Usage of Merkle Tree Hash in Data Version Synchronization to Cut Data Transmission Size

Aidil Rezjki Suljztan Syawaludin 13517070[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13517070@std.stei.itb.ac.id

Abstract—**Data has become essential to the world. Nowadays, almost everybody on Earth needs to have access to any form of digital data through various forms of devices in one way or another. The Internet is one such example, allowing people to get access to information from all around the world. Information is transmitted through the Internet in a form of data. To get information, the client needs to download the data through networks. Everything transmitted through the Internet is transmitted in a form of data. For example, when one downloads an application or file, it is downloaded in a form of data. Data makes up most of the digital world. As data gains more significance and importance, data transmission naturally also becomes a requirement. Data transmission through the Internet requires bandwidth and bandwidth is not an unlimited resource. In this paper, will be explored a mechanism which uses Merkle Tree Hash to hash the files or parts, to then decide which actual parts of the files, which will be called chunks, have changed, and download those parts only, to cut the data size to be transmitted or downloaded.**

***Keywords*—data, the Internet, transmitted, information**

I. Introduction

Data has become essential to the world. Nowadays, almost everybody on Earth needs to have access to any form of digital data through various forms of devices in one way or another. The Internet is one such example, allowing people to get access to information from all around the world. Information is transmitted through the Internet in a form of data. To get information, the client needs to download the data through networks. Everything transmitted through the Internet is transmitted in a form of data. For example, when one downloads an application or file, it is downloaded in a form of data.

As stated above, data makes up most of the digital world. As data gains more significance and importance, data transmission naturally also becomes a requirement. This comes at a cost. Data transmission through the Internet requires bandwidth and bandwidth is not an unlimited resource. The larger the data, the more bandwidth needed to transmit that data in a short amount of time. Though one can still transmit data with small bandwidth, it is not ideal because the time required to transmit the data will also get longer.

One example of data transmission requirement is when files or applications need to be synchronized with the latest one in the server. This is achieved by doing updates, downloading the required files or changes to local devices. As stated before, this transmission requires bandwidth. The basic mechanism to provide synchronization of files or applications is by redownloading everything. This is not ideal because some files or parts may not actually have changed, thus downloading everything would waste bandwidth. Another mechanism is by using hashes generated by the files digested by a hashing algorithm, then decide which files have changed and download. This however, may still waste bandwidth when the change in a file occurs in only a small part of the file, for example a few bytes on the ending part of the file.

In this paper, will be explored a mechanism which uses Merkle Tree Hash to hash the files or parts, to then decide which actual parts of the files, which will be called chunks, have changed, and download those parts only, to cut the data size to be transmitted or downloaded.

II. Data Transmission Through Networks

The Internet actually consists of a massive number of networks interconnected to each other, hence the name. Data transmitted through the Internet is actually transmitted over several networks. A network is defined by one or several connections between devices that allows the devices to communicate with each other in that network. This network consists of several layers, each having their own concerns regarding data transmission.

The first model of networking is the OSI model. In this reference model, a network is divided into seven layers with their own concerns[1]. The first layer is the physical layer. In this layer, the main concern is transmitting the raw bits of data through physical cables or channels that connect the devices in the network. This layer needs to make sure that if one end of the channel sends a bit 1, the other receiving end receives the same bit 1. This layer is more concerned with the physical attributes and characteristics of the medium used to transmit the data. For example, this layer needs to define voltages used to define bit 1 and bit 0 of data, the time needed to transmit that voltage, the pins to use, or wireless channels and other things that are more related to mechanical, electrical, or physical concerns.

The second layer in the OSI model is the data link layer. This data link layer is acting as an interface between the network layer and the physical layer. This layer handles how data received by the physical layer appear free of transmission errors to the network layer. In this layer, data is transmitted in a form of data frames. The sender is required to break the data into several data frames, which then is enveloped by frame header and frame footer. The receiver then receives those data frames and constructs the data back.

The next layer is the network layer. In this layer, routing of packets becomes a concern. A packet that is about to be sent needs to have a destination. The network layer decides where to send that packet. Thus, in this layer, routing is required. Routing is deciding where to send the packet. For example, a packet may be addressed to a device that is in the same network as the sender. In this case, the routing will decide to send the packet to that device directly. However, another case of sending a packet may involve a device that exists in a different network than the sender. In that case, the routing will decide to send that packet according to the policy set by the routing mechanism. One example is to send that packet to the gateway of the network. After sending the packet to the gateway, the network of the gateway device will then decide where to send the packet next. This is done through several networks until the packet is received by the correct device or dropped by the network.

