* b;
667   SPLArray_char* tmp_15;
668   int i_5;
669   int finalLength;
670
671   finalLength = (end - start);
672   tmp_15 = newSPLArray_char( finalLength);
673   b = tmp_15;
674   i_5 = 0;
675   while (true) {
676     if (!((i_5 < finalLength))) {
677       break;
678     }
679     (b->arr[i_5]) = (a->arr[(i_5 + start)]);
680     i_5 = (i_5 + 1);
681   }
682   return b;
683 }
684
685 int doWeUpload () {
686   int res_5;
```

```
687   bool written_1;
688   SPLArray_char* updown;
689   SPLArray_char* uChar;
690   int tmp_20;
691   SPLArray_char* tmp_19;
692   SPLArray_char* tmp_18;
693   SPLArray_char* tmp_17;
694   SPLArray_char* tmp_16;
695   SPLArray_char* text_1;
696   int numChars_2;
697   SPLArray_char* copy_1;
698
699   tmp_16 = newSPLArray_char( 30);
700   updown = tmp_16;
701   tmp_17 = newSPLArray_char( 28);
702   text_1 = tmp_17;
703   (text_1->arr[0]) = ((char) 117);
704   (text_1->arr[1]) = ((char) 112);
705   (text_1->arr[2]) = ((char) 108);
706   (text_1->arr[3]) = ((char) 111);
707   (text_1->arr[4]) = ((char) 97);
708   (text_1->arr[5]) = ((char) 100);
709   (text_1->arr[6]) = ((char) 32);
710   (text_1->arr[7]) = ((char) 40);
711   (text_1->arr[8]) = ((char) 117);
712   (text_1->arr[9]) = ((char) 41);
713   (text_1->arr[10]) = ((char) 32);
714   (text_1->arr[11]) = ((char) 111);
715   (text_1->arr[12]) = ((char) 114);
716   (text_1->arr[13]) = ((char) 32);
717   (text_1->arr[14]) = ((char) 100);
718   (text_1->arr[15]) = ((char) 111);
719   (text_1->arr[16]) = ((char) 119);
720   (text_1->arr[17]) = ((char) 110);
721   (text_1->arr[18]) = ((char) 108);
722   (text_1->arr[19]) = ((char) 111);
```

```
723   (text_1->arr[20]) = ((char) 97);
724   (text_1->arr[21]) = ((char) 100);
725   (text_1->arr[22]) = ((char) 32);
726   (text_1->arr[23]) = ((char) 40);
727   (text_1->arr[24]) = ((char) 100);
728   (text_1->arr[25]) = ((char) 41);
729   (text_1->arr[26]) = ((char) 58);
730   (text_1->arr[27]) = ((char) 0);
731   written_1 = gputs(text_1);
732   free(text_1);
733
734   if ((!written_1)) {
735     free(updown);
736
737     return (-1);
738   } else {
739     numChars_2 = ggets(updown);
740     if ((numChars_2 == 2)) {
741       tmp_18 = newSPLArray_char( 2);
742       copy_1 = tmp_18;
743       (copy_1->arr[0]) = (updown->arr[0]);
744       (copy_1->arr[1]) = ((char) 0);
745       free(updown);
746
747       tmp_19 = newSPLArray_char( 2);
748       uChar = tmp_19;
749       (uChar->arr[0]) = ((char) 117);
750       (uChar->arr[1]) = ((char) 0);
751       tmp_20 = gstrcmp(uChar, copy_1);
752       if ((tmp_20 == 0)) {
753         free(uChar);
754
755         free(copy_1);
756
757         return 1;
758       } else {
```

```
759           free(uChar);
760
761           free(copy_1);
762
763           return 0;
764         }
765       } else {
766         free(updown);
767
768         return (-1);
769       }
770     }
771     return res_5;
772 }
773
774 bool downloadFile (int cmdFd_2, int dataFD, SPLArray_char* filename_2) {
775     bool success_13;
776     int written_2;
777     SPLArray_char* tmp_22;
778     SPLArray_char* tmp_21;
779     int size_1;
780     int saveFD;
781     bool retrDone;
782     int recvData;
783     SPLArray_char* okMsg_2;
784     int ok_2;
785     int cmdSize;
786     int close;
787     bool checked_2;
788     SPLArray_char* buffer;
789
790     if ((((filename_2->length) <= 0) || ((filename_2->length) > (65535 -
          6)))) {
791       return false;
792     }
793     cmdSize = (6 + (filename_2->length));
```

```
794   size_1 = sizeHelp(cmdFd_2, filename_2, cmdSize);
795   if ((size_1 < 0)) {
796     return false;
797   }
798   tmp_21 = newSPLArray_char( size_1);
799   buffer = tmp_21;
800   retrDone = retrHelp(cmdFd_2, filename_2, cmdSize);
801   if ((!retrDone)) {
802     free(buffer);
803 
804     return false;
805   }
806   recvData = tcp_recv(dataFD, buffer);
807   tmp_22 = newSPLArray_char( 4);
808   okMsg_2 = tmp_22;
809   ok_2 = tcp_recv(cmdFd_2, okMsg_2);
810   checked_2 = checkServerResponse_200_226_250(okMsg_2);
811   free(okMsg_2);
812 
813   if ((!checked_2)) {
814     free(buffer);
815 
816     return false;
817   }
818   if ((recvData < 0)) {
819     free(buffer);
820 
821     return false;
822   }
823   saveFD = gopen(filename_2, ((O_CREAT | O_WRONLY) | O_TRUNC));
824   if ((saveFD < 0)) {
825     free(buffer);
826 
827     return false;
828   }
829   written_2 = gwrite(saveFD, buffer);
```

```
830   free(buffer);
831
832   if ((written_2 < 0)) {
833     return false;
834   }
835   close = gclose(saveFD);
836   return true;
837 }
838
839 bool equals (SPLArray_char* first, SPLArray_char* second) {
840   bool res_6;
841   int i_7;
842
843   if ((!((first->length) == (second->length)))) {
844     return false;
845   }
846   i_7 = 0;
847   while (true) {
848     if (!(((i_7 < (first->length)) && ((first->arr[i_7]) == (second->arr
          [i_7]))))) {
849       break;
850     }
851     i_7 = (i_7 + 1);
852   }
853   if ((i_7 >= (first->length))) {
854     return true;
855   } else {
856     return false;
857   }
858   return res_6;
859 }
860
861 int gstrcat (SPLArray_char* str1_1, SPLArray_char* str2_1) {
862   int res_7;
863   int l2;
864   int l1;
```

```
865   int i_11;
866   int copy_size;
867
868   l1 = gstrlen(str1_1);
869   l2 = gstrlen(str2_1);
870   copy_size = ((str2_1->length) - l2);
871   if ((copy_size > l1)) {
872     copy_size = l1;
873   }
874   i_11 = 0;
875   while (true) {
876     if (!((i_11 < copy_size))) {
877       break;
878     }
879     (str2_1->arr[(i_11 + l2)]) = (str1_1->arr[i_11]);
880     i_11 = (i_11 + 1);
881   }
882   if (((l2 + copy_size) < (str2_1->length))) {
883     (str2_1->arr[(l2 + copy_size)]) = ((char) 0);
884   }
885   return copy_size;
886 }
887
888 int gstrcmp (SPLArray_char* s1, SPLArray_char* s2) {
889   int res_8;
890   int i_13;
891
892   i_13 = 0;
893   while (true) {
894     if (!((((i_13 < (s1->length)) && (i_13 < (s2->length))) && ((s1->arr
          [i_13]) == (s2->arr[i_13]))))) {
895       break;
896     }
897     i_13 = (i_13 + 1);
898   }
899   if (((i_13 >= (s1->length)) && (i_13 >= (s2->length)))) {
```

```
900       return 0;
901     } else {
902       if ((i_13 >= (s1->length))) {
903         return (-1);
904       } else {
905         if ((i_13 >= (s2->length))) {
906           return 1;
907         } else {
908           if (((s1->arr[i_13]) < (s2->arr[i_13]))) {
909             return (-1);
910           } else {
911             return 1;
912           }
913         }
914       }
915     }
916     return res_8;
917 }
918
919 SPLArray_char* gstrdup (SPLArray_char* str_2) {
920     SPLArray_char* res_9;
921     SPLArray_char* tmp_23;
922     int i_15;
923     SPLArray_char* copy_2;
924
925     tmp_23 = newSPLArray_char( (str_2->length));
926     copy_2 = tmp_23;
927     i_15 = 0;
928     while (true) {
929       if (!((i_15 < (str_2->length)))) {
930         break;
931       }
932       (copy_2->arr[i_15]) = (str_2->arr[i_15]);
933       i_15 = (i_15 + 1);
934     }
935     return copy_2;
```

```
936 }
937
938 int gstrlen (SPLArray_char* str_3) {
939   int res_10;
940   int i_16;
941
942   i_16 = 0;
943   while (true) {
944     if (!(((i_16 < (str_3->length)) && (!((str_3->arr[i_16]) == ((char)
          0)))))) {
945       break;
946     }
947     i_16 = (i_16 + 1);
948   }
949   return i_16;
950 }
951
952 bool retrHelp (int cmdFd_3, SPLArray_char* filename_3, int cmdSize_1) {
953   bool success_16;
954   SPLArray_char* tmp_25;
955   SPLArray_char* tmp_24;
956   int sent_3;
957   SPLArray_char* recvMsg;
958   SPLArray_char* okMsg_3;
959   int ok_3;
960   int copied_1;
961   bool checked_3;
962
963   tmp_24 = newSPLArray_char( cmdSize_1);
964   recvMsg = tmp_24;
965   (recvMsg->arr[0]) = ((char) 82);
966   (recvMsg->arr[1]) = ((char) 69);
967   (recvMsg->arr[2]) = ((char) 84);
968   (recvMsg->arr[3]) = ((char) 82);
969   (recvMsg->arr[4]) = ((char) 32);
970   (recvMsg->arr[5]) = ((char) 0);
```

