CSC/ECE 573 (001) – Internet Protocols
Fall 2018
Final Report

File Transfer Protocol over UDP
Supporting Out-of-Order Reception

Submitted: December 3, 2018

James Cross
Daniel Monazah
Nitesh Reddy
Deepak Patil

Introduction

Bandwidth is a very valuable resource, and networks must not waste it on redundant transfers and unnecessary overhead bits which are inherent with certain protocols. Especially in large enterprise networks, a lot of the bandwidth is wasted on control fields of “heavier" protocols, sacrificing throughput without significantly improving the user experience. For example, TCP introduces high overhead into the system because of all of the services it provides. However, these services are not always required, and a higher throughput may be more desirable. In addition, many versions of lighter weight FTPs such as TFTP discard out-of-order packets, requiring retransmissions of correctly-received packets, which is problematic for conserving bandwidth.

The purpose of this design project is to develop a solution for efficient utilization of available bandwidth and to reduce waste by implementing a lightweight solution in the application of remote directory exploration and file transfers.

Design

To meet the design requirements, UDP was used to implement a low-overhead but reliable file transfer server supporting out-of-order reception of segments. Reliability is achieved using the Selective Repeat ARQ at the application layer. Furthermore, buffering out-of-order packets within a window of blocks keeps retransmissions to a minimum, reducing the amount of required bandwidth. Since this project focuses on file transfers, the conventional client-server model is used to explore the directory structure and pull data. Clients are able to interact with the server through a UNIX-like Command Line Interface (CLI) to explore the accessible directories and files on the server.

Figure 1 below shows the high-level block diagram of the system. Information flows in the direction of the arrows between subsystem blocks. Requests and data are eventually sent using socket communications after calling the selective repeat API.

Each major aspect of the project is described in detail in the following sections.

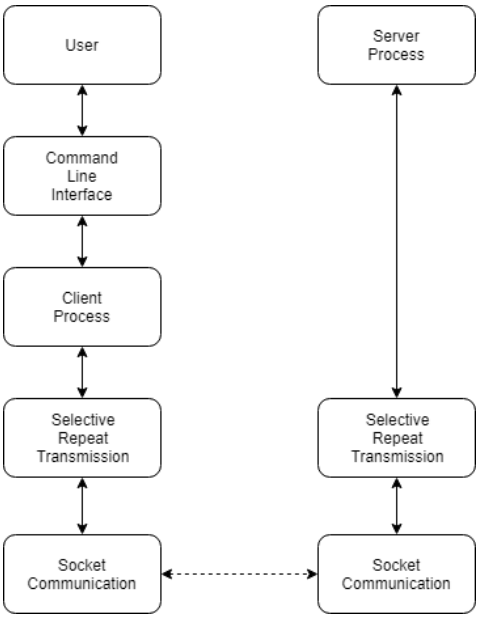


Figure 1: Block diagram showing high-level operation of the system

Server

On initialization, the server executable reads the entire directory structure and initializes a JSON tree in its heap. The server then initializes a socket using the local machine’s IPv4 address and a static UDP port number. Once initialized, it binds to its socket and listens for incoming messages from clients. The server is iterative in nature, which means that while multiple clients can communicate with the server, it can only serve one client at a time. The server expects GET and DGET commands on its end, used for fetching files and directory contents as JSON strings respectively. It parses the commands, fills up the buffer with the appropriate output, and does a blocking send back to the client IP and port.

A detailed flowchart of the server side can be found in Appendix A.

Client

At startup, the client determines the server’s socket by requesting its address and port from user input. Once the socket is initialized, the client requests the initial directory structure from the server. After initializing its directory tree, the client uses the CLI to interact with the user. User commands are checked for validity before invoking the appropriate response. List commands are handled locally. The directory tree will already be up-to-date, so the client can simply print any file names and directories that are present.

Changing directories is more complex. First the client must check to see if the directory is empty. If it's not empty, the client can simply change to that directory. However, if the directory is empty, the client has to see if this if the first time checking that directory. If it is, the JSON message sent by the server used to build the tree may not have had enough space to include this directory's contents; therefore, the client must send a DGET command to the server, requesting that it send information about this directory. If this is not the first time checking the directory, then the client knows that this is, in fact, an empty directory, and it will change to it without sending a DGET command.

Once the user has navigated to the correct folder, they can download a file using the `get` command. This results in sending a GET message with the absolute file path to the server. The server will respond with the desired file, sent using the selective repeat API. The client will then proceed to save the sent file to the client's local directory.

A detailed flowchart of the client side can be found in Appendix B.

CLI

The CLI is a lightweight interface that allows the user to navigate the file hierarchy and select the desired files to download. The CLI supports several UNIX-like, predefined commands: `cd`, `ls`, `get`, `close`, `quit`, and `exit`. These commands can be used to request JSON objects, which are transferred to the client on demand.

