Data Transmission Simulation in C

A comprehensive model of data transmission

Peter Buonaiuto

October 1, 2023

UID 10097626

**Table of Contents**

System Documentation

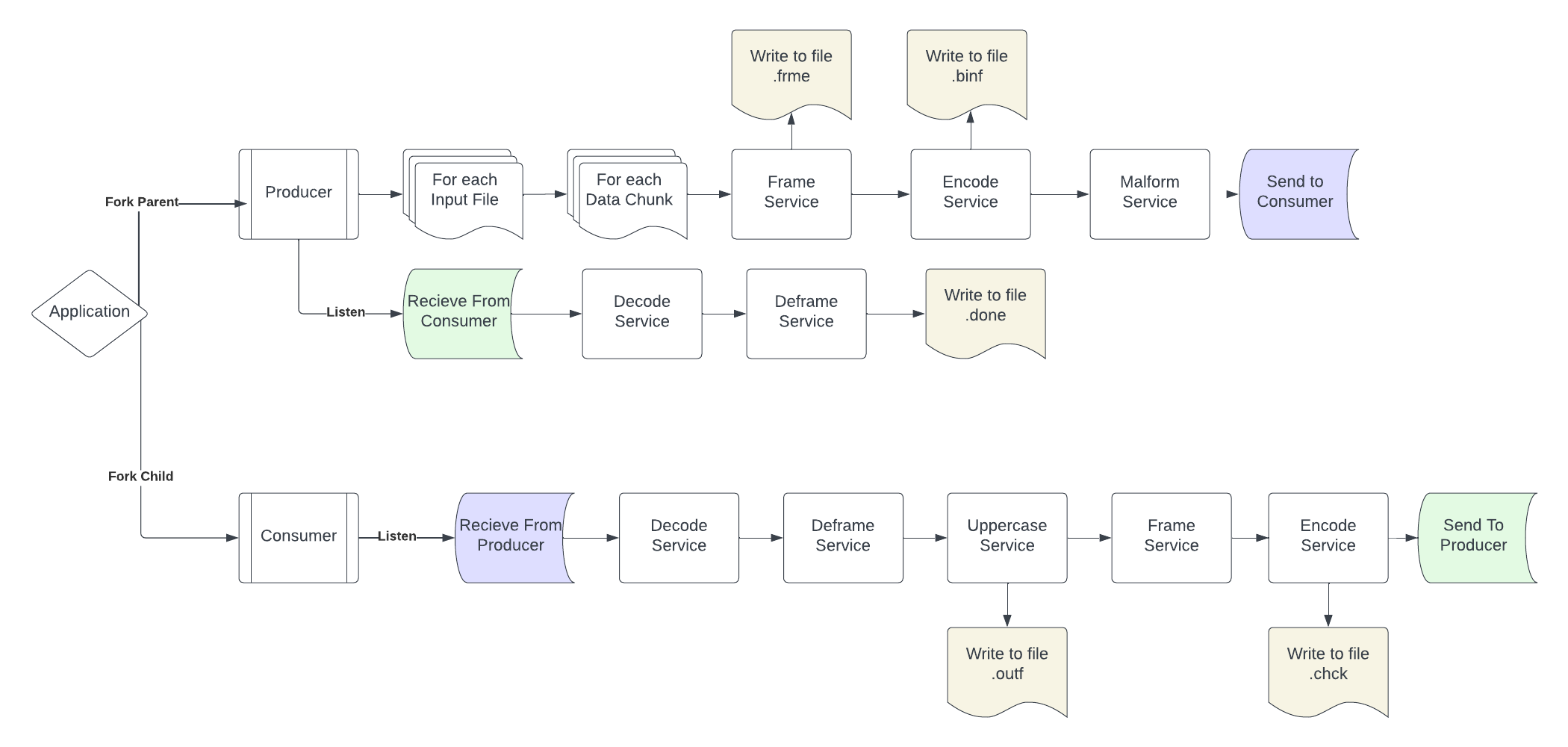
1. High Level Data Flow....................................................................................3
2. Routines & Descriptions...............................................................................4
3. Implementation Details..............................................................................4-6

Testing.....................................................................................................................6

User Documentation...............................................................................................7

**System Documentation**

High Level Data Flow



Data flows in the following manner:

1. The producer splits up each input file into chunks of 64 characters.
2. Each chunk gets passed through a pipe to a new process to be framed.
3. The result its written to .frme, and piped to a new process to be encoded.
4. This is written to .binf, and piped to a new process for a transmission error.
5. This malformed binary frame is written to the consumer through a pipe.
6. The consumer has been waiting on this frame. It pipes to decode it.
7. Then it pipes to deframe, and again to uppercase, which is wrriten to .outf.
8. It pipes and forks again to be reframed, encoded and written to .chck.
9. This is sent through the second main pipe to go back to the producer.
10. The producer pipes and forks to be decoded, and again to be deframed.
11. The result is written to .done and this chunk is complete.