```
971   copied_1 = gstrcat(filename_3, recvMsg);
972   sent_3 = tcp_send(cmdFd_3, recvMsg, cmdSize_1);
973   free(recvMsg);
974
975   tmp_25 = newSPLArray_char( 4);
976   okMsg_3 = tmp_25;
977   ok_3 = tcp_recv(cmdFd_3, okMsg_3);
978   checked_3 = checkServerResponse_200_150(okMsg_3);
979   free(okMsg_3);
980
981   if ((!checked_3)) {
982     return false;
983   }
984   return true;
985 }
986
987 int setupDataConnection (int cmdFd_4, struct SocketAddressIP4* address_8
        , SPLArray_char* port_2) {
988   int connectedDataFD_1;
989   SPLArray_char* tmp_26;
990   int sent_4;
991   SPLArray_char* portMsg;
992   bool datalistening;
993   int dataFD_1;
994   int copied_2;
995   int closeData;
996   bool bound;
997
998   dataFD_1 = create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
999   if ((dataFD_1 == (-1))) {
1000    return (-1);
1001  }
1002  bound = bind4(dataFD_1, address_8);
1003  if ((!bound)) {
1004    return (-1);
1005  }
```

```
1006    datalistening = glisten(dataFD_1, 10);
1007    if ((!datalistening)) {
1008      return (-1);
1009    }
1010    if ((((port_2->length) < 0) || ((port_2->length) > (65535 - 6)))) {
1011      return (-1);
1012    }
1013    tmp_26 = newSPLArray_char( (6 + (port_2->length)));
1014    portMsg = tmp_26;
1015    (portMsg->arr[0]) = ((char) 80);
1016    (portMsg->arr[1]) = ((char) 79);
1017    (portMsg->arr[2]) = ((char) 82);
1018    (portMsg->arr[3]) = ((char) 84);
1019    (portMsg->arr[4]) = ((char) 32);
1020    (portMsg->arr[5]) = ((char) 0);
1021    copied_2 = gstrcat(port_2, portMsg);
1022    sent_4 = tcp_send(cmdFd_4, portMsg, (portMsg->length));
1023    if ((!(sent_4 == (portMsg->length)))) {
1024      free(portMsg);
1025
1026      return (-1);
1027    }
1028    free(portMsg);
1029
1030    connectedDataFD_1 = accept4(dataFD_1, address_8);
1031    closeData = gclose(dataFD_1);
1032    return connectedDataFD_1;
1033  }
1034
1035  int sizeHelp (int cmdFd_5, SPLArray_char* filename_4, int cmdSize_2) {
1036    int success_17;
1037    SPLArray_char* tmp_28;
1038    SPLArray_char* tmp_27;
1039    SPLArray_char* sizeMsg;
1040    SPLArray_char* sizeBuff;
1041    int size_4;
```

```
1042    int sent_5;
1043    int recvData_1;
1044    int copied_3;
1045    bool checked_4;
1046
1047    tmp_27 = newSPLArray_char( cmdSize_2);
1048    sizeMsg = tmp_27;
1049    (sizeMsg->arr[0]) = ((char) 83);
1050    (sizeMsg->arr[1]) = ((char) 73);
1051    (sizeMsg->arr[2]) = ((char) 90);
1052    (sizeMsg->arr[3]) = ((char) 69);
1053    (sizeMsg->arr[4]) = ((char) 32);
1054    (sizeMsg->arr[5]) = ((char) 0);
1055    copied_3 = gstrcat(filename_4, sizeMsg);
1056    sent_5 = tcp_send(cmdFd_5, sizeMsg, cmdSize_2);
1057    free(sizeMsg);
1058
1059    tmp_28 = newSPLArray_char( 128);
1060    sizeBuff = tmp_28;
1061    recvData_1 = tcp_recv(cmdFd_5, sizeBuff);
1062    checked_4 = checkServerResponse_213(sizeBuff);
1063    if ((!checked_4)) {
1064      free(sizeBuff);
1065
1066      return (-1);
1067    }
1068    size_4 = atoiFrom(sizeBuff, 4);
1069    free(sizeBuff);
1070
1071    if (((size_4 <= 0) || (size_4 > 65535))) {
1072      return (-1);
1073    }
1074    return size_4;
1075 }
1076
1077 bool store_send_help (int cmdFd_6, SPLArray_char* filename_5) {
```

```
1078    bool success_18;
1079    SPLArray_char* tmp_30;
1080    SPLArray_char* tmp_29;
1081    int sent_6;
1082    SPLArray_char* sendMsg_1;
1083    SPLArray_char* okMsg_4;
1084    int ok_4;
1085    int copied_4;
1086    int commandSize;
1087    bool checked_5;
1088
1089    commandSize = ((filename_5->length) + 6);
1090    tmp_29 = newSPLArray_char( commandSize);
1091    sendMsg_1 = tmp_29;
1092    (sendMsg_1->arr[0]) = ((char) 83);
1093    (sendMsg_1->arr[1]) = ((char) 84);
1094    (sendMsg_1->arr[2]) = ((char) 79);
1095    (sendMsg_1->arr[3]) = ((char) 82);
1096    (sendMsg_1->arr[4]) = ((char) 32);
1097    (sendMsg_1->arr[5]) = ((char) 0);
1098    copied_4 = gstrcat(filename_5, sendMsg_1);
1099    sent_6 = tcp_send(cmdFd_6, sendMsg_1, commandSize);
1100    free(sendMsg_1);
1101
1102    tmp_30 = newSPLArray_char( 4);
1103    okMsg_4 = tmp_30;
1104    ok_4 = tcp_recv(cmdFd_6, okMsg_4);
1105    checked_5 = checkServerResponse_200_150(okMsg_4);
1106    free(okMsg_4);
1107
1108    if ((!checked_5)) {
1109      return false;
1110    }
1111    return true;
1112 }
1113
```

```
1114 SPLArray_char* strconcat (SPLArray_char* str1_2, SPLArray_char* str2_2)
       {
1115   SPLArray_char* res_11;
1116   SPLArray_char* tmp_31;
1117   int l2_1;
1118   int l1_1;
1119   int i_19;
1120   SPLArray_char* copy_3;
1121
1122   l1_1 = gstrlen(str1_2);
1123   l2_1 = gstrlen(str2_2);
1124
1125   tmp_31 = newSPLArray_char( (l1_1 + l2_1));
1126   copy_3 = tmp_31;
1127   i_19 = 0;
1128   while (true) {
1129     if (!((i_19 < l1_1))) {
1130       break;
1131     }
1132     (copy_3->arr[i_19]) = (str1_2->arr[i_19]);
1133     i_19 = (i_19 + 1);
1134   }
1135   while (true) {
1136     if (!((i_19 < (l1_1 + l2_1)))) {
1137       break;
1138     }
1139     (copy_3->arr[i_19]) = (str2_2->arr[(i_19 - l1_1)]);
1140     i_19 = (i_19 + 1);
1141   }
1142   return copy_3;
1143 }
1144
1145 bool uploadFile (int cmdFd_7, int dataFD_2, SPLArray_char* filename_6) {
1146   bool success_19;
1147   SPLArray_char* tmp_33;
1148   SPLArray_char* tmp_32;
```

```
1149    bool stored;
1150    int size_5;
1151    int sentData;
1152    int read;
1153    int opened;
1154    SPLArray_char* okMsg_5;
1155    int ok_5;
1156    int close_1;
1157    bool checked_6;
1158    SPLArray_char* buffer_7;
1159    bool allo_check;
1160
1161    size_5 = fileSize(filename_6);
1162    if (((size_5 < 0) || (size_5 > 65535))) {
1163      return false;
1164    }
1165    allo_check = allo_help(size_5, cmdFd_7);
1166    if ((!allo_check)) {
1167      return false;
1168    }
1169    opened = gopen(filename_6, (O_CREAT | O_RDONLY));
1170    if ((opened < 0)) {
1171      return false;
1172    }
1173    tmp_32 = newSPLArray_char( size_5);
1174    buffer_7 = tmp_32;
1175    read = gread(opened, buffer_7);
1176    if ((read < 0)) {
1177      free(buffer_7);
1178
1179      return false;
1180    }
1181    stored = store_send_help(cmdFd_7, filename_6);
1182    if ((!stored)) {
1183      free(buffer_7);
1184
```

```
1185      return false;
1186    }
1187    sentData = tcp_send(dataFD_2, buffer_7, size_5);
1188    tmp_33 = newSPLArray_char( 4);
1189    okMsg_5 = tmp_33;
1190    ok_5 = tcp_recv(cmdFd_7, okMsg_5);
1191    checked_6 = checkServerResponse_200_226_250(okMsg_5);
1192    free(okMsg_5);
1193
1194    free(buffer_7);
1195
1196    if ((!checked_6)) {
1197      return false;
1198    }
1199    close_1 = gclose(opened);
1200    if ((sentData == size_5)) {
1201      return true;
1202    } else {
1203      return false;
1204    }
1205    return success_19;
1206 }
1207
1208 /*
1209  * Main Function, here for compilability
1210  */
1211 int main(int argc, char *argv[]) {
1212    assert(argc <= 2);
1213    int s = 0;
1214    if (argc > 1) {
1215      for(s = 0; argv[1][s] != 0; s++) { }
1216      s++;
1217    }
1218    SPLArray_char* a = newSPLArray_char(s);
1219    for(int i = 0; i < s; i++) {
1220      a->arr[i] = argv[1][i];
```

```
1221     }
1222     return Main(a);
1223 }
```

Below is the C implementation of my server.spl code. GRASShopper generated the C code.