The `close`, `quit`, and `exit` commands simply provide the user a clean way to stop execution of the client process. The `cd` command provides a way to change to and from parent and subdirectories. The `ls` command lists the entire contents of a directory: both files and nested folders. Lastly, the `get` command requests the specified file from the server, and the client begins to download it.

Selective Repeat

Selective Repeat (SR) is a reliable data transfer protocol that introduces some overhead but has several advantages over Go-Back-N. SR maintains a buffer at the receiver's end to handle out-of-order delivery. Instead of discarding correctly-received but out-of-order packets, SR buffers them until it receives the missing packets for eventual in-order delivery to the application. This saves bandwidth by reducing the amount of unnecessary retransmissions.

In addition, SR maintains a timer for each packet. In this implementation, sequence numbers for packets from 0-3 with a window size of 2 were used. These parameters were kept configurable so they can be changed if needed. Packet timeout is set to 10 seconds, but it can also be configured with only minor adjustments in the code. Current buffer size is kept at 2048 bytes. This buffer can also be configured. Another configurable parameter is the maximum segment size. By default, it is set to 1500, but it could be changed as needed.

It is advised that only the admin of the application with access to the source code should change these parameters to meet the end-user's needs. Multi-threading is used for the transfer of packets so that a number of packets can be processed at the same time. A locking mechanism is also used to synchronize each thread's access to the window.

The general block diagram for selective repeat is shown in Appendix C. The small, circled numbers show the logical flow between the sender and the receiver. Appendix D is a flowchart showing the implementation of the sender, whereas Appendix E details the behavior of the receiver. Both of these diagrams have subflows. These sub-flows show how the window, packets, and ACKs are handled in this project's implementation. Subflow A in the sender flowchart explains the Packet Handler Thread, which handles multiple packets. Subflow C, originating from subflow A, shows how an individual packet is handled. Subflow B represents the flow of handling ACKs. Subflow A in the receiver flowchart represents the flow of packet handling at the receiver side.

Directory Tree JSON Representation

Conventional file servers usually use markup language files such as XML files to convey a directory tree starting from the root. This format describes the depth of the tree by having nested <dir> headers. Files are represented through the <file> header, and the delimiter for all types are given by forward slash and possibly followed by the header type name. They are usually be represented in the following way:

```
<dir name="Test">
  <file name="Edulinq.pdf" />
  <file name="Microsoft NET Framework 4.pdf" />
  <dir name="Test1">
    <file name="Sams - Silverlight 4 Unleashed.pdf" />
    <file name="Silverlight 2 Unleashed.pdf" />
    <file name="WhatsNewInSilverlight4.pdf" />
  </dir>
```

```
  <dir name="Test1 - Copy">
  </dir>
</dir>
```

While XML formats are very user readable, they are very taxing on bandwidth since a large amount of redundant information is being transferred in the form of the XML headers and delimiters to convey a relatively small amount of information. We could have been using this data to send more useful information instead. As the number of objects (both directories and files) increases, the percentage of actual useful information reduces considerably. This format suffers from diminishing returns.

An alternative way to handle transferring the directory tree, for UNIX-based servers especially, is to send a UNIX list of all the file names with their absolute paths, starting from the root (represented as '.') in one payload. This format is as follows:

```
./Test/
./Test/Edulinq.pdf
./Test/Microsoft NET Framework 4.pdf
./Test/Test1/
./Test/Test1/Sams - Silverlight 4 Unleashed.pdf
./Test/Test1/Silverlight 2 Unleashed.pdf
./Test/Test1/WhatsNewInSilverlight4,pdf
./Test/Test1 - Copy/
```

This format packs in a lot of redundant information as well, but it is comparatively lighter than XML. Since the parent directory name is repeated for every file within the directory, it uses up a lot of buffer space by conveying the same repeated messages. This is required for this format since it doesn't have the intelligence of depth within the formatting and hence the list adds the full relative paths from the root of the tree for the client to interpret and create a directory tree on its side.

Javascript Object Notation (JSON) is very lightweight data-interchange format. It is deeply rooted in REST APIs, and in applications today, it is used to handle indexing and transferring objects as trees to requesting clients.

It uses very little redundant information as opposed to XML, and it doesn't repeat information incessantly as in the case of the UNIX file list format. The JSON format also has the intelligence of representing depth in its structure through the simple use of curly braces '{' and '}'. This results in decreased user-readability of the received payload, but the significant reduction in redundant information easily outweighs the need for readability of the payload. It is also important to understand that user-readability of the payload is only really useful during debugging, so it cannot possibly affect the layman's experience of using a file server to simply download a file. That is, when using the application, the end-user will not need to read the payload of the JSON. The application will interpret the data and translate it into useful information.

