Experimentation with Blockchain Enforced, Trust-less, Peer-to-Peer, Distributed File System

Josiah Nosek[1], Geoffry Huang[1] and Jonah Alexander[1]

[1] *City University of New York, Hunter College, Computer Science department, NY, USA*

Reception date of the project description: 05/22/2020

Abstract— With the rapid development of technology and increasing amount of data, there are huge chances for information loss and possible data tampering, which is a serious threat to the integrity and authenticity of the data records. Storing information in a central server may lead to problems in efficiency. There is a need for a distributed system that is both efficient and secure. Blockchain attempts to solve these issues by creating tamper-proof events of file manipulation in a distributed environment. Interplanetary File System (IPFS) is a protocol designed to store hypermedia in a peer-to-peer distributed file storage with content-addressability. The experimentation described in this paper attempts to combine both these technologies and other simplified encryption methods to create a secure model of file storage with access control methods. The system utilizes blockchain smart contracts to store the provenance metadata information written to a FAT table by the IPFS file system to the blockchain network to create a tamper-proof record of file creation and modification.

Keywords—Blockchain, Machine Learning, File systems, Networking

I. Introduction

The Purpose of Hunter College's Advanced Applications: A Capstone for Majors course is to challenge computer science students to explore complex system design and to push the boundary of their understanding. Students work closely in small groups to produce a multi-language systems implementation. Students are asked to look deeper than full stack development and try to gain understanding closer to the metal. By looking deeper into fundamental operating system processes, without the use of language libraries, students emerge with a more comprehensive understanding of the relationship between software and computer hardware. For this project we decided to explore implementing a distributed file system that utilizes custom file allocation tables (FAT) in which entries will be verified using blockchain technology.

II. Experiments and observations

Our first use case was to create a two node network, nodes "A" and "B", where "A" could pass a message to "B", and "B" could pass a message to "A." This use case was to provide the base case for a multi-server multi-client network. Essentially, this was to introduce us to socket programming, and challenge us to begin exploring concepts like a three way handshake – similar to the SYN, SYN/ACK, ACK handshake of the TCP/IP network protocol. At this point, we were also asked to begin thinking about how to prevent a buffer overflow occurrence, and to think about how to handle incomplete/dropped packets.

To begin understanding the implementation design, our professor had us look at a tutorial for implementing a socket chatroom server which walked us through socket programming in Python. We were able to set up a server, accept a connection from a client, receive a message and pass it to the clients. After we were able to get one client to pass a message to the server, we set up a range of ports for the server to listen to, and created a loop to cycle through them to see if any clients were connected and trying to send a message. Once we are able to achieve this, we set up a continuous loop to receive messages from all of our client sockets, then send all of the messages out to all of the client sockets. One implementation we considered adding here was to broadcast messages to all clients, omitting the source client. From here, our group had gained enough understanding to be able to transfer information as strings across network entities. We agreed that sending Python dictionaries, or other objects, between languages might be a difficult place to begin, so we decided to start by using plain text files and sending information between our programs as strings.

Our next use case was to expand our two node network to three nodes and begin striping data from a client on one node to the other two. At this point, however, we were unsure as to how to implement redundancy into the system, so we began experimenting with listing whole files on multiple computers. We began by expanding our network to three nodes, and we implemented an NTFS style client-server mapping. We created a .csv file to use as a FAT system to tell clients where they should look for a file on each network (see Table 1). To

Contact data: Josiah H. Nosek, Josiah.Nosek69@myhunter.cuny.edu

TABLE 1: NTFS SYTLE FILE MAPPING TO ENSURE REDUNDANCY

local_filename	remote_filename	server_addr	server_port	primary_server
file1	file1.txt	localhost	11001	yes
file2	file2.txt	localhost	11001	no
file3	file3.txt	localhost	11001	no
file4	file4.txt	localhost	11001	yes
file5	file5.txt	localhost	11001	no
file6	file6.txt	localhost	11001	no
file1	file1.txt	localhost	11002	no
file2	file2.txt	localhost	11002	yes
file3	file3.txt	localhost	11002	no
file4	file4.txt	localhost	11002	no
file5	file5.txt	localhost	11002	yes
file6	file6.txt	localhost	11002	no
file1	file1.txt	localhost	11003	no
file2	file2.txt	localhost	11003	no
file3	file3.txt	localhost	11003	yes
file4	file4.txt	localhost	11003	no
file5	file5.txt	localhost	11003	no
file6	file6.txt	localhost	11003	yes

explore distributing server load, we also assigned each file a primary server to which a client would look first, and then if the server was unreachable, the client would try the other two servers in order of port number. Our load balancing implementation was trivial in that it did not account for actual server load, nor file size. As new files were added, they were assigned by a file number%3 algorithm. For this implementation we were able to retrieve files from any of the servers, and the loss of a server would not interrupt the ability to retrieve files, however, we had not yet implemented striping files across the network.