```
1  /*
2   * Includes
3   */
4  #include <stdbool.h>
5  #include <stdlib.h>
6  #include <assert.h>
7  #include <netinet/in.h>
8  #include <fcntl.h>
9
10 /*
11  * Preloaded Code
12  */
13
14 typedef struct {
15   int length;
16   int arr[];
17 } SPLArray_int;
18
19 SPLArray_int* newSPLArray_int(int size) {
20   SPLArray_int* a = (SPLArray_int*)malloc(sizeof(SPLArray_int) + size *
         sizeof(int));
21   assert(a != NULL);
22   a->length = size;
23   return a;
24 }
25
26 typedef struct {
27   int length;
28   bool arr[];
29 } SPLArray_bool;
30
31 SPLArray_bool* newSPLArray_bool(int size) {
32   SPLArray_bool* a = (SPLArray_bool*)malloc(sizeof(SPLArray_bool) + size
         * sizeof(bool));
```

```
33    assert(a != NULL);
34    a->length = size;
35    return a;
36  }
37
38  typedef struct {
39    int length;
40    char arr[];
41  } SPLArray_char;
42
43  SPLArray_char* newSPLArray_char(int size) {
44    SPLArray_char* a = (SPLArray_char*)malloc(sizeof(SPLArray_char) + size
          * sizeof(char));
45    assert(a != NULL);
46    a->length = size;
47    return a;
48  }
49
50  typedef struct {
51    int length;
52    void* arr[];
53  } SPLArray_generic;
54
55  SPLArray_generic* newSPLArray_generic(int size) {
56    SPLArray_generic* a = (SPLArray_generic*)malloc(sizeof(
          SPLArray_generic) + size * sizeof(void*));
57    assert(a != NULL);
58    a->length = size;
59    return a;
60  }
61
62  /*
63   * Structs
64   */
65  struct SocketAddressIP4;
66  struct SocketAddressIP6;
```

```
67
68 typedef struct SocketAddressIP4 {
69   int sin4_addr;
70   int sin4_port;
71 } SocketAddressIP4;
72
73 typedef struct SocketAddressIP6 {
74   SPLArray_char* sin6_addr;
75   int sin6_flowinfo;
76   int sin6_port;
77   int sin6_scope_id;
78 } SocketAddressIP6;
79
80 /*
81  * Procedures
82  */
83 int Main (SPLArray_char* args);
84 int accept4 (int fd, struct SocketAddressIP4* address);
85 int accept6 (int fd_1, struct SocketAddressIP6* address_1);
86 int accept_incoming_file (int dataFd, SPLArray_char* filename, int
       allo_size);
87 int allo_help (int cmdFd, SPLArray_char* sizeB);
88 int atoiFrom (SPLArray_char* str, int startIdx);
89 int atoiG (SPLArray_char* str_1);
90 bool bind4 (int fd_2, struct SocketAddressIP4* address_2);
91 bool bind6 (int fd_3, struct SocketAddressIP6* address_3);
92 SPLArray_char* concat (SPLArray_char* str1, SPLArray_char* str2);
93 bool connect4 (int fd_4, struct SocketAddressIP4* address_4);
94 bool connect6 (int fd_5, struct SocketAddressIP6* address_5);
95 int connectMeCommand (int cmdFd_1, struct SocketAddressIP4* cmdAddr);
96 SPLArray_char* copy_byte_slice (SPLArray_char* a, int start, int end);
97 int create_socket (int inet_type, int socket_type, int protocol);
98 bool equals (SPLArray_char* first, SPLArray_char* second);
99 int fileSize (SPLArray_char* pathname);
100 int gclose (int fd_7);
101 struct SocketAddressIP4* get_address4 (SPLArray_char* node,
```

```
    SPLArray_char* service);
102 struct SocketAddressIP6* get_address6 (SPLArray_char* node_1,
    SPLArray_char* service_1);
103 int ggets (SPLArray_char* buffer_1);
104 bool glisten (int fd_8, int backlog);
105 int gopen (SPLArray_char* pathname_1, int flags);
106 bool gputs (SPLArray_char* buffer_2);
107 int gread (int fd_10, SPLArray_char* buffer_3);
108 int greadOffset (int fd_11, SPLArray_char* buffer_4, int offset);
109 int gstrcat (SPLArray_char* str1_1, SPLArray_char* str2_1);
110 int gstrcmp (SPLArray_char* s1, SPLArray_char* s2);
111 SPLArray_char* gstrdup (SPLArray_char* str_2);
112 int gstrlen (SPLArray_char* str_3);
113 int gwrite (int fd_12, SPLArray_char* buffer_5);
114 int gwrite2 (int fd_13, SPLArray_char* buffer_6, int size_1);
115 bool handleAuth (int cmdFd_2);
116 bool is_allo (SPLArray_char* cmd);
117 bool is_quit (SPLArray_char* cmd_1);
118 bool is_retr (SPLArray_char* cmd_2);
119 bool is_size (SPLArray_char* cmd_3);
120 bool is_stor (SPLArray_char* cmd_4);
121 char makeStrFromInt (int myInt);
122 SPLArray_char* process_string (SPLArray_char* thing);
123 int recvDataConnection (int cmdFd_3, struct SocketAddressIP4* addr);
124 bool retr_help (int cmdFd_4, int dataFd_1, SPLArray_char* filename_1);
125 bool send_outgoing_file (int dataFd_2, SPLArray_char* filename_2);
126 int server ();
127 bool size_help (int cmdFd_6, SPLArray_char* filename_4);
128 bool store_help (int cmdFd_7, int dataFd_4, SPLArray_char* filename_5,
    int allo_size_3);
129 SPLArray_char* strconcat (SPLArray_char* str1_2, SPLArray_char* str2_2);
130 int tcp_recv (int fd_15, SPLArray_char* msg);
131 int tcp_send (int fd_16, SPLArray_char* msg_1, int len);
132 int udp_recv4 (int fd_17, SPLArray_char* msg_2, struct SocketAddressIP4*
     from);
133 int udp_recv6 (int fd_18, SPLArray_char* msg_3, struct SocketAddressIP6*
```

```
    from_1);
int udp_send4 (int fd_19, SPLArray_char* msg_4, int len_1, struct
    SocketAddressIP4* address_8);
int udp_send6 (int fd_20, SPLArray_char* msg_5, int len_2, struct
    SocketAddressIP6* address_9);

int Main (SPLArray_char* args) {
  int res;
  res = server();
  return res;
}

int accept_incoming_file (int dataFd, SPLArray_char* filename, int
    allo_size) {
  int res_1;
  int written;
  SPLArray_char* tmp;
  int recv;
  int fileFd;
  int closed;
  SPLArray_char* buffer;

  tmp = newSPLArray_char( allo_size);
  buffer = tmp;
  recv = tcp_recv(dataFd, buffer);
  if ((recv < 0)) {
    free(buffer);

    return (-1);
  }
  fileFd = gopen(filename, ((O_CREAT | O_TRUNC) | O_WRONLY));
  if ((fileFd < 0)) {
    free(buffer);

    return (-1);
  }
```

```
166   written = gwrite(fileFd, buffer);
167   free(buffer);
168
169   closed = gclose(fileFd);
170   if (((written < 0) || (closed < 0))) {
171     return (-1);
172   }
173   return 1;
174 }
175
176 int allo_help (int cmdFd, SPLArray_char* sizeB) {
177   int allo_size_1;
178   SPLArray_char* tmp_3;
179   SPLArray_char* tmp_2;
180   SPLArray_char* tmp_1;
181   int sent_2;
182   int sent_1;
183   int sent;
184   SPLArray_char* notOkay_1;
185   SPLArray_char* notOkay;
186   SPLArray_char* goodPacket;
187   SPLArray_char* allo_size_arr;
188
189   if (((sizeB->length) < 2)) {
190     tmp_1 = newSPLArray_char( 4);
191     notOkay = tmp_1;
192     (notOkay->arr[0]) = ((char) 53);
193     (notOkay->arr[1]) = ((char) 53);
194     (notOkay->arr[2]) = ((char) 50);
195     (notOkay->arr[3]) = ((char) 0);
196     sent = tcp_send(cmdFd, notOkay, 4);
197     free(notOkay);
198
199     return (-1);
200   }
201   allo_size_arr = copy_byte_slice(sizeB, 0, 1);
```

```
202   allo_size_1 = ((int) (allo_size_arr->arr[0]));
203   free(allo_size_arr);
204
205   if (((allo_size_1 < 1) || (allo_size_1 > 65535))) {
206     tmp_2 = newSPLArray_char( 4);
207     notOkay_1 = tmp_2;
208     (notOkay_1->arr[0]) = ((char) 53);
209     (notOkay_1->arr[1]) = ((char) 53);
210     (notOkay_1->arr[2]) = ((char) 50);
211     (notOkay_1->arr[3]) = ((char) 0);
212     sent_1 = tcp_send(cmdFd, notOkay_1, 4);
213     free(notOkay_1);
214
215     return (-1);
216   }
217   tmp_3 = newSPLArray_char( 4);
218   goodPacket = tmp_3;
219   (goodPacket->arr[0]) = ((char) 50);
220   (goodPacket->arr[1]) = ((char) 48);
221   (goodPacket->arr[2]) = ((char) 48);
222   (goodPacket->arr[3]) = ((char) 0);
223   sent_2 = tcp_send(cmdFd, goodPacket, 4);
224   free(goodPacket);
225
226   return allo_size_1;
227 }
228
229 int atoiFrom (SPLArray_char* str, int startIdx) {
230   int res_2;
231   bool isPositive;
232   int i;
233   bool foundStart;
234   bool foundEnd;
235   int digit;
236
237   res_2 = 0;
```