Thus, it is only natural to go ahead with the JSON representation to transfer the directory tree structure. In this implementation, three types of JSON values are utilized: JSON objects, general JSON name-value pairs, and JSON strings.

JSON objects, given by enclosed '{' and '}', are used to represent whole directories. General name-value pairs are used to generalize strings and objects, and JSON strings are used for file names. Directory names are given in the name part of the name-value pair, whereas files have an 'F' in the name, and the string for the file name is the value. The JSON directory tree representation will be as follows:

```
{"Test":{"Test1":{"F":"File.pdf"},"Test1 - Copy":
{"F":"File.pdf"}}, "F":"Edulinq.pdf"}
```

Note the lack of repetition of the directory name and the explicit expression of a tag for directory and file names.

Depth is implied in the use of '{' in the JSON string, and it also makes it easier for parsing on the client side.

Procedural Generation of JSON String

In addition to using a lightweight representation of the tree, another area where the conservation of bandwidth needs to be considered is the transmission of the directory tree itself. Conventional methods simply format the entire directory tree into the syntax described before.

It is often seen that when the number of files and directories is large, the entire directory tree is transferred as a very large file to client (segmented in lower layers). The server doesn't take into consideration the depth of where the client could possibly traverse in the session.

It is possible that the server had conveyed the entire tree for the requested directory of maximum depth 8 containing many directories and even more files, and the client just wanted to perform a download on a file present in a directory directly below the

root. The client may request that file and end the session. This is wasteful since the rest of the information of received from the server is rendered useless for that session.

The only way a server could possibly provide exactly the same amount of the information that the client needed in the first place is if it could accurately predict what the client wanted. This isn't possible as it requires the server to have foresight. Instead, the server does the best it can to mitigate the issue of wasted directory tree information.

The JSON directory tree string is thus procedurally generated on demand on the client side. Initially, when the client starts, it requests the directory tree of the root '.' by default. In lieu of sending the JSON directory tree in its entirety as a string, the server sends a "shallow" tree with a maximum depth of 4.

The shallow tree is created through what is essentially a breadth-first search of all the nodes in the directory tree. While the JSON object string is being generated, it continuously checks to see if adding the contents of a particular directory (and assuming the next nodes in the search sequence are empty directories and don't have much to contribute to the string) will cause it to overflow the buffer. If so, the server shouldn't add the contents of the current directory into the string.

This adds two dimensions when procedurally generating the JSON directory tree string. One dimension is the actual length limitation of the string buffer for the transfer. The second dimension is the depth in the directory tree. Directories whose content couldn't be transferred in the current transaction of the string buffer can be requested later by the client if the client tries to access the directory, but until that point, it is equivalent to being empty to the client.

This method transmits the number of segments as it would if the entire directory was being transmitted in the worst case of having to traverse the whole tree. Shallow trees, however, are quicker to transmit, and if the client only needs a file within the already generated tree, then there is no further need to transfer more information. Overall, this saves bandwidth by withholding information.

One trade-off worth noting is that the client cannot differentiate between a directory that is actually empty and a directory which has not been requested yet. If the client sees an empty directory, it must request the contents of the tree underneath this specific node.

On-demand building of the directory has its disadvantages. First, if the user visited the file server before and is certain on a second visit that the tree has not been changed, the user cannot enter the command for changing directories directly to a folder beyond the depth of the currently generated tree. A problem could arise when the client repeatedly traverses the directory which in reality is empty and the client would have to keep requesting the contents. This issue is addressed simply by placing a restriction that all directories must be requested only once.

In practice, the user does not realize when the empty directories are requested, assuming that there are no heavy losses in the network at that point of time. Else, the user may notice delays between each change directory command. When the user traverses the tree using the `cd` command, the JSON directory tree is internally downloaded and attached to the already constructed tree. If the entire tree has been traversed, the client would have the whole picture of the tree on the server.

Implementation

Both the C and Python programming languages were used for the project's development. To maintain compatibility across devices, all Python code was written for version 3.6. The two languages interact using a ctypes wrapper.

Development was done using the Windows Operating System, which is the target OS for this project. The compiled Dynamic Link Libraries (DLLs), which allow communication between the languages, were built using DevC++ for the Windows OS and are not intended for use on other systems.

This project is comprised of many source files, but a few essentials ones are highlighted below.

Key Source Files

UFTP_Server.py
: Houses all functionality abstracted as the application server. It opens a socket and accepts and responds to incoming messages from clients.

UFTP_Client.py
: Houses all functionality abstracted as the application client. It connects to a

user-defined server address via socket communications and exchanges messages with the server..

CLI.py
: Uses the cmd module to create a Command Line Interface to interpret user commands.

s_JSONTreeBuild.c
: Contains server directory tree initialization modules.