Routines & Descriptions

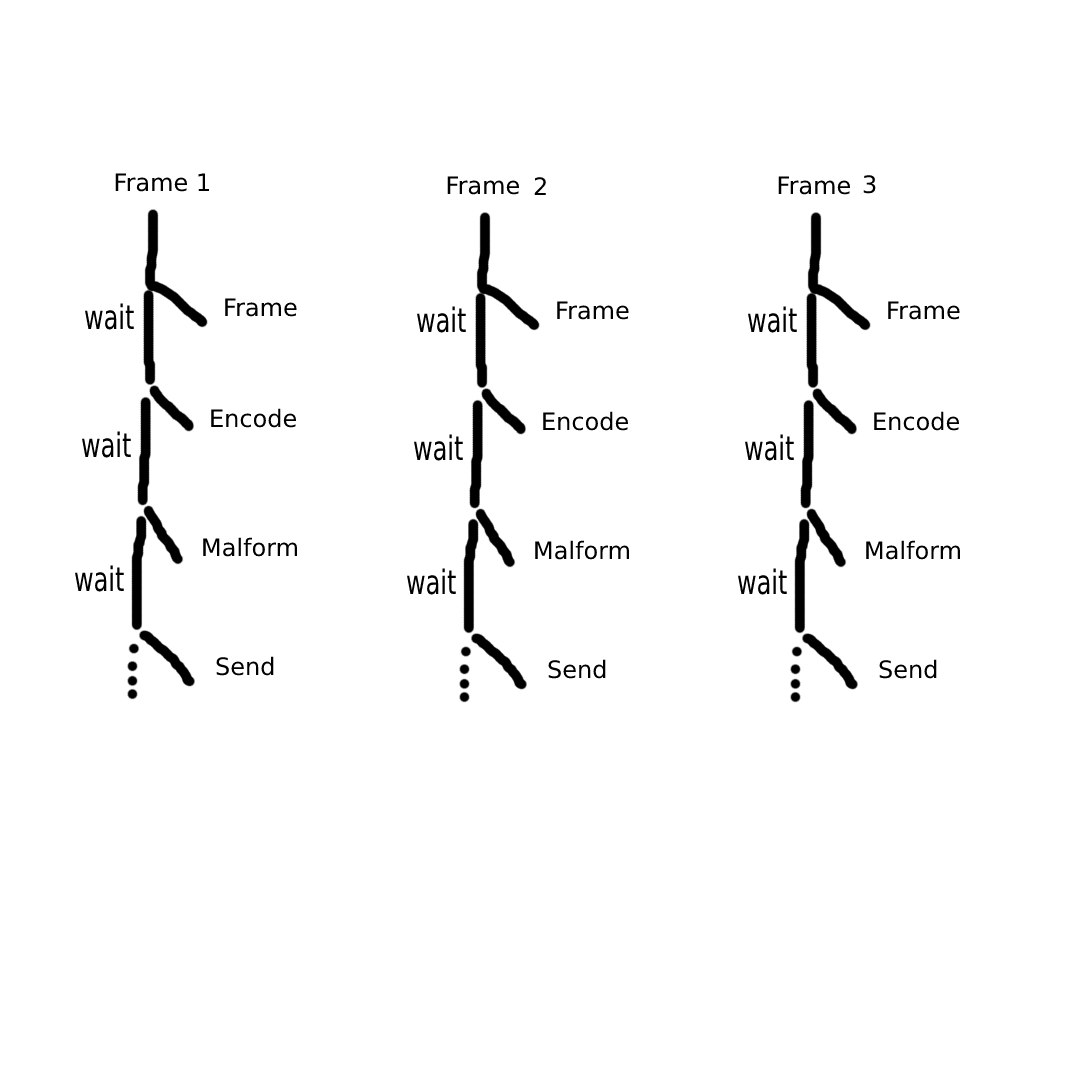
* **frameService**: This service reads an incoming chunk of data from a pipe and sends back a version which incudes control characters: Two SYN characters marking the beginning of the frame, followed by a character whose ascii code is the length of the data section which follows.
* **encodeService**: This service will take a framed chunk of data through a pipe and send back a version of length 8\*n where each character of the input is converted to its 8-bit representation, with the first being a parity bit.
* **malformService**: Operates on the incoming binary frame by choosing a random bit that is not a control bit or parity bit and then flips it for the purpose of simulating transmission errors. Sends the modified frame back.
* **decodeService**: Reads an encoded binary frame through its pipe and returns the characters that correspond to each set of 8 bits, removing the parity bit as well.
* **deframeService**: Takes a decoded frame and removes the control characters to yield the final resulting data as it was before it was transmitted through the service. Sends the result back through the pipe.
* **UppercaseService**: Returns through the pipe a version of the data chunk that replaces any lowercase characters with their uppercase counterparts.