```
238   i = startIdx;
239   if ((i > (str->length))) {
240     i = (str->length);
241   }
242   foundStart = false;
243   foundEnd = false;
244   isPositive = true;
245   while (true) {
246     if (!(((i < (str->length)) && (!foundStart)))) {
247       break;
248     }
249     if (((((((str->arr[i]) == ((char) 9)) || ((str->arr[i]) == ((char)
          10))) || ((str->arr[i]) == ((char) 11))) || ((str->arr[i]) == ((
          char) 12))) || ((str->arr[i]) == ((char) 13))) || ((str->arr[i])
           == ((char) 32)))) {
250       i = (i + 1);
251     } else {
252       foundStart = true;
253     }
254   }
255   if ((i < (str->length))) {
256     if (((str->arr[i]) == ((char) 45))) {
257       isPositive = false;
258       i = (i + 1);
259     }
260   }
261   while (true) {
262     if (!(((i < (str->length)) && (!foundEnd)))) {
263       break;
264     }
265     if ((((str->arr[i]) >= ((char) 48)) && ((str->arr[i]) <= ((char) 57)
          ))) {
266       digit = ((int) ((str->arr[i]) - ((char) 48)));
267       res_2 = (res_2 * 10);
268       res_2 = (res_2 + digit);
269       i = (i + 1);
```

```
270     } else {
271       foundEnd = true;
272     }
273   }
274   if ((!isPositive)) {
275     res_2 = ((-1) * res_2);
276   }
277   return res_2;
278 }
279 
280 int atoiG (SPLArray_char* str_1) {
281   int res_3;
282   res_3 = atoiFrom(str_1, 0);
283   return res_3;
284   return res_3;
285 }
286 
287 SPLArray_char* concat (SPLArray_char* str1, SPLArray_char* str2) {
288   SPLArray_char* res_4;
289   SPLArray_char* tmp_4;
290   int i_2;
291   SPLArray_char* copy;
292 
293 
294   tmp_4 = newSPLArray_char( ((str1->length) + (str2->length)));
295   copy = tmp_4;
296   i_2 = 0;
297   while (true) {
298     if (!((i_2 < (str1->length)))) {
299       break;
300     }
301     (copy->arr[i_2]) = (str1->arr[i_2]);
302     i_2 = (i_2 + 1);
303   }
304   while (true) {
305     if (!((i_2 < ((str1->length) + (str2->length))))) {
```

```
306         break;
307       }
308       (copy->arr[i_2]) = (str2->arr[(i_2 - (str1->length))]);
309       i_2 = (i_2 + 1);
310     }
311     return copy;
312 }
313 
314 int connectMeCommand (int cmdFd_1, struct SocketAddressIP4* cmdAddr) {
315     int res_5;
316     bool listening;
317     int connFd;
318 
319     listening = glisten(cmdFd_1, 10);
320     if ((!listening)) {
321       return (-1);
322     }
323     connFd = accept4(cmdFd_1, cmdAddr);
324     return connFd;
325 }
326 
327 SPLArray_char* copy_byte_slice (SPLArray_char* a, int start, int end) {
328     SPLArray_char* b;
329     SPLArray_char* tmp_5;
330     int i_5;
331     int finalLength;
332 
333     finalLength = (end - start);
334     tmp_5 = newSPLArray_char( finalLength);
335     b = tmp_5;
336     i_5 = 0;
337     while (true) {
338       if (!((i_5 < finalLength))) {
339         break;
340       }
341       (b->arr[i_5]) = (a->arr[(i_5 + start)]);
```

```
342     i_5 = (i_5 + 1);
343   }
344   return b;
345 }
346
347 bool equals (SPLArray_char* first, SPLArray_char* second) {
348   bool res_6;
349   int i_7;
350
351   if ((!((first->length) == (second->length)))) {
352     return false;
353   }
354   i_7 = 0;
355   while (true) {
356     if (!(((i_7 < (first->length)) && ((first->arr[i_7]) == (second->arr
          [i_7]))))) {
357       break;
358     }
359     i_7 = (i_7 + 1);
360   }
361   if ((i_7 >= (first->length))) {
362     return true;
363   } else {
364     return false;
365   }
366   return res_6;
367 }
368
369 int gstrcat (SPLArray_char* str1_1, SPLArray_char* str2_1) {
370   int res_7;
371   int l2;
372   int l1;
373   int i_11;
374   int copy_size;
375
376   l1 = gstrlen(str1_1);
```

```
377   l2 = gstrlen(str2_1);
378   copy_size = ((str2_1->length) - l2);
379   if ((copy_size > l1)) {
380     copy_size = l1;
381   }
382   i_11 = 0;
383   while (true) {
384     if (!((i_11 < copy_size))) {
385       break;
386     }
387     (str2_1->arr[(i_11 + l2)]) = (str1_1->arr[i_11]);
388     i_11 = (i_11 + 1);
389   }
390   if (((l2 + copy_size) < (str2_1->length))) {
391     (str2_1->arr[(l2 + copy_size)]) = ((char) 0);
392   }
393   return copy_size;
394 }
395
396 int gstrcmp (SPLArray_char* s1, SPLArray_char* s2) {
397   int res_8;
398   int i_13;
399
400   i_13 = 0;
401   while (true) {
402     if (!((((i_13 < (s1->length)) && (i_13 < (s2->length))) && ((s1->arr
          [i_13]) == (s2->arr[i_13]))))) {
403       break;
404     }
405     i_13 = (i_13 + 1);
406   }
407   if (((i_13 >= (s1->length)) && (i_13 >= (s2->length)))) {
408     return 0;
409   } else {
410     if ((i_13 >= (s1->length))) {
411       return (-1);
```

```
412     } else {
413       if ((i_13 >= (s2->length))) {
414         return 1;
415       } else {
416         if (((s1->arr[i_13]) < (s2->arr[i_13]))) {
417           return (-1);
418         } else {
419           return 1;
420         }
421       }
422     }
423   }
424   return res_8;
425 }
426
427 SPLArray_char* gstrdup (SPLArray_char* str_2) {
428   SPLArray_char* res_9;
429   SPLArray_char* tmp_6;
430   int i_15;
431   SPLArray_char* copy_1;
432
433   tmp_6 = newSPLArray_char( (str_2->length));
434   copy_1 = tmp_6;
435   i_15 = 0;
436   while (true) {
437     if (!((i_15 < (str_2->length)))) {
438       break;
439     }
440     (copy_1->arr[i_15]) = (str_2->arr[i_15]);
441     i_15 = (i_15 + 1);
442   }
443   return copy_1;
444 }
445
446 int gstrlen (SPLArray_char* str_3) {
447   int res_10;
```

```
448   int i_16;
449
450   i_16 = 0;
451   while (true) {
452     if (!(((i_16 < (str_3->length)) && (!((str_3->arr[i_16]) == ((char)
          0)))))) {
453       break;
454     }
455     i_16 = (i_16 + 1);
456   }
457   return i_16;
458 }
459
460 bool handleAuth (int cmdFd_2) {
461   bool success_6;
462   SPLArray_char* userName;
463   SPLArray_char* tmp_12;
464   int tmp_11;
465   SPLArray_char* tmp_10;
466   SPLArray_char* tmp_9;
467   SPLArray_char* tmp_8;
468   SPLArray_char* tmp_7;
469   SPLArray_char* sentPassword;
470   int sent_5;
471   int sent_4;
472   int sent_3;
473   SPLArray_char* response;
474   int recvpass;
475   int recvUser;
476   SPLArray_char* pass;
477   bool isOkay;
478
479   tmp_7 = newSPLArray_char( 12);
480   userName = tmp_7;
481   recvUser = tcp_recv(cmdFd_2, userName);
482   free(userName);
```

```
483
484   tmp_8 = newSPLArray_char( 4);
485   response = tmp_8;
486   (response->arr[0]) = ((char) 51);
487   (response->arr[1]) = ((char) 51);
488   (response->arr[2]) = ((char) 49);
489   (response->arr[3]) = ((char) 0);
490   sent_3 = tcp_send(cmdFd_2, response, 4);
491   free(response);
492
493   tmp_9 = newSPLArray_char( 15);
494   sentPassword = tmp_9;
495   recvpass = tcp_recv(cmdFd_2, sentPassword);
496   tmp_10 = newSPLArray_char( 15);
497   pass = tmp_10;
498   (pass->arr[0]) = ((char) 80);
499   (pass->arr[1]) = ((char) 65);
500   (pass->arr[2]) = ((char) 83);
501   (pass->arr[3]) = ((char) 83);
502   (pass->arr[4]) = ((char) 32);
503   (pass->arr[5]) = ((char) 97);
504   (pass->arr[6]) = ((char) 110);
505   (pass->arr[7]) = ((char) 111);
506   (pass->arr[8]) = ((char) 110);
507   (pass->arr[9]) = ((char) 121);
508   (pass->arr[10]) = ((char) 109);
509   (pass->arr[11]) = ((char) 111);
510   (pass->arr[12]) = ((char) 117);
511   (pass->arr[13]) = ((char) 115);
512   (pass->arr[14]) = ((char) 0);
513   isOkay = false;
514   tmp_11 = gstrcmp(pass, sentPassword);
515   if ((tmp_11 == 0)) {
516     isOkay = true;
517   }
518   free(sentPassword);
```