After we successfully implemented a redundant, multi-server, multi-client, 3-node network file system, our group moved to tackling striping the file system across the servers instead of copying each file to each server. Our stretch goal for implementing this method was to allow variable sized files with fixed size blocks. The FAT table would record how many blocks, and to what servers they are assigned. We also hoped to introduce a variable number of servers. With this implementation the number of blocks would vary, as well as the number of servers they are striped across. To simplify the use case to provide us a stepping stone to the previously mentioned implementation, we assumed a fixed number of servers, and a file division of 4 blocks, no matter the size of the file. Thus, in this stage of the implementation each blocksize would be variable. This design also allowed us to continue to experiment with redundancy, and by writing a block and backup block to each server, we were able to maintain file retrieval with any 3 of the 4 servers online. The following represents the file mapping we used:

```
int file_pieces_mapping[4][4][2] = {
    {{1, 2}, {2, 3}, {3, 4}, {4, 1}},
    {{4, 1}, {1, 2}, {2, 3}, {3, 4}},
    {{3, 4}, {4, 1}, {1, 2}, {2, 3}},
    {{2, 3}, {3, 4}, {4, 1}, {1, 2}}};
```

The next use case we had to tackle was retrieval of a file by a client. Because we had a well structured, static file structure, this made understanding and implementing a method for retrieval more approachable. Additionally, it made retrieving a file in the condition where a server was down also easy to implement. If a given server was found to be unreachable, the client would simply try server n-1.

```
void fetch_remote_splits(int *conn_fds, int
    conn_count, file_split_struct *file_split, int
    mod)
{
  // Fetching file_split from remote servers with
    using the previous server in case current
    server is not available
  int i, socket, split_id, server;
  bool flag;
  u_char proceed_sig = (u_char)PROCEED_SIG;
  for (i = 0; i < conn_count; i++)
  {
    flag = false;
    socket = conn_fds[i];
    server = i + 1;
    if (socket == -1)
    {
      //If server is down, check move to n-1
      server = (i != 0) ? i : conn_count;
      socket = conn_fds[(i != 0) ? (i - 1) :
    conn_count - 1];
      flag = (i == 0) ? true : false;
      if (i != 0)
      {
        // Socket is expeciing a sig
        send_to_socket(socket, &proceed_sig,
    sizeof(u_char));
      }
    }
    //using the file_pieces_mapping to create id
    split_id = file_pieces_mapping[mod][i][0];
    /*fprintf(stderr, "Fetching split: %d from
    Server: %d\n", split_id, server);*/
    send_int_value_socket(socket, split_id);
    file_split->split_count++;
    if ((file_split->splits[split_id - 1] = (
    split_struct *)malloc(sizeof(split_struct)))
    == NULL)
    {
      DEBUGSS("Failed to malloc", strerror(errno))
    ;
    }
    write_split_from_socket_as_stream(socket,
    file_split->splits[split_id - 1]);
    if (flag)
      send_to_socket(socket, &proceed_sig, sizeof(
    u_char));
  }
  send_signal(conn_fds, conn_count, (u_char)
    RESET_SIG);
}
```

Once we created the basic functions of read and write files to the IPFS, we decided to implement a shell that a client user could interact with. For this use case we were able to implement "GET", "PUT", "MKDIR", and "LIST". One issue we had here was "MKDIR" needed all 4 servers online to function properly. We did not yet have a way to add a directory to a previously downed server once it came back online, however, we hoped to overcome that in the final implementations of the FAT system. For our next use case, we were also asked to consider file permissions, so, for this implementation we created users, passwords, and a trivial encryption algorithm. For now, each user had their own sub directory, and files could only be read/written within that users directory. Ownership and permissions were something else we were exploring in the FAT table design.