JSONTreeBuild.c
: Contains client initialization modules and JSON string parsing logic.

serverperprocessmodules.c
: Consists of JSON string fetch method for procedural generation.

Sender.py
: This file calls the send() function which in turn calls SenderWindow.py, SenderPacketHandler.py, and SenderACKHandler.py to handle sender-side flow for SR.

Receiver.py
: This file calls the receive() function which in turn calls ReceiverWindow.py and ReceiverPacketHandler.py to handle receiver-side flow for SR.

Demonstration

To demonstrate the proper functionality of this project, several clients will navigate the file directory and request various files from the server running on a separate machine.

Several machines will be set up: one as the server and the others as clients. A folder will be made on the server side, and an arbitrary text file will be stored in it. Then, the server process will start and run throughout the demonstration.

Next, each client process will run, and the appropriate commands will be entered. In order to view the contents of the current folder, use `ls`. This will display the name of the subfolder(s) the client can navigate to using the `cd` command. The client can then initiate the custom TFTP file transfer using the `get` command followed by the file name.

Debugging and verification are done by printing out information to the console during runtime to allow the users and developers to see how the system behaves. Print statements are used by the client to notify the user that the transfer is pending by displaying the appropriate message. The client process also indicates when the file has been successfully received. Additional debug statements are present throughout the code and can be used as needed.

To simulate a real network, the code uses random processes to drop packets or force bit errors. When a packet is intentionally dropped or altered, a message is printed to the console to let the user know, and the system will detect the missing or incorrect packet and recover from it. The protocol will continue transmitting until the file is successfully transferred from the server to the client.

Results and Discussion

The selective repeat ARQ protocol has better link utilization than the Stop-And-Wait protocol used in conventional lightweight FTPs (TFTP), and it supports out-of-order reception which aids in the conservation of valuable bandwidth; bandwidth that would've been wasted in the event of losses for Go-Back-N.

This can be demonstrated through a file transfer of a file 100 KB in size, which is split into segments of 2048 bytes. Suppose the transmission rate seen at the transport layer is 1 Mbps and the round trip time for ACKs at the transport layer is 1 second. The link utilization for the Stop and Wait protocol would be 1.6%. This is bound to be even lower in the case of errors. In the case of selective repeat, this implementation uses a window of 2 buffers for transfer, and in the case of no-errors on the channel, the link utilization doubles (multiple of the window size) and becomes 3.2% and is proportional with the window size. This, of course, has a trade-off of storing out-of-order packets in more buffers as well as a more complex implementation.

In addition, the redundant information percentage for the XML format, UNIX list format, and JSON tree string format were compared for a test directory tree with names "Test" appended by a depth in the tree and arbitrarily named files. It was seen that the XML format packed 48.34% redundant information (including the line feed characters), the UNIX list format packed 40.69% redundant information, and the JSON directory tree format packed merely 11.17% redundant information.