```
519
520   free(pass);
521
522   tmp_12 = newSPLArray_char( 4);
523   response = tmp_12;
524   if (isOkay) {
525     (response->arr[0]) = ((char) 50);
526     (response->arr[1]) = ((char) 51);
527     (response->arr[2]) = ((char) 48);
528     (response->arr[3]) = ((char) 0);
529     sent_4 = tcp_send(cmdFd_2, response, 4);
530     free(response);
531
532     return true;
533   } else {
534     (response->arr[0]) = ((char) 53);
535     (response->arr[1]) = ((char) 51);
536     (response->arr[2]) = ((char) 48);
537     (response->arr[3]) = ((char) 0);
538     sent_5 = tcp_send(cmdFd_2, response, 4);
539     free(response);
540
541     return false;
542   }
543   return success_6;
544 }
545
546 bool is_allo (SPLArray_char* cmd) {
547   bool is;
548   int tmp_14;
549   SPLArray_char* tmp_13;
550   SPLArray_char* allo;
551
552   tmp_13 = newSPLArray_char( 5);
553   allo = tmp_13;
554   (allo->arr[0]) = ((char) 65);
```

```
555   (allo->arr[1]) = ((char) 76);
556   (allo->arr[2]) = ((char) 76);
557   (allo->arr[3]) = ((char) 79);
558   (allo->arr[4]) = ((char) 0);
559   tmp_14 = gstrcmp(cmd, allo);
560   if ((tmp_14 == 0)) {
561     free(allo);
562
563     return true;
564   } else {
565     free(allo);
566
567     return false;
568   }
569   return is;
570 }
571
572 bool is_quit (SPLArray_char* cmd_1) {
573   bool is_1;
574   int tmp_16;
575   SPLArray_char* tmp_15;
576   SPLArray_char* quit;
577
578   tmp_15 = newSPLArray_char( 5);
579   quit = tmp_15;
580   (quit->arr[0]) = ((char) 81);
581   (quit->arr[1]) = ((char) 85);
582   (quit->arr[2]) = ((char) 73);
583   (quit->arr[3]) = ((char) 84);
584   (quit->arr[4]) = ((char) 0);
585   tmp_16 = gstrcmp(cmd_1, quit);
586   if ((tmp_16 == 0)) {
587     free(quit);
588
589     return true;
590   } else {
```

```
591     free(quit);
592
593     return false;
594   }
595   return is_1;
596 }
597
598 bool is_retr (SPLArray_char* cmd_2) {
599   bool is_2;
600   int tmp_18;
601   SPLArray_char* tmp_17;
602   SPLArray_char* retr;
603
604   tmp_17 = newSPLArray_char( 5);
605   retr = tmp_17;
606   (retr->arr[0]) = ((char) 82);
607   (retr->arr[1]) = ((char) 69);
608   (retr->arr[2]) = ((char) 84);
609   (retr->arr[3]) = ((char) 82);
610   (retr->arr[4]) = ((char) 0);
611   tmp_18 = gstrcmp(cmd_2, retr);
612   if ((tmp_18 == 0)) {
613     free(retr);
614
615     return true;
616   } else {
617     free(retr);
618
619     return false;
620   }
621   return is_2;
622 }
623
624 bool is_size (SPLArray_char* cmd_3) {
625   bool is_3;
626   int tmp_20;
```

```
627   SPLArray_char* tmp_19;
628   SPLArray_char* size_2;
629
630   tmp_19 = newSPLArray_char( 5);
631   size_2 = tmp_19;
632   (size_2->arr[0]) = ((char) 83);
633   (size_2->arr[1]) = ((char) 73);
634   (size_2->arr[2]) = ((char) 90);
635   (size_2->arr[3]) = ((char) 69);
636   (size_2->arr[4]) = ((char) 0);
637   tmp_20 = gstrcmp(cmd_3, size_2);
638   if ((tmp_20 == 0)) {
639     free(size_2);
640
641     return true;
642   } else {
643     free(size_2);
644
645     return false;
646   }
647   return is_3;
648 }
649
650 bool is_stor (SPLArray_char* cmd_4) {
651   bool is_4;
652   int tmp_22;
653   SPLArray_char* tmp_21;
654   SPLArray_char* stor;
655
656   tmp_21 = newSPLArray_char( 5);
657   stor = tmp_21;
658   (stor->arr[0]) = ((char) 83);
659   (stor->arr[1]) = ((char) 84);
660   (stor->arr[2]) = ((char) 79);
661   (stor->arr[3]) = ((char) 82);
662   (stor->arr[4]) = ((char) 0);
```

```
663   tmp_22 = gstrcmp(cmd_4, stor);
664   if ((tmp_22 == 0)) {
665     free(stor);
666
667     return true;
668   } else {
669     free(stor);
670
671     return false;
672   }
673   return is_4;
674 }
675
676 SPLArray_char* process_string (SPLArray_char* thing) {
677   SPLArray_char* out;
678   SPLArray_char* tmp_23;
679   int copied;
680
681   tmp_23 = newSPLArray_char( ((thing->length) + 1));
682   out = tmp_23;
683   (out->arr[0]) = ((char) 0);
684   copied = gstrcat(thing, out);
685   free(thing);
686
687   (out->arr[((out->length) - 1)]) = ((char) 0);
688   return out;
689 }
690
691 int recvDataConnection (int cmdFd_3, struct SocketAddressIP4* addr) {
692   int res_11;
693   bool tmp_26;
694   int tmp_25;
695   SPLArray_char* tmp_24;
696   int response_1;
697   SPLArray_char* resp;
698   SPLArray_char* portArray;
```

```
699   int fd_14;
700   int closing_1;
701   int closing;
702
703   tmp_24 = newSPLArray_char( 11);
704   resp = tmp_24;
705   response_1 = tcp_recv(cmdFd_3, resp);
706   fd_14 = create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
707   if ((fd_14 == (-1))) {
708     free(resp);
709
710     return (-1);
711   }
712   portArray = copy_byte_slice(resp, 5, 10);
713   free(resp);
714
715   tmp_25 = atoiG(portArray);
716   (addr->sin4_port) = tmp_25;
717   free(portArray);
718
719   if ((((addr->sin4_port) < 0) || ((addr->sin4_port) >= 65536))) {
720     (addr->sin4_port) = 0;
721     closing = gclose(fd_14);
722     return (-1);
723   }
724   tmp_26 = connect4(fd_14, addr);
725   if (tmp_26) {
726     return fd_14;
727   } else {
728     closing_1 = gclose(fd_14);
729     return (-1);
730   }
731   return res_11;
732 }
733
734 bool retr_help (int cmdFd_4, int dataFd_1, SPLArray_char* filename_1) {
```

```
735   bool fail;
736   SPLArray_char* tmp_29;
737   SPLArray_char* tmp_28;
738   SPLArray_char* tmp_27;
739   int sent_6;
740   SPLArray_char* ok;
741   SPLArray_char* notOk;
742   SPLArray_char* goodPacket_1;
743   bool done;
744
745   tmp_27 = newSPLArray_char( 4);
746   ok = tmp_27;
747   (ok->arr[0]) = ((char) 49);
748   (ok->arr[1]) = ((char) 53);
749   (ok->arr[2]) = ((char) 48);
750   (ok->arr[3]) = ((char) 0);
751   sent_6 = tcp_send(cmdFd_4, ok, 4);
752   free(ok);
753
754   done = send_outgoing_file(dataFd_1, filename_1);
755   if ((!done)) {
756     tmp_28 = newSPLArray_char( 4);
757     notOk = tmp_28;
758     (notOk->arr[0]) = ((char) 53);
759     (notOk->arr[1]) = ((char) 53);
760     (notOk->arr[2]) = ((char) 48);
761     (notOk->arr[3]) = ((char) 0);
762     sent_6 = tcp_send(cmdFd_4, notOk, 4);
763     free(notOk);
764
765     return true;
766   }
767   tmp_29 = newSPLArray_char( 4);
768   goodPacket_1 = tmp_29;
769   (goodPacket_1->arr[0]) = ((char) 50);
770   (goodPacket_1->arr[1]) = ((char) 53);
```

```
771   (goodPacket_1->arr[2]) = ((char) 48);
772   (goodPacket_1->arr[3]) = ((char) 0);
773   sent_6 = tcp_send(cmdFd_4, goodPacket_1, 4);
774   free(goodPacket_1);
775
776   return false;
777 }
778
779 bool send_outgoing_file (int dataFd_2, SPLArray_char* filename_2) {
780   bool res_12;
781   SPLArray_char* tmp_30;
782   int sent_7;
783   int read;
784   bool flag;
785   int fileS;
786   int fileFd_1;
787   int closed_1;
788   SPLArray_char* buffer_7;
789
790   fileFd_1 = gopen(filename_2, (O_CREAT | O_RDONLY));
791   flag = false;
792   if ((fileFd_1 < 0)) {
793     flag = true;
794   }
795   fileS = fileSize(filename_2);
796   if (((fileS < 0) || (fileS > 65535))) {
797     fileS = 0;
798     flag = true;
799   }
800   tmp_30 = newSPLArray_char( fileS);
801   buffer_7 = tmp_30;
802   if ((!flag)) {
803     read = gread(fileFd_1, buffer_7);
804     closed_1 = gclose(fileFd_1);
805     if (((read < 0) || (closed_1 < 0))) {
806       flag = true;
```