The fourth layer is the transport layer. The main concern of this layer is to break data from the upper layers into smaller units which will then be sent by the network layer as packets and ensure that the data is sent and received properly. This layer isolates the upper layers from the possible changes of hardware technology of the lower layers. This layer determines the protocol of the data transmission, such as an error-free protocol that ensures the data is transmitted without error, or another protocol that focuses on the speed of the transmission without ensuring that the data is transmitted without error.

The fifth layer is the session layer. This layer focuses on establishing and maintaining sessions between users in different devices. This layer has dialog controls to keep track of whose turns to transmit through the network, mutual exclusion to avoid two parties entering critical operations at the same time, and synchronization to allow data transmission to continue without having to restart from the beginning.

The sixth layer, the presentation layer, mainly focuses on how to present the data. This layer makes sure that devices that have different data structures can still communicate through the network. This is done by defining an abstract data structure. Unlike the lower layers, this layer is not concerned about transmitting bits of data, but more about how the data can be understood by having semantics and syntaxes.

The last layer is the application layer. This layer is an application level layer that uses various protocols. One example is the HTTP (HyperText Transfer Protocol) which is the basis of World Wide Web applications. Other examples of the protocols in this layer is the FTP (File Transfer Protocol) which is used to transfer data between devices, or the IMAP, POP3, and SMTP which are used in email applications.

The OSI model may have defined the layers in detail. However, the currently used model for networking is actually not the OSI model. The widely used model of networking is the TCP/IP model. Similar to the OSI model, this model divides into four layers with separation of concerns.

The lowest layer is the host-to-network layer. This layer replaces the first and second layers of the OSI model. Thus, this layer has main concerns regarding data transmission on the physical level. The TCP/IP model does not define this layer in detail, leaving the actual implementation of this layer to the device manufacturers.

The second layer is the internetwork layer, which is the equivalent of the network layer in the OSI model. In this layer, a new protocol called the Internet Protocol (IP) is introduced. Similar to the network layer, this layer focuses on routing the packets to the correct destination based on an address defined in the packet. This address is known as an IP address. Devices connected to a network will be assigned an IP address, which will then be used by a router to determine destinations of packets. This layer allows connection between networks, thus allowing devices from different networks to communicate.

The third layer is the transport layer. This layer is similar to the transport layer in the OSI model. Two protocols are defined in this layer, in accordance with the two concerns of the transport layer in the OSI model. The first protocol is the TCP (Transmission Control Protocol), which focuses on establishing a reliable connection ensuring that no error occurs when transmitting the data. The second protocol is the UDP (User Datagram Protocol). This protocol focuses on lightweight data transmission allowing data to be transmitted in a short time, however not error-free. The TCP is suited for data transmission that requires error-free connection, such as file downloads or email transfer, while the UDP is suited for data streaming such as VOIP (Voice over Internet Protocol) or video streaming.

The last layer in the TCP/IP model is the application layer. The previous presentation and session layers were no longer needed, since those layers were rarely used when the OSI model was used. This layer is the same as the application layer in the OSI model. The usual protocols used in this layer are HTTP, FTP, SMTP, IMAP. POP3. Other protocols are added such as DNS for domain name resolution. Applications may also define their own protocol to use, hence this layer is flexible to the needs of the application.

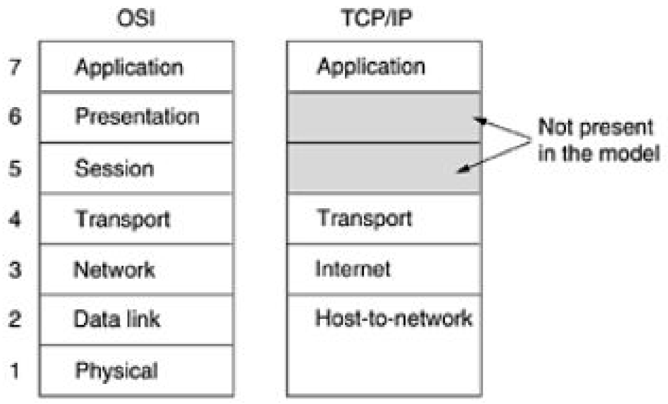

Figure 1. Correspondence of the OSI model to TCP/IP model. Source: Tanenbaum & Wetherall, 2014