Related Work and References

J. F. Kurose and K. W. Ross, *Computer networking: a top-down approach*. Hoboken, NJ: Pearson, 2017.

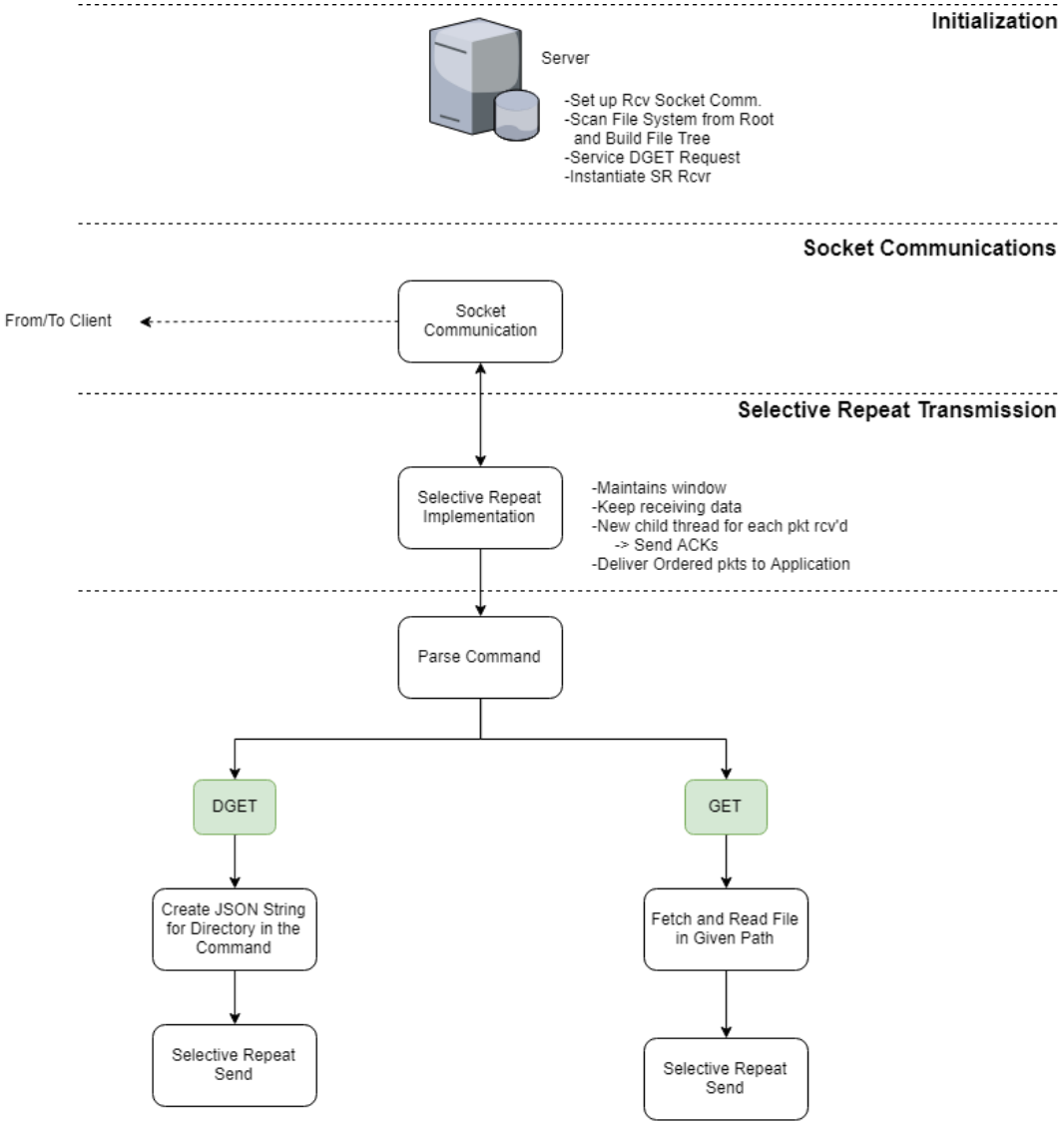
"Python 3.6.7 documentation," *Python*, 2018. [Online]. Available: https://docs.python.org/3.6/. [Accessed: Oct-2018].

"Python 2.7.15 documentation," *Python*, 2018. [Online]. Available: https://docs.python.org/2.7/. [Accessed: Oct-2018].

RFC1350 The TFTP Protocol (Revision 2). K. Sollins. July 1992. (Format: TXT=24599 bytes) (Obsoletes RFC0783) (Updated by RFC1782, RFC1783, RFC1784, RFC1785,RFC2347, RFC2348, RFC2349) (Also STD0033) (Status: INTERNET STANDARD) (DOI: 10.17487/RFC1350).