**Implementation Details**

The services were implemented through a series of pipes and forks in a fashion that allows subroutines to execute independently of the main execution line, returning their data through a pipe before terminating. The application initiates with a fork of the main process, where the original process acts as the producer, and the new processes will execute as the consumer. When a service is desired, which first occurs when the program transforms the input file into frames, the program will create a pipe, and then fork and assign the child process the task of executing the service. In this fashion, the pipe can be accessed by both processes since it was created first, and then duplicated during the fork. This is necessary because it establishes a means of communication between two otherwise independent processes, where the input data is shared through the parent’s write end and accessed by the service’s read end. The data can then be sent back to the main process and read by the parent, because the child process will die and we want to transport the data to the spawning service by means of the pipe that way the data is not lost when the service is terminated.

The program’s use of streams of pipes and forks leads to an efficient and cohesive method of processing segments of data efficiently. The process described above of spawning a service will repeat within itself, because services depend on others. For example, encoding requires a framed set of data, so the encode process is nested within the frame process so that once the framed data leaves the pipe created for the frame service, the result data can immediately enter a new pipe to be transferred to a new service, which encodes the data.

It is important to note that although the program forks In a nested nature, it is the parent process that performs a fork. If the child process were to attempt a nested fork, its processes would die prematurely when the highest level services terminates because the owning process is killed at this point, which would unexpectedly kill the downstream subroutines. So there is a tree like structure of service forks, where the parent process continues downstream and branches off a new service each time we are at a new stage such as frame, encode, etc. There is one straight downstream path for each chunk of data read from input, and each branches off to a new process as in the following picture:



This allows for efficient pipelined execution, where each chunk of data can be processed right behind each other. Rather than waiting for every stage of preparing one frame before starting the next, when frame 1 enters stage 2, frame 2 enters stage 1 right behind, so as soon as there is computational space available, it is used to avoid any gaps or stalls.

**Testing**

The program was tested by using the supplied winVirus.inpf, and multiple intermediate files were generated through the program’s execution to observe the result of each service and ensure expected results.

1. After framing, I generated a .frme file first to ensure that I could visualize the proper location and existence of synchronization characters as well as the length control character.
2. After encoding, .binf is create to ensure that the correct number, value and placement of bits are present. Here we can expect a file of length 8\*n where n is the length of the input file, as each character should be converted into a 7bit ASCII representation, including a parity bit at the front that we can also verify is correct in this file.
   1. While testing, I also included a space between each 8 ‘bits’ to properly identify sequences of control characters and to visualize where frames began and ended.
3. After the consumer decodes, deframes, and capitalizes the data, I write the result to .outf to ensure that I have the exact input file, capitalized, with one character error (due to the transmission error simulation). I was able to visualize this file and repeatedly run the program to visualize in real time characters being malformed by the malform service.
4. After this result is passed back to the producer (by means of encoding, sending, and then decoding and deframing on the producer backend) the final result is written to .done, and we can visualize that it is the same as .outf if it was successfully transferred.
5. .chck file Is also generated after encoding the previous step, but before transmitting back to the producer. This allows us to see that the control characters are original, the bits for lowercase letters have been modified to capitalize the letter, and one single bit was flipped for means of simulation.

In addition to the test files, the status of file descriptors for pipes as well as service execution information were printed to the console to ensure proper timing of and data transfer between pipes and services with the main thread.

**User Documentation**

The program can be ran through the following procedure:

1. Navigate to /layer\_application
2. Run ./app

Place any input files in the input folder. All output files will be generated in a subfolder which is created for each input file that exists. You can have as many or as few input files.

*If you want to edit source code and recompile,*

1. Main app: gcc -o app app.c -lm
2. Service files: gcc -o <serviceName> <serviceName>.c

The program was developed on and for the linux operating system. If not done already, the GCC compiler must be installed as follows:

1. sudo apt update
2. sudo apt install gcc