III. Hashes and Merkle Tree Hash

A. Hash Function

Hash function is a function that generates a value with a predefined length by digesting a data input, which may be in a form of text, file, or other digital form of data[2]. This hash function is a one-way function. This means that this function generates a value based on the input, however the input cannot be determined or reconstructed from the generated value. Hashes are commonly used in order to determine whether data has been changed or not. This is due to the characteristic of hash functions, that is, hash functions produce a whole different value even when only a few bits or bytes of the original data is changed. Therefore, it is suited to be used for tamper-checking or error-checking.

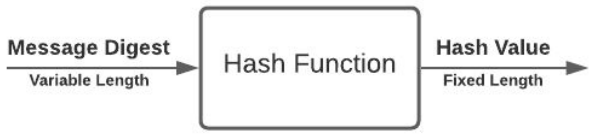

Figure 2. Flow diagram of a hash function.

As stated before, a hash function receives an input of data, which may be called as a message digest *m*. The function then generates a value of hash *n*. A hash function needs to make sure that for every possible *n*, there exists only one message digest *m* that results in *n*. This requirement is called collision resistance. A function that has multiple message digests m_x and m_y that results in the same value *n* cannot be used to do tamper-checking, because the data may actually have gotten changed even though the hash value is the same.

Generally, a hash function needs to have the following characteristics:

1. Accepts variable length input without any limits of input size.
2. Generates hash values with fixed length.
3. For any input of message digest *m*, the hash value *n* can be generated relatively easily by the hash function.
4. The hash function is a one-way function. The generated hash value *n* cannot be reconstructed back into the message digest *m*.
5. The hash function is collision resistant.

B. Merkle Tree Hash

Merkle tree hash is first introduced by Ralph Merkle. This concept is based on basic tree data structure and hashes[3]. Merkle tree is a binary tree, where the values on the leaves (nodes that have no child nodes) of the tree are the hash values of chunks of the data. The value of the other nodes in the Merkle tree is determined by generating hash values of the corresponding child nodes of the node. For example, given a tree with leaves of hash values H_1 and H_2, the root of that tree will have a value of H_3 which is the hash value generated by using the child values of H_1 and H_2 as the input for the hash function. To better understand, an example of a Merkle tree is provided below.

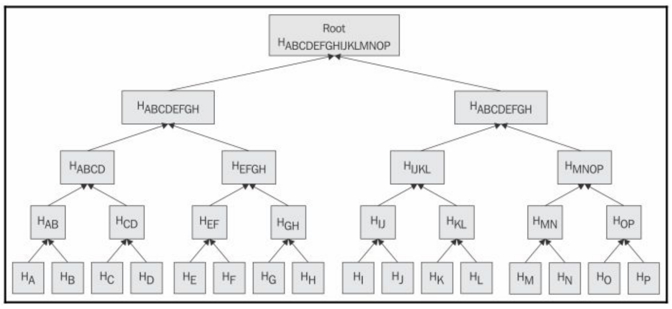

Figure 3. An example of a Merkle tree hash.
Source: Bashir, 2017

The value of the root in a Merkle tree will be determined by the values of the other nodes in the tree. This means that when at least one part of the tree changes, the value of the root will also change. Therefore, a Merkle tree can be used to determine whether a change has occurred on the data or not.

A change can be detected easily by comparing the values of the nodes in the Merkle trees. Merkle trees also allow for finding parts of the data that have changed. This can be done by traversing the tree being checked and comparing the values of the nodes to the original Merkle tree. When the value of a node is different, it may be caused by a change in the left child node or the right child node, which can be checked easily. This means that finding the changed parts can be done easily and relatively quickly.

To use this Merkle tree on a file, one can divide the file into several chunks of data. Then, the Merkle tree can be constructed by using the chunks as message digests to generate the hash values for the leaf nodes. After the leaves are constructed, the Merkle tree can be further constructed by using the method stated before. However, the number of the leaf nodes may not actually even. For example, the leaf nodes may only consist of 7 nodes. In this case, the algorithm can add a shadow node with a predetermined value, perhaps 0 or other values, which will then be included in the hash calculation for the parent child. This process will then be repeated for any nodes that do not have complete child nodes and are not one of the chunk nodes. Thus, Merkle trees can be constructed for any kind of files and digital data.