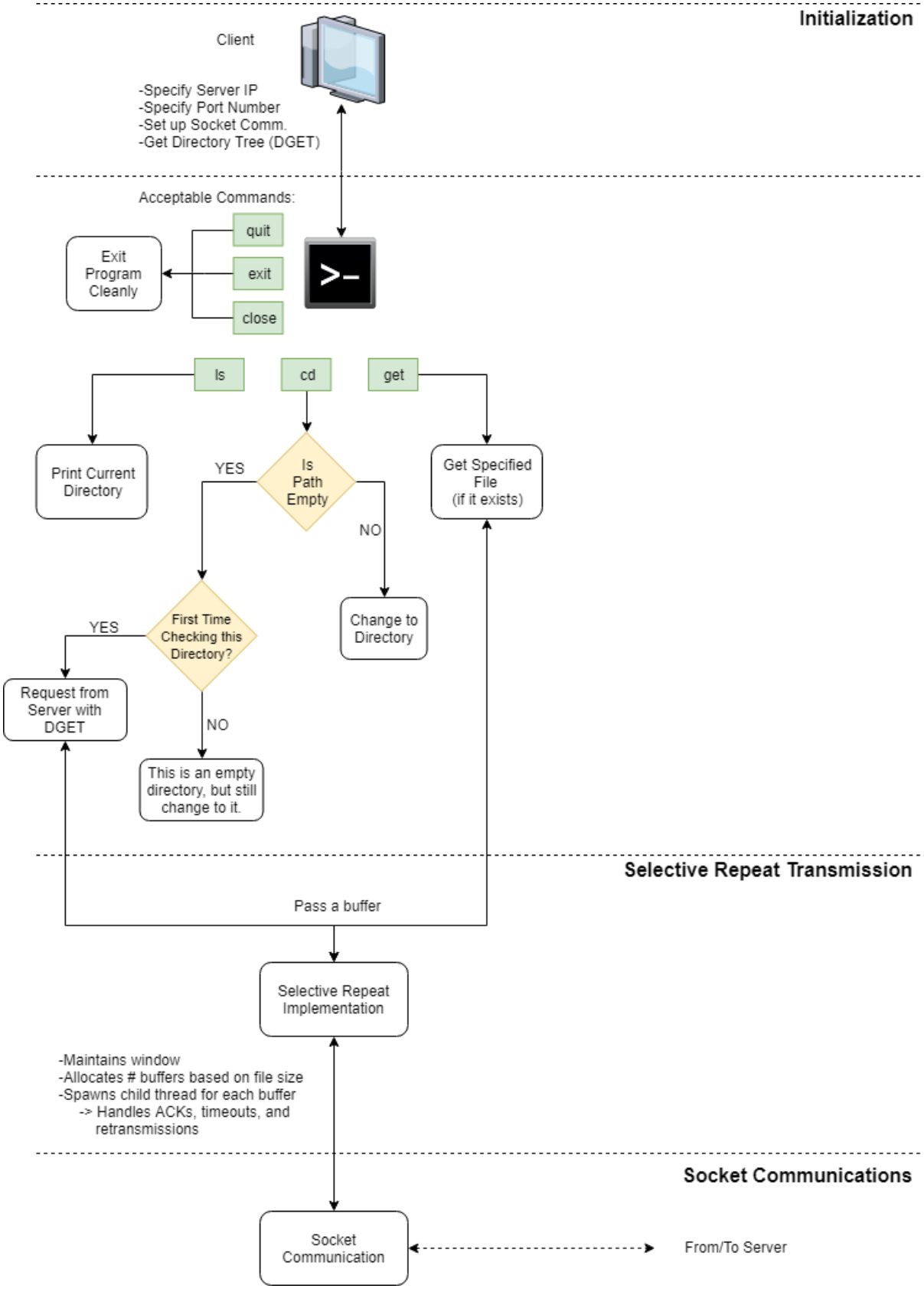
RFC768 User Datagram Protocol. J. Postel. August 1980. (Format: TXT=5896 bytes) (Also STD0006) (Status: INTERNET STANDARD) (DOI: 10.17487/RFC0768).

Appendix A



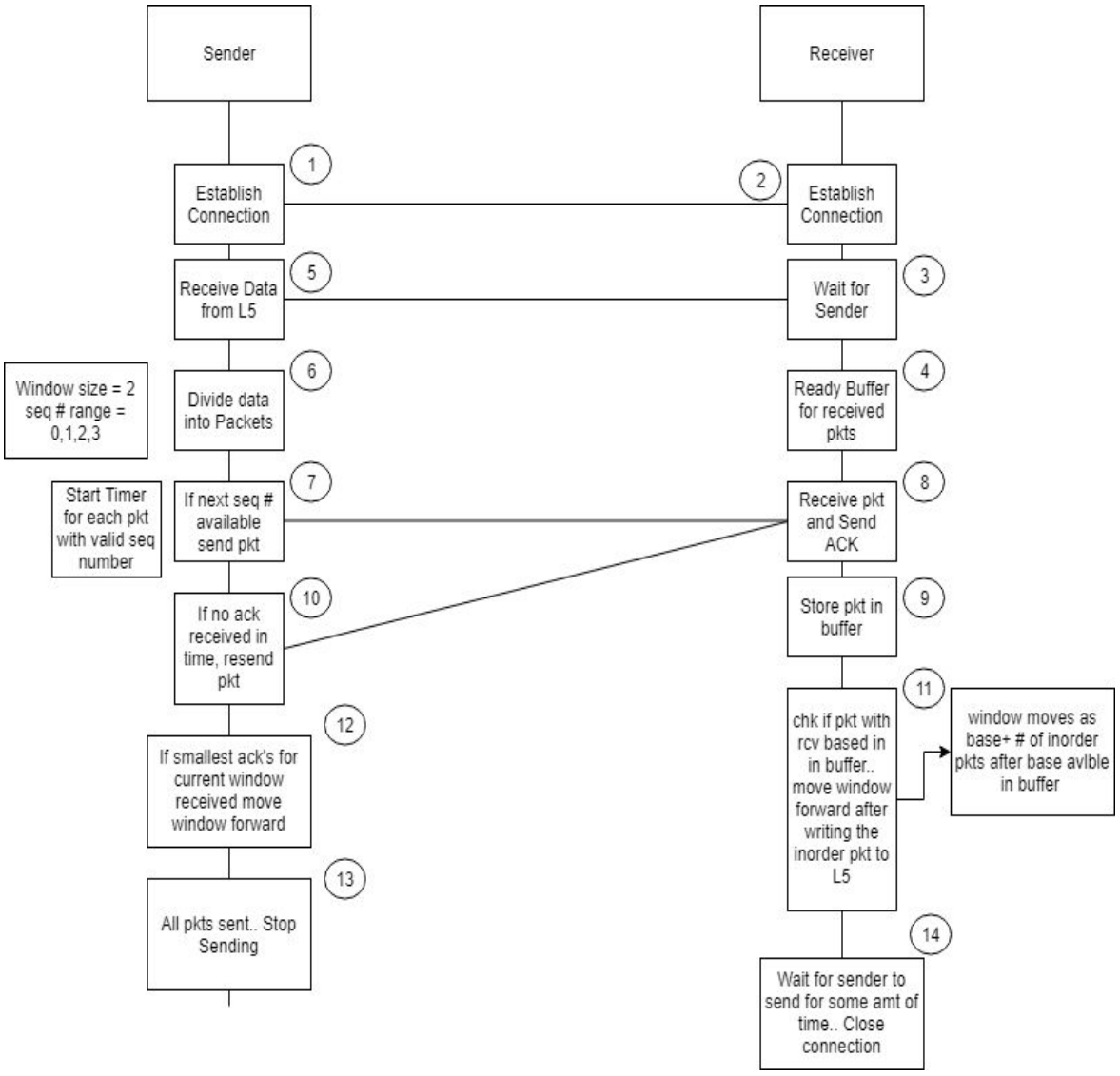
Detailed flowchart showing server functionality.