```
807       }
808     }
809     if ((!flag)) {
810       sent_7 = tcp_send(dataFd_2, buffer_7, fileS);
811       if ((sent_7 < 0)) {
812         flag = true;
813       }
814     }
815     free(buffer_7);
816 
817     if (flag) {
818       return false;
819     } else {
820       return true;
821     }
822     return res_12;
823 }
824 
825 int server () {
826     int res_13;
827     SPLArray_char* typeCom;
828     SPLArray_char* tmp_38;
829     bool tmp_37;
830     bool tmp_36;
831     bool tmp_35;
832     bool tmp_34;
833     bool tmp_33;
834     SPLArray_char* tmp_32;
835     SPLArray_char* tmp_31;
836     int tempCmdFd;
837     bool temp_2;
838     bool temp_1;
839     bool temp;
840     int sent_8;
841     SPLArray_char* request;
842     int recd;
```

```
843   bool properQuit;
844   SPLArray_char* port;
845   bool iQuit;
846   SPLArray_char* final;
847   SPLArray_char* filename_3;
848   int dataFd_3;
849   int cmdFd_5;
850   struct SocketAddressIP4* cmdAddr_1;
851   int closedTemp;
852   int closed_5;
853   int closed_4;
854   int closed_3;
855   int closed_2;
856   bool bound;
857   SPLArray_char* badPacket;
858   bool authenticated;
859   int allo_size_2;
860
861   tmp_31 = newSPLArray_char( 5);
862   port = tmp_31;
863   (port->arr[0]) = ((char) 52);
864   (port->arr[1]) = ((char) 52);
865   (port->arr[2]) = ((char) 52);
866   (port->arr[3]) = ((char) 52);
867   (port->arr[4]) = ((char) 0);
868   cmdFd_5 = (-1);
869   if (((port->length) > 65535)) {
870     return (-1);
871   }
872   cmdAddr_1 = get_address4(NULL, port);
873   free(port);
874
875   if ((cmdAddr_1 == NULL)) {
876     return (-1);
877   }
878   tempCmdFd = create_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
```

```
879   if ((tempCmdFd < 0)) {
880     free(cmdAddr_1);
881
882     return (-1);
883   }
884   bound = bind4(tempCmdFd, cmdAddr_1);
885   if ((!bound)) {
886     free(cmdAddr_1);
887
888     closed_2 = gclose(tempCmdFd);
889     return (-1);
890   }
891   while (true) {
892     if (!((cmdFd_5 < 0))) {
893       break;
894     }
895     cmdFd_5 = connectMeCommand(tempCmdFd, cmdAddr_1);
896   }
897   closedTemp = gclose(tempCmdFd);
898   dataFd_3 = recvDataConnection(cmdFd_5, cmdAddr_1);
899   free(cmdAddr_1);
900
901   if ((dataFd_3 <= (-1))) {
902     closed_3 = gclose(cmdFd_5);
903     return (-1);
904   }
905   authenticated = handleAuth(cmdFd_5);
906   if ((!authenticated)) {
907     closed_4 = gclose(cmdFd_5);
908     closed_4 = gclose(dataFd_3);
909     return (-1);
910   }
911   iQuit = false;
912   properQuit = false;
913   allo_size_2 = 65535;
914   while (true) {
```

```
915     if (!((!iQuit))) {
916       break;
917     }
918     tmp_32 = newSPLArray_char( 150);
919     request = tmp_32;
920     recd = tcp_recv(cmdFd_5, request);
921     typeCom = copy_byte_slice(request, 0, 4);
922     final = process_string(typeCom);
923     filename_3 = copy_byte_slice(request, 5, ((request->length) - 1));
924     free(request);
925
926     tmp_33 = is_allo(final);
927     if (tmp_33) {
928       allo_size_2 = allo_help(cmdFd_5, filename_3);
929       if (((allo_size_2 < 1) || (allo_size_2 > 65535))) {
930         allo_size_2 = 65535;
931         free(final);
932
933         iQuit = true;
934       }
935     } else {
936       tmp_34 = is_stor(final);
937       if (tmp_34) {
938         if (((allo_size_2 < 1) || (allo_size_2 > 65535))) {
939           free(filename_3);
940
941           free(final);
942
943           iQuit = true;
944         } else {
945           temp = store_help(cmdFd_5, dataFd_3, filename_3, allo_size_2);
946           if (temp) {
947             free(filename_3);
948
949             free(final);
950
```

```
951             }
952             iQuit = temp;
953           }
954         } else {
955           tmp_35 = is_size(final);
956           if (tmp_35) {
957             temp_1 = size_help(cmdFd_5, filename_3);
958             if (temp_1) {
959               free(filename_3);
960
961               free(final);
962
963             }
964             iQuit = temp_1;
965           } else {
966             tmp_36 = is_retr(final);
967             if (tmp_36) {
968               temp_2 = retr_help(cmdFd_5, dataFd_3, filename_3);
969               if (temp_2) {
970                 free(filename_3);
971
972                 free(final);
973
974               }
975               iQuit = temp_2;
976             } else {
977               tmp_37 = is_quit(final);
978               if (tmp_37) {
979                 properQuit = true;
980                 iQuit = true;
981               } else {
982                 tmp_38 = newSPLArray_char( 4);
983                 badPacket = tmp_38;
984                 (badPacket->arr[0]) = ((char) 53);
985                 (badPacket->arr[1]) = ((char) 48);
986                 (badPacket->arr[2]) = ((char) 48);
```

```
987                 (badPacket->arr[3]) = ((char) 0);
988                 sent_8 = tcp_send(cmdFd_5, badPacket, 4);
989                 free(badPacket);
990 
991                 iQuit = true;
992               }
993             }
994           }
995         }
996       }
997       free(filename_3);
998 
999       free(final);
1000 
1001    }
1002    closed_5 = gclose(cmdFd_5);
1003    closed_5 = gclose(dataFd_3);
1004    if (properQuit) {
1005      return 0;
1006    } else {
1007      return (-1);
1008    }
1009    return res_13;
1010  }
1011 
1012  bool size_help (int cmdFd_6, SPLArray_char* filename_4) {
1013    bool fail_1;
1014    SPLArray_char* tmp_42;
1015    char tmp_41;
1016    SPLArray_char* tmp_40;
1017    SPLArray_char* tmp_39;
1018    SPLArray_char* sizePacket;
1019    int sizeF;
1020    int sent_10;
1021    int sent_9;
1022    SPLArray_char* goodPacket_2;
```

```
1023    int finished;
1024    SPLArray_char* badPacket_1;
1025
1026    sizeF = fileSize(filename_4);
1027    if ((sizeF < 0)) {
1028      tmp_39 = newSPLArray_char( 4);
1029      badPacket_1 = tmp_39;
1030      (badPacket_1->arr[0]) = ((char) 53);
1031      (badPacket_1->arr[1]) = ((char) 53);
1032      (badPacket_1->arr[2]) = ((char) 48);
1033      (badPacket_1->arr[3]) = ((char) 0);
1034      sent_9 = tcp_send(cmdFd_6, badPacket_1, 4);
1035      free(badPacket_1);
1036
1037      return true;
1038    }
1039    tmp_40 = newSPLArray_char( 2);
1040    sizePacket = tmp_40;
1041    tmp_41 = makeStrFromInt(sizeF);
1042    (sizePacket->arr[0]) = tmp_41;
1043    (sizePacket->arr[1]) = ((char) 0);
1044    tmp_42 = newSPLArray_char( 6);
1045    goodPacket_2 = tmp_42;
1046    (goodPacket_2->arr[0]) = ((char) 50);
1047    (goodPacket_2->arr[1]) = ((char) 49);
1048    (goodPacket_2->arr[2]) = ((char) 51);
1049    (goodPacket_2->arr[3]) = ((char) 32);
1050    (goodPacket_2->arr[4]) = ((char) 0);
1051    finished = gstrcat(sizePacket, goodPacket_2);
1052    sent_10 = tcp_send(cmdFd_6, goodPacket_2, 6);
1053    free(sizePacket);
1054
1055    free(goodPacket_2);
1056
1057    return false;
1058  }
```

```
1059
1060 bool store_help (int cmdFd_7, int dataFd_4, SPLArray_char* filename_5,
       int allo_size_3) {
1061   bool fail_2;
1062   SPLArray_char* tmp_45;
1063   SPLArray_char* tmp_44;
1064   SPLArray_char* tmp_43;
1065   int stored;
1066   int sent_11;
1067   SPLArray_char* ok_1;
1068   SPLArray_char* notOk_1;
1069   SPLArray_char* goodPacket_3;
1070
1071   tmp_43 = newSPLArray_char( 4);
1072   ok_1 = tmp_43;
1073   (ok_1->arr[0]) = ((char) 49);
1074   (ok_1->arr[1]) = ((char) 53);
1075   (ok_1->arr[2]) = ((char) 48);
1076   (ok_1->arr[3]) = ((char) 0);
1077   sent_11 = tcp_send(cmdFd_7, ok_1, 4);
1078   free(ok_1);
1079
1080   stored = accept_incoming_file(dataFd_4, filename_5, allo_size_3);
1081   if ((stored < 0)) {
1082     tmp_44 = newSPLArray_char( 4);
1083     notOk_1 = tmp_44;
1084     (notOk_1->arr[0]) = ((char) 53);
1085     (notOk_1->arr[1]) = ((char) 53);
1086     (notOk_1->arr[2]) = ((char) 48);
1087     (notOk_1->arr[3]) = ((char) 0);
1088     sent_11 = tcp_send(cmdFd_7, notOk_1, 4);
1089     free(notOk_1);
1090
1091     return true;
1092   }
1093   tmp_45 = newSPLArray_char( 4);
```