While we were still considering our next use case, we implemented our basic FAT table using the "file_piece_mapping" function. This basic implementation made writing the block stripe structure to the FAT much more simple (See function below). Using this implementation, we were also able to begin testing the blockchain validation and consensus (see appendix A) to integrate a trust-less design. To simplify this version of the FAT system, we chose to not yet build in redundant mapping of blocks, because we were still exploring block size, and our instructor asked us to consider deleting and, even more challenging, resizing a file with static block sizes.

```python
#Writes the fat to a file "fAT.txt"
def writeFat(target):
    #creates a file titled nFat.txt
    #w+ indicates write permissions and generation
      of a new file if not there
    f = open(target,"w+")
    #for i in range(20):
    for i in netFat:
        name = "(" + netFat[i][0] + "),"
        size = "({size}),".format(size = netFat[i
    ][1])
        b1,b2,b3,b4 = "(", "(", "(", "("
        temp = str(len(netFat[i][2]))
        b1 = b1 + temp
        temp = str(len(netFat[i][3]))
        b2 = b2 + temp
        temp = str(len(netFat[i][4]))
        b3 = b3 + temp
        temp = str(len(netFat[i][5]))
        b4 = b4 + temp
        for j in netFat[i][2]:
            temp = str(j)
            b1 = b1 + "," + temp
        for j in netFat[i][3]:
            temp = str(j)
            b2 = b2 + "," + temp
        for j in netFat[i][4]:
            temp = str(j)
            b3 = b3 + "," + temp
        for j in netFat[i][5]:
            temp = str(j)
            b4 = b4 + "," + temp
        b1 = b1 + "),"
        b2 = b2 + "),"
        b3 = b3 + "),"
        b4 = b4 + ")"
        #print(name,size,b1,b2,b3,b4)
        f.write(name + size + b1 + b2 + b3 + b4)
        f.write("\r")
    f.close()
```

When a file is added to the system, the file is broken and broadcast out. To track where the files go, we used the above file allocation table. It records the name of the file, the size of the file, the number of nodes containing each block, and a list of nodes for each block. If a specific file is requested, the table could be checked for the nodes for each given block. A copy of this file allocation table was stored locally on each client since the system is decentralized. We were also able to create a function to read in the FAT table as a text file and add it to a local blockchain. Any time a change to the FAT table is made the "add_block_from_file" function is called and the blockchain is updated.

One issue we identified in this use case is that while it is easier to keep a copy of the file allocation table on every client, it will likely cause problems as the number of files and node increases. The table will eventually reach a massive size and use significant space on all the clients. This could be avoided by using a distributed hash table and storing fragments of it across multiple machines. Though this would be much more efficient with a large-scale network with many files, there is minimal impact on a small scale one.

III. Demonstrations

a. Blockchain

We have provided a demo file to run in order to test a standalone local blockchain. The demo will create a local blockchain, add 3 transactions (or blocks) to it, attempt to tamper with one of the transactions and run the validate chain algorithm. The algorithm will spot the tampered block and print the location of the tampered block. By default, our consensus (appendix A) difficulty is 2. Our validation algorithm (appendix A) can be found in the "blockchain.py" file and is done in a function called "validate_chain."

The readFAT.py function will read in a FAT file called "fat.txt" and add it to a local blockchain. The "add_block_from_file" function is called anytime a change is made to the FAT file, thus the blockchain acts as a FAT file record keeper.

b. File System

The program for the FAT only runs when a change is made to the system. It starts by reading in the file allocation table already present in "fat.txt" before storing the data. After storing the current iteration of the FAT, it prints it to a secondary file titled "backup.txt" in case the original needs to be restored for some reason. A file titled "update.txt" is then read in and changes are made to the file allocation table based on the instructions stored in the text file. The "fat.txt" file then has the updated file allocation table printed in which completely replaces the original contents. After the new table is printed, the "update.txt" file is wiped to avoid any outdated instructions being performed again. The updated file allocation table is then added to the blockchain as a new transaction and is also broadcast to the other nodes in the system.

c. Networking

Documentation for running the Networking sub project can be found on the github project documentation page. There is a make file that takes the argument "make all" to spin up for servers on ports 11001-11004. From there a client can be run accepting the commands "MKDIR", "PUT", "GET", and "LIST." A debug console is present to show what transations are occuring, and what servers are being utilized. These actions are also recorded in the logs directory for each server.

IV. Difficulties and Lessons Learned

a. Blockchain

Blockchain is a relatively new and upcoming technology. As such, compared to other technologies, it is more on the theoretical side with new breakthroughs still occurring frequently. The downside to this is that resources for building your own blockchain are scarce. There is also a great deal of variation between implementations as blockchain is a very versatile technology. Every implementation offers a lot to learn ranging from what the transaction is to how consensus is achieved. In the case of our project, we had to find a way to save an entire FAT file in each block. We were able to do this by simply converting the FAT files into a string and passing the strings into the data portion of the blocks. This project proved challenging but helped in developing a fundamental understanding of blockchain.