Appendix B



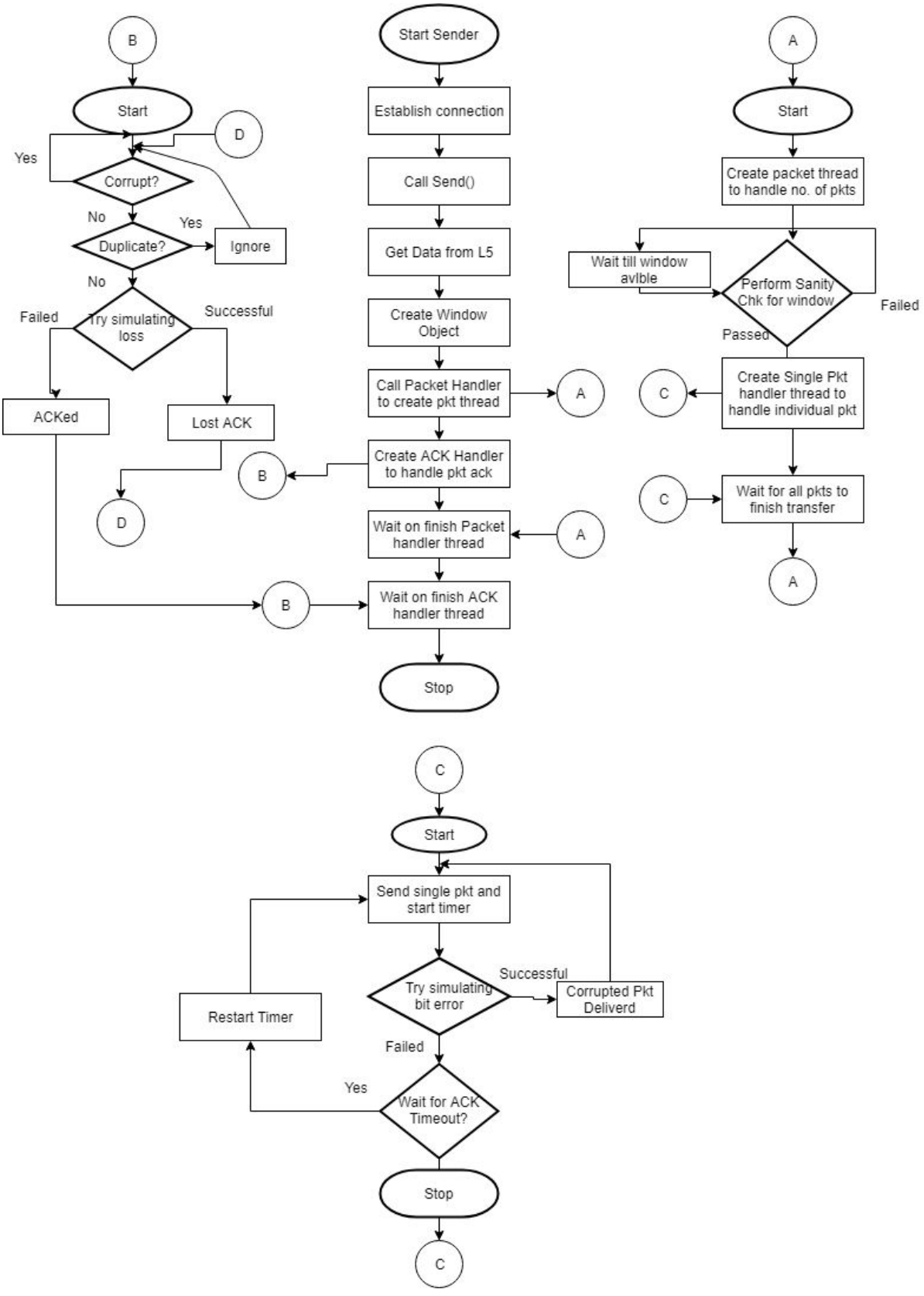
Detailed flowchart showing client functionality.