IV. Usage of Merkle Tree Hash in Data Version Synchronization to Cut Data Transmission Size

In this paper, an idea is proposed to use Merkle Tree Hash in synchronizing data versions to the hosted data version in order to cut the amount of data needed to be downloaded. The main idea can be divided into several parts as follows.

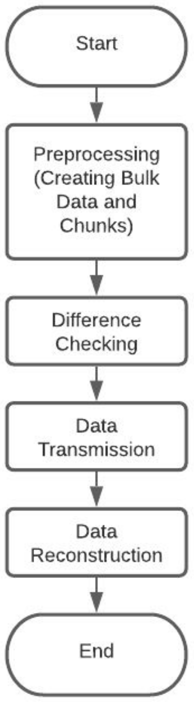

Figure 4. Flow chart of the proposed idea.

A. Merkle Tree Hash Construction

The first part of the idea is to construct the Merkle tree of the related data. This idea is focused mainly on handling a single bulk of data, thus to handle multiple files, a preprocessing may be needed though not mandatory since one can also traverse and operate on each file individually.

The preprocessing is done by creating a single bulk of data from the multiple files. The idea is to construct the bulk by appending the bytes of the related files. This can be done relatively easily by traversing each file then appending the bytes of the file into the singular bulk of data. The idea can be seen from the following figure.

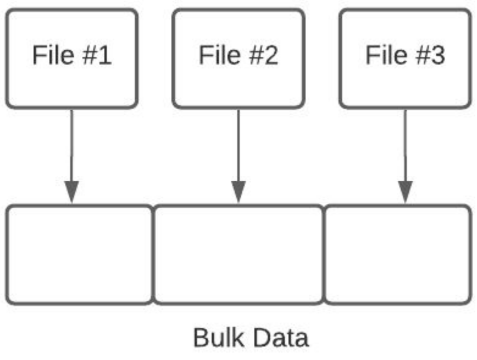

Figure 5. Construction of the bulk data.

After the preprocessing is done, the bulk of data will then be divided into small parts called a *chunk*. To do this, a fixed size of chunk needs to be determined first. The size may vary however needs to be consistent throughout the implementation. For example, if decided that a chunk will be 32KB, then both the client and server will later need to have the same chunk size.

The next step is to build the Merkle tree. The chunks will be used as message digest for the hash function to generate the hash values for the Merkle tree leaf nodes. After the leaf nodes are generated, the tree can be further constructed by using the Merkle tree method as stated before. Each hash of the chunks will be hashed in pairs to construct their respective parent nodes. This process will be done until the tree converges into one single node that will be the root node of the Merkle tree.

B. Version Difference Checking

The next part of the proposed idea is to check the difference of the versions in the client and server. This will be done by comparing the Merkle tree of the client and the server. Thus, the server also needs to do the first part, which is constructing their own Merkle tree.

After the first part is done by both the client and server (the server only needs to do this once when a change has occurred), the version difference checking can be started. The idea is to compare each node of the Merkle tree and determine which parts of the data are different. This will be done by traversing both the Merkle tree of the client and the server.

First, the client that wants to synchronize requests the Merkle tree of the server. After receiving the response, the client then traverses the node of the Merkle tree starting from the root. On each node, the client compares the value of the node with the local node. If the values are the same, then the traversal will be stopped. If the values are different, then the traversal continues. This process will be repeated until all the traversings are stopped or reaches the leaf nodes.

When the traversing reaches a leaf node, that means the chunk corresponding to the leaf node has changed and needs to be redownloaded. The idea is to keep track of the chunks that need to be redownloaded by using a simple data structure such as arrays that stores the index of the chunk.

There may be a case where the number of leaf nodes in the local Merkle tree is different from the Merkle tree of the server. This happens because the server changed the data and added new things into it. For example, when the server adds a new file or edits the existing file that results in enlarging the size of the file. When this happens, the client needs to also add those chunks to the list of chunks to download.

After all the traversings are done, the chunks needed have been determined and can be downloaded from the server in the next part. The use of the Merkle tree in this proposed idea allows for faster difference checking and determination of chunks required. Instead of traversing through all the chunks, in this idea one only needs to traverse the relevant nodes in the tree. Only when a difference is found will the child nodes be further traversed.