```
1094   goodPacket_3 = tmp_45;
1095   (goodPacket_3->arr[0]) = ((char) 50);
1096   (goodPacket_3->arr[1]) = ((char) 53);
1097   (goodPacket_3->arr[2]) = ((char) 48);
1098   (goodPacket_3->arr[3]) = ((char) 0);
1099   sent_11 = tcp_send(cmdFd_7, goodPacket_3, 4);
1100   free(goodPacket_3);
1101
1102   return false;
1103 }
1104
1105 SPLArray_char* strconcat (SPLArray_char* str1_2, SPLArray_char* str2_2)
       {
1106   SPLArray_char* res_14;
1107   SPLArray_char* tmp_46;
1108   int l2_1;
1109   int l1_1;
1110   int i_19;
1111   SPLArray_char* copy_2;
1112
1113   l1_1 = gstrlen(str1_2);
1114   l2_1 = gstrlen(str2_2);
1115
1116   tmp_46 = newSPLArray_char( (l1_1 + l2_1));
1117   copy_2 = tmp_46;
1118   i_19 = 0;
1119   while (true) {
1120     if (!((i_19 < l1_1))) {
1121       break;
1122     }
1123     (copy_2->arr[i_19]) = (str1_2->arr[i_19]);
1124     i_19 = (i_19 + 1);
1125   }
1126   while (true) {
1127     if (!((i_19 < (l1_1 + l2_1)))) {
1128       break;
```

```
1129      }
1130      (copy_2->arr[i_19]) = (str2_2->arr[(i_19 - l1_1)]);
1131      i_19 = (i_19 + 1);
1132    }
1133    return copy_2;
1134 }
1135
1136 /*
1137  * Main Function, here for compilability
1138  */
1139 int main(int argc, char *argv[]) {
1140    assert(argc <= 2);
1141    int s = 0;
1142    if (argc > 1) {
1143      for(s = 0; argv[1][s] != 0; s++) { }
1144      s++;
1145    }
1146    SPLArray_char* a = newSPLArray_char(s);
1147    for(int i = 0; i < s; i++) {
1148      a->arr[i] = argv[1][i];
1149    }
1150    return Main(a);
1151 }
```

B.2 Linked-In C Wrapper Function Implementations

Below are the C implementations for the functions in console.spl, which Zufferey wrote.

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <stdbool.h>
4
5  /*
6   * Preloaded Code
7   */
8
9  typedef struct {
10   int length;
11   char arr[];
12 } SPLArray_char;
13
14 /*
15  * Procedures
16  */
17
18 bool gputs(SPLArray_char* buffer) {
19   bool null_terminated = false;
20   for (int i = 0; i < buffer->length && !null_terminated; i++) {
21     null_terminated = buffer->arr[i] != 0;
22   }
23   if (null_terminated) {
24     return fputs(buffer->arr, stdout) >= 0;
25   } else {
26     char str[buffer->length + 1]; //allocate on the stack for simplicity
27     memcpy(str, buffer->arr, buffer->length);
28     str[buffer->length] = 0;
29     return fputs(str, stdout) >= 0;
30   }
31 }
32
```

```
33 int ggets(SPLArray_char* buffer) {
34   if (fgets(buffer->arr, buffer->length, stdin) == NULL) {
35     return -1;
36   } else {
37     return strnlen(buffer->arr, buffer->length);
38   }
39 }
```

Below are the function headers for the file.spl code, which GRASShopper generated.

```
1  /*
2   * Includes
3   */
4  #include <fcntl.h>
5  #include <unistd.h>
6  #include <stdbool.h>
7  #include <stdlib.h>
8  #include <stdio.h>
9  #include <errno.h>
10 #include <string.h>
11 #include <assert.h>
12
13 /*
14  * Preloaded Code
15  */
16
17 typedef struct {
18   int length;
19   int arr[];
20 } SPLArray_int;
21
22 typedef struct {
23   int length;
24   bool arr[];
25 } SPLArray_bool;
26
27 typedef struct {
28   int length;
29   char arr[];
30 } SPLArray_char;
31
32 typedef struct {
33   int length;
34   void* arr[];
```

```
35 } SPLArray_generic;
36
37 /*
38  * Procedures
39  */
40 int gclose (int fd);
41 int gopen (SPLArray_char* pathname, int flags);
42 int gread (int fd_2, SPLArray_char* buffer);
43 int greadOffset (int fd_3, SPLArray_char* buffer_1, int offset);
44 int gwrite (int fd_4, SPLArray_char* buffer_2);
45 int gwrite2 (int fd_4, SPLArray_char* buffer_2, int size);
46 int fileSize (SPLArray_char* pathname);
```

Below are the function implementations for the file.spl code, which Zufferey and I wrote.

```
1 #include <fcntl.h>
2 #include <unistd.h>
3 #include <stdbool.h>
4 #include <stdlib.h>
5 #include "file.h"
6 #include <stdio.h>
7 #include <errno.h>
8 #include <string.h>
9 #include <sys/stat.h>
10
11
12
13 int gclose (int fd){
14   return close(fd);
15 }
16
17 int gopen (SPLArray_char* pathname, int flags){
18   //check if null terminated
19   bool null_terminated = false;
20   for(int i = 0; i < pathname->length && !null_terminated; i++) {
21     null_terminated |= (pathname->arr[i] == 0);
22   }
23   if (null_terminated) {
24     //valid C string
25     if ((flags & O_CREAT) != 0) {
26         return open(pathname->arr, flags, S_IRUSR | S_IWUSR | S_IRGRP |
               S_IROTH);
27     } else {
28         return open(pathname->arr, flags);
29     }
30   } else {
31     //create a valid C string
32     char name[pathname->length + 1]; //allocate on the stack for
           simplicity
```

```
33      for(int i = 0; i < pathname->length; i++) {
34        name[i] = pathname->arr[i];
35      }
36      name[pathname->length] = 0;
37      if ((flags & O_CREAT) != 0) {
38          return open(name, flags, S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
39      } else {
40          return open(name, flags);
41      }
42    }
43  }
44
45  int gread (int fd_2, SPLArray_char* buffer) {
46    return read(fd_2, buffer->arr, buffer->length);
47  }
48
49  int gwrite (int fd_4, SPLArray_char* buffer_2){
50    return write(fd_4, buffer_2->arr, buffer_2->length);
51  }
52
53  int gwrite2 (int fd_4, SPLArray_char* buffer_2, int size){
54    return write(fd_4, buffer_2->arr, size);
55  }
56
57  int greadOffset (int fd_3, SPLArray_char* buffer_1, int offset){
58    return pread(fd_3, buffer_1->arr, buffer_1->length, offset);
59  }
60
61  int fileSize(SPLArray_char* pathname) {
62    struct stat st;
63    bool null_terminated = false;
64    for(int i = 0; i < pathname->length && !null_terminated; i++) {
65      null_terminated = (pathname->arr[i] == 0);
66    }
67
68    if (null_terminated) {
```

```
69      if (stat(pathname->arr, &st) == 0) {
70        return (int)st.st_size;
71      }
72    } else {
73      char name[pathname->length + 1]; //allocate on the stack for
            simplicity
74      memcpy(name, pathname->arr, pathname->length);
75      name[pathname->length] = 0;
76      if (stat(name, &st) == 0) {
77        return (int)st.st_size;
78      }
79    }
80    return -1;
81  }
```

Below is the function implementation for the makeStr.spl code, which I wrote.

```
1 #include <stdlib.h>
2
3 int makeStrFromInt(int myInt){
4   char c = myInt + '0';
5   return c;
6 }
```