Appendix C



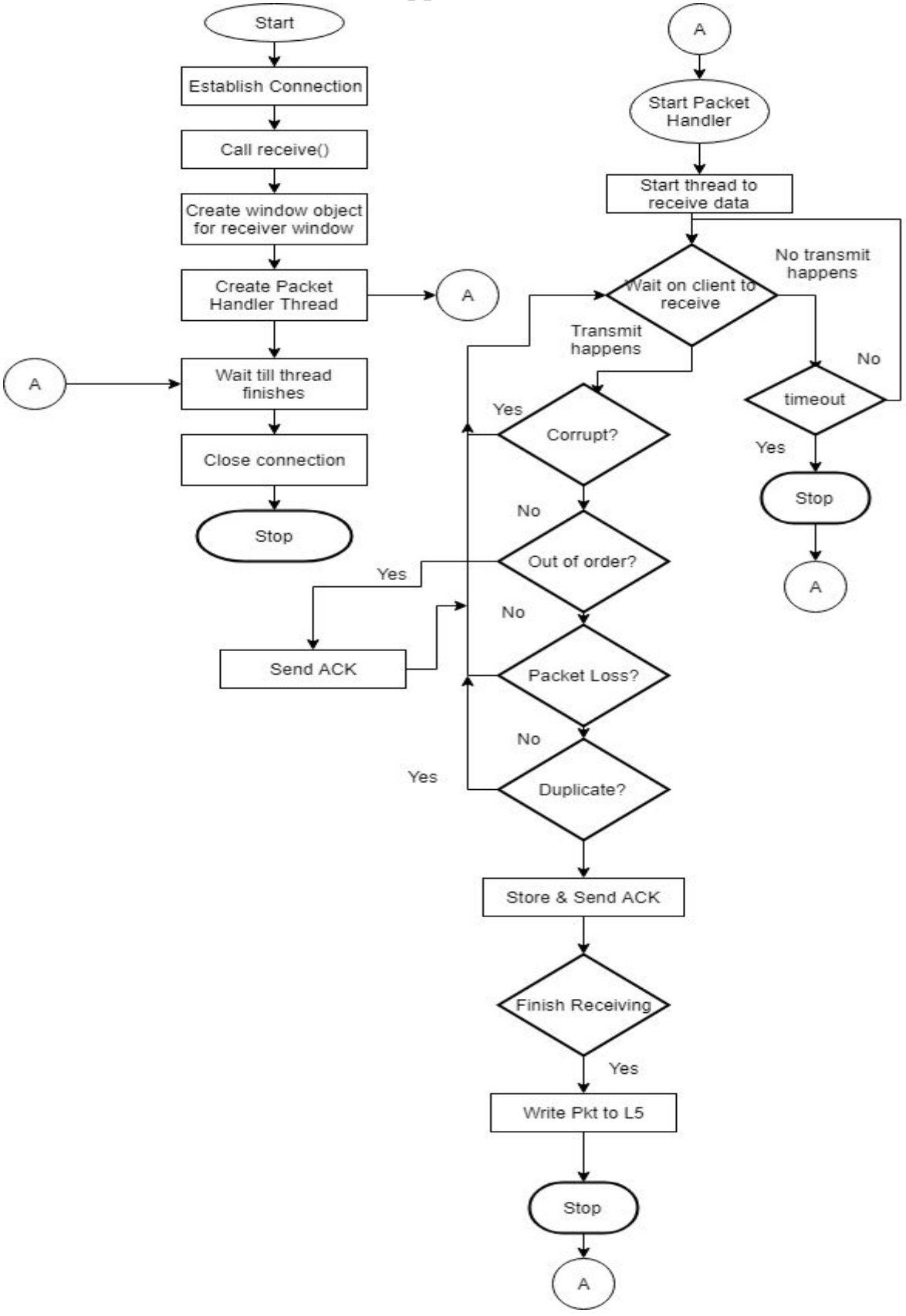
Selective Repeat (SR) Block Diagram

Appendix D



Sender Flow Chart

Appendix E



Receiver Flow Chart