C. Data Transmission

The next part is to do the data transmission. In this part, the clients that have previously determined the chunks to redownload will proceed to request those chunks to the server. This means that the server needs to support downloads of the chunks based on the index of the chunk.

The server previously has constructed the bulk data and divided it into small chunks. The server then needs to serve those chunks and make it available to download by using indices of the chunks. For example, the client has determined before that the client needs chunks C_3, C_8, and C_9. The client will proceed to request those chunks to the server. The server needs to serve those chunks.

The method to serve those chunks are not determined directly in this paper. However, one can serve a web server that has an endpoint allowing to download the chunks by their indices. Another idea is to serve those chunks directly using existing web server programs. The client can download the chunks by using a predetermined pattern of address naming for the chunks.

After the chunks have been downloaded by the client, the client can then proceed to the next part.

D. Data Reconstruction

The last part of the proposed idea is to reconstruct the chunks downloaded from the server. The client has previously built the bulk data, divided it into small chunks, then built the Merkle tree, compared the Merkle tree with the one in the server, determined the chunks to download, and downloaded the chunks from the server. After that, those downloaded chunks need to be reconstructed back into the bulk data.

The idea is to put the chunks right into the existing bulk data. This means overwriting over the existing bulk data. For that, the client needs to know where to overwrite each chunk. An idea to solve this issue is by requesting that information to the server. The server needs to have metadata regarding the chunks it serves. That metadata will contain the required information for the client to write the chunks at the right location.

The method to provide the metadata is not directly ascertained in this paper, however there are several ideas regarding this issue. The first idea is for the server to serve an endpoint for the client to retrieve this metadata. The client will create a request to that endpoint, which then receives the metadata. That metadata will then be used to write the chunk into the correct place. Another idea is to bind the metadata on the data transmission. The server may add the metadata on the transmitted data, possibly by appending the metadata to the data about to be transmitted. The client then receives the data and parses the metadata first.

After the chunks have been correctly written on top of the existing bulk data, the next step is to reconstruct the files that were appended to construct the chunk data. This step is not necessary if the bulk data method is not used in the first part and uses each file in place of the bulk data instead. To reconstruct the files, the client needs to know how to divide the bulk data back into the files. Again, this paper does not directly decide the mechanism for the client to get this information. However, one possibility is by having the server serve this information.

Similar to before, in this solution the server serves the information about how to divide the bulk data back into the files. This can be done by serving an endpoint that returns the information. This information may come in the form of a list of filenames with the starting and end position of the byte (may use disposition from the start of the bulk data). The client will create a request to this endpoint and retrieve the information.

After the client knows how to divide the bulk data, it may proceed to reconstructing the files. The client will write the bytes of the bulk data with regard to the information before and recreate the files. The files will then be recreated and synchronized with the files in the server.

E. Data Transmission Size

The main purpose of this idea is to cut the size of data needed to be transmitted or downloaded when a client wants to synchronize files or data with the server. In this idea, the data that is transmitted are actually in a form of small chunks. The size of those chunks may be determined first to tune the performance.

By using this approach, when a client wants to synchronize the files, the client does not need to redownload the whole file or data from the server. Instead, the client only needs to download the small chunks that contain the changes from the server. This will naturally cut the size of data needed to be transmitted through the network, which saves bandwidth of both the client and the server.

For example, a client wants to synchronize multiple files with the size of 1GB each. A change has occurred in the server that only affected 512KB of data in two of the files. When using this proposed idea with a chunk size of 32KB, the clients only need to download the affected chunks which will not take more than 1024KB for the two changed files. The client does not need to download 2GB of data.

V. Downsides of The Proposed Idea

The proposed idea may cut the size of the data needed to be transmitted. However, there is a weakness of this idea found in this exploration. When the data that is about to be synchronized is a compressed data, the size of data required to be downloaded may actually turn out to be bigger compared to then the data is not compressed.

This may happen when the data is compressed, especially when compressed with a method that utilizes a dictionary to compress the data. This is due to the fact that the compression process changes the pattern of the data. When the uncompressed data is changed and then compressed, the compression process may change the bytes differently. For example, when using compression techniques with dictionaries, the resulting compressed data uses the patterns found to compress the data and the dictionary decompresses it back into the original form. This may end up changing more bytes than what should have been changed when the data is not compressed.

This issue may be solved by not using compression first, but using compression last when the chunks are about to