Below are the function implementations for the socket.spl, which Zufferey wrote.

```
1  #include <stdbool.h>
2  #include <stdlib.h>
3  #include <assert.h>
4  #include <errno.h>
5  #include <string.h>
6
7  #include <stdio.h>
8  #include <sys/types.h>
9  #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <arpa/inet.h>
13
14 /*
15  * Preloaded Code
16  */
17
18 typedef struct {
19   int length;
20   char arr[];
21 } SPLArray_char;
22
23 SPLArray_char* _newSPLArray_char(int size) {
24   SPLArray_char* a = (SPLArray_char*)malloc(sizeof(SPLArray_char) + size
         * sizeof(char));
25   assert(a != NULL);
26   a->length = size;
27   return a;
28 }
29
30 typedef struct SocketAddressIP4 {
31   //int sin4_addr_lower;
32   //int sin4_addr_upper;
33   int sin4_addr;
34   int sin4_port;
```

```
35 } SocketAddressIP4;
36
37 typedef struct SocketAddressIP6 {
38   SPLArray_char* sin6_addr;
39   int sin6_flowinfo;
40   int sin6_port;
41   int sin6_scope_id;
42 } SocketAddressIP6;
43
44 /*
45  * Procedures
46  */
47
48 void to_grass_addr(struct sockaddr_in* c, struct SocketAddressIP4*
       address) {
49   address->sin4_port = ntohs(c->sin_port);
50   //address->sin4_addr_upper = (int)(c->sin_addr.s_addr >> 32);
51   //address->sin4_addr_lower = (int)(c->sin_addr.s_addr);
52   address->sin4_addr = c->sin_addr.s_addr;
53 }
54
55 void from_grass_addr(struct SocketAddressIP4* address, struct
       sockaddr_in* c) {
56   c->sin_family = AF_INET;
57   c->sin_port = htons((short) address->sin4_port);
58   //c->sin_addr.s_addr = (((long)address->sin4_addr_upper) << 32) |
         address->sin4_addr_lower;
59   c->sin_addr.s_addr = address->sin4_addr;
60   c->sin_zero[0] = 0;  c->sin_zero[1] = 0;
61   c->sin_zero[2] = 0;  c->sin_zero[3] = 0;
62   c->sin_zero[4] = 0;  c->sin_zero[5] = 0;
63   c->sin_zero[6] = 0;  c->sin_zero[7] = 0;
64 }
65
66 void to_grass_addr6(struct sockaddr_in6* c, struct SocketAddressIP6*
       address) {
```

```
67    address->sin6_port = c->sin6_port;
68    address->sin6_flowinfo = c->sin6_flowinfo;
69    address->sin6_scope_id = c->sin6_scope_id;
70    for(int i = 0; i < 16; i++) {
71      address->sin6_addr->arr[i] = c->sin6_addr.s6_addr[i];
72    }
73  }
74
75  void from_grass_addr6(struct SocketAddressIP6* address, struct
        sockaddr_in6* c) {
76    c->sin6_family = AF_INET6;
77    c->sin6_port = (short)address->sin6_port;
78    c->sin6_flowinfo = address->sin6_flowinfo;
79    c->sin6_scope_id = address->sin6_scope_id;
80    for (int i = 0; i < 16; i++) {
81      c->sin6_addr.s6_addr[i] = address->sin6_addr->arr[i];
82    }
83  }
84
85  struct SocketAddressIP4* get_address4(SPLArray_char* node, SPLArray_char
        * service) {
86    if (node == NULL) {
87      struct sockaddr_in sa;
88      memset(&sa, 0, sizeof sa);
89      sa.sin_family = AF_INET;
90      sa.sin_port = htons(atoi(service->arr));
91      sa.sin_addr.s_addr = htonl(INADDR_ANY);
92
93      //printf("(local) IP address is: %s\n", inet_ntoa(sa.sin_addr));
94
95      struct SocketAddressIP4* address = malloc(sizeof(struct
            SocketAddressIP4));
96      to_grass_addr(&sa, address);
97
98      return address;
99
```

```
100   } else {
101     struct addrinfo hint;
102     memset(&hint, 0, sizeof hint);
103     hint.ai_family = AF_INET;
104     hint.ai_socktype = SOCK_DGRAM; //SOCK_STREAM
105     hint.ai_protocol = IPPROTO_UDP; //IPPROTO_TCP
106     struct addrinfo *servinfo = NULL;
107     int rv;
108     if (node == NULL) {
109       rv = getaddrinfo(NULL, service->arr, &hint, &servinfo);
110     } else {
111       rv = getaddrinfo(node->arr, service->arr, &hint, &servinfo);
112     }
113     if (rv != 0) {
114       return NULL;
115     } else {
116       struct addrinfo *p;
117       for(p = servinfo; p != NULL; p = p->ai_next) {
118         if (p->ai_family == AF_INET) {
119           struct SocketAddressIP4* address = malloc(sizeof(struct
                SocketAddressIP4));
120           assert(address != NULL);
121           struct sockaddr_in* a = (struct sockaddr_in*)p->ai_addr;
122           //printf("(remote) IP address is: %s\n", inet_ntoa(a->sin_addr
                ));
123           to_grass_addr(a, address);
124           freeaddrinfo(servinfo);
125           return address;
126         }
127       }
128       freeaddrinfo(servinfo);
129       return NULL;
130     }
131   }
132 }
133
```

```
134 struct SocketAddressIP6* get_address6(SPLArray_char* node, SPLArray_char
      * service) {
135   struct addrinfo hint;
136   memset(&hint, 0, sizeof hint);
137   hint.ai_family = AF_INET6;
138   hint.ai_socktype = SOCK_DGRAM; //SOCK_STREAM
139   hint.ai_protocol = IPPROTO_UDP; //IPPROTO_TCP
140   struct addrinfo *servinfo = NULL;
141   //TODO
142   int rv;
143   if (node == NULL) {
144     rv = getaddrinfo(NULL, service->arr, &hint, &servinfo);
145   } else {
146     rv = getaddrinfo(node->arr, service->arr, &hint, &servinfo);
147   }
148   if (rv != 0) {
149     return NULL;
150   } else {
151     struct addrinfo *p;
152     for(p = servinfo; p != NULL; p = p->ai_next) {
153       if (p->ai_family == AF_INET6) {
154   struct SocketAddressIP6* address = malloc(sizeof(struct
        SocketAddressIP6));
155   assert(address != NULL);
156   address->sin6_addr = _newSPLArray_char(16);
157   struct sockaddr_in6* a = (struct sockaddr_in6*)p->ai_addr;
158   to_grass_addr6(a, address);
159   freeaddrinfo(servinfo);
160   return address;
161       }
162     }
163     freeaddrinfo(servinfo);
164     return NULL;
165   }
166 }
167
```

```
168 bool bind4(int fd, struct SocketAddressIP4* address) {
169   struct sockaddr_in sa;
170   from_grass_addr(address, &sa);
171   //printf("IP address is: %s\n", inet_ntoa(sa.sin_addr));
172   //printf("port is: %d\n", (int) ntohs(sa.sin_port));
173   return bind(fd, (struct sockaddr *)&sa, sizeof sa) == 0;
174 }
175
176 bool bind6(int fd, struct SocketAddressIP6* address) {
177   struct sockaddr_in6 sa;
178   from_grass_addr6(address, &sa);
179   return bind(fd, (struct sockaddr *)&sa, sizeof sa) == 0;
180 }
181
182 int create_socket(int inet_type, int socket_type, int protocol) {
183   return socket(inet_type, socket_type, protocol);
184 }
185
186 int udp_recv4(int fd, SPLArray_char* msg, struct SocketAddressIP4* from)
      {
187   struct sockaddr_in sa;
188   int len = sizeof sa;
189   int res = recvfrom(fd, msg->arr, msg->length, 0, (struct sockaddr*)&sa
        , &len);
190   if (res >= 0) {
191     assert(len == (sizeof sa));
192     to_grass_addr(&sa, from);
193   }
194   return res;
195 }
196
197 int udp_recv6(int fd, SPLArray_char* msg, struct SocketAddressIP6* from)
      {
198   struct sockaddr_in6 sa;
199   int len = sizeof sa;
200   int res = recvfrom(fd, msg->arr, msg->length, 0, (struct sockaddr*)&sa
```

```
      , &len);
201   if (res >= 0) {
202     assert(len == (sizeof sa));
203     to_grass_addr6(&sa, from);
204   }
205   return res;
206 }
207
208 int udp_send4(int fd, SPLArray_char* msg, int len, struct
      SocketAddressIP4* address) {
209   struct sockaddr_in sa;
210   from_grass_addr(address, &sa);
211   return sendto(fd, msg->arr, len, 0, (struct sockaddr*) &sa, sizeof sa)
        ;
212 }
213
214 int udp_send6(int fd, SPLArray_char* msg, int len, struct
      SocketAddressIP6* address) {
215   struct sockaddr_in6 sa;
216   from_grass_addr6(address, &sa);
217   return sendto(fd, msg->arr, len, 0, (struct sockaddr*) &sa, sizeof sa)
        ;
218 }
219
220 bool glisten(int fd,  int backlog) {
221   return listen(fd, backlog) == 0;
222 }
223
224 int accept4(int fd, struct SocketAddressIP4* address) {
225   struct sockaddr_in sa;
226   int len = sizeof sa;
227   int res = accept(fd, (struct sockaddr*)&sa, &len);
228   if (res >= 0) {
229     to_grass_addr(&sa, address);
230   }
231   return res;
```

```
232 }
233
234 int accept6(int fd, struct SocketAddressIP6* address) {
235   struct sockaddr_in6 sa;
236   int len = sizeof sa;
237   int res = accept(fd, (struct sockaddr*)&sa, &len);
238   if (res >= 0) {
239     to_grass_addr6(&sa, address);
240   }
241   return res;
242 }
243
244 bool connect4(int fd, struct SocketAddressIP4* address)
245 {
246   struct sockaddr_in sa;
247   from_grass_addr(address, &sa);
248
249   //char ipPrint[INET_ADDRSTRLEN];
250   //inet_ntop(AF_INET, &(sa.sin_addr), ipPrint, INET_ADDRSTRLEN);
251   //printf("IP is: %s\n", ipPrint);
252   //printf("port is: %d\n", (int) ntohs(sa.sin_port));
253
254   int res = connect(fd, (struct sockaddr*) &sa, sizeof sa);
255   //if (res < 0) {
256   //  printf("error: %s\n", strerror(errno));
257   //}
258   return res == 0;
259
260 }
261
262 bool connect6(int fd, struct SocketAddressIP6* address)
263 {
264   struct sockaddr_in6 sa;
265   from_grass_addr6(address, &sa);
266   return connect(fd, (struct sockaddr*) &sa, sizeof sa) == 0;
267 }
```

```
268
269 int tcp_send(int fd, SPLArray_char* msg, int len) {
270   //printf("sending is: (%d) %s\n", len, msg->arr);
271   return send(fd, msg->arr, len, 0);
272 }
273
274 int tcp_recv(int fd, SPLArray_char* msg) {
275   return recv(fd, msg->arr, msg->length, 0);
276 }
```

B.3 Helper Script

Here is the shell script I wrote for compiling the client and server with all the linked-in files.

```
1 gcc -c ${1} ../lib/console.c -o console.o
2 gcc -c ${1} ../lib/file.c -o file.o
3 gcc -c ${1} ../lib/socket.c -o socket.o
4 gcc -c ${1} ../lib/makeStr.c -o makeStr.o
5 gcc -c ${1} ftpClient.c -o ftpClient.o
6 gcc -c ${1} ftpServer.c -o ftpServer.o
7 gcc ${1} console.o file.o socket.o makeStr.o ftpServer.o -o ftpServer
8 gcc ${1} console.o file.o socket.o makeStr.o ftpClient.o -o ftpClient
```