b. File System

One troublesome part of this assignment was making the programs written in different languages function together. Receiving inputs from the networking portion written using C and processing it with a program written in Python was perplexing at first. We considered making a shell or some sort of interface to accept inputs, but that crashed more often than not. In the end it was easier to use text files to push inputs around instead. We could simply write whatever was necessary to an appropriately named file and have the programs in the other languages process it as needed.

c. Networking

There were many challenging aspects to implementing a network stack for this project. Handling buffers, memory allocation, and broken network connectivity were some of the more difficult objectives our group faced when working on this project. Converting ints into chars to be read in and out of a stream proved to be a particularly interesting implementation issue. Fortunately we found this post on when to use byte n 8 that did an amazing job explaining how to use bitwise operators to convert data types. Using that description, we were able to implement a version in C:

```c
void encode_int_to_uchar(u_char* buffer, int n)
{
  buffer[0] = (n >> 24) & 0xFF;
  buffer[1] = (n >> 16) & 0xFF;
  buffer[2] = (n >> 8) & 0xFF;
  buffer[3] = n & 0xFF;
}

void decode_int_from_uchar(u_char* buffer, int* n)
{
  int temp;

  temp = buffer[0] << 24;
  temp |= buffer[1] << 16;
  temp |= buffer[2] << 8;
  temp |= buffer[3];

  *n = temp;
}
```

V. Conclusions

From a FAT system standpoint, this implementation currently has issues with scale. While the file allocation table being stored on all the clients is currently not an issue with four clients and a limited number of files, a larger scale will result in large amounts of wasted space as the bloated table is stored locally on every machine. The solution for this could be addressed with a distributed hash table. Implementing a larger network would also require determining how many servers had to be included in a file write to maintain the level of redundancy we desired. As a whole, if we took this project further, tackling the problems with scale would provide a large number of unknowns that we would have to overcome, and would provide an enriching extension of this project.

Our group was able to explore and solve use case problems presented to us during the course. We gained a comprehensive understanding of the topics introduced in our operating systems and networking courses, and were exposed to programming unknowns that cause issues on a complex project such as this one. The project provided a challenging platform on which we could learn to work as a team with different technologies, which strengthened us as engineers and coders.

VI. Acknowledgments

We thank our advisor and instructor for the duration of this project, Soumik Dey.

We also acknowledge user BigBangData, whose Python implementation of a distributed file system outlined a lot of what we implemented in C for our networking stack. Their repo PythonDFS was invaluable.

A. Appendices

a. Definitions

Blockchain: The simplest way to describe a blockchain is a record of transactions that are saved in chronological order. The blocks in the chain store the transactions or data the user wants saved and a blockchain is simply a growing linked list of blocks. The linked list is connected by hashing each block and pointing to the hash of the previous block. There are many different implementations of blockchains, but the 3 main principles in a blockchain are some variation of proof of work (PoW), validation and consensus.

Validation: Blockchains are usually implemented in a network where each participant in the network will have their own copy of the blockchain. Depending on implementation, each participant will have to routinely validate their own and others' blockchains. They will run through the entire blockchain and check that none of the nodes have been tampered with. If the validation checks out, the participant's blockchain is valid. The validation algorithm will fail if an attacker changes any information in a block on the chain since any change to that block will change the hash. With a new hash, all future blocks will have will no longer be linked to the tampered block as next block points to the old hash. Unfortunately, the validation algorithm is not enough to create a secure blockchain since an attacker could tamper all future blocks and pass the validation algorithm; for this reason, we need a consensus.

Majority Consensus: A decentralized blockchain has many participants; each with their own copy of the blockchain. There are many ways to reach consensus, but one of the most common and straight forward ways is to have majority. An attacker can bypass the validation algorithm by altering all blocks following the initial block they intend on altering. This would cause an obvious change to the entire blockchain, but it would not be obvious unless there were other copies of the blockchain to compare to. In a majority consensus, all participants' blockchains will be compared to one another and the blockchain that 51% or more participants have in common is the one that reaches consensus. This makes the blockchain a lot more secure because it is a lot less plausible that an attacker could compromise 51% of a decent sized network. Even the majority consensus algorithm can vary from requiring a consensus before being added to the blockchain to checking each participants' chains afterwards. Part of our consensus implementation uses something called proof of work. The idea behind proof of work is that when a block is properly mined (or created) by a participant, the participant will broadcast to all the other participants of the network proof that they did the work, hence the name. The proof in our implementation is a value called a nonce that is used along with the data in each block to create the hash of each block. When passing the nonce and data combined into the hash function you should get a resulting hash with a set number of ending zeros. The number is equal to the difficulty you set for the proof of work.

b. Files

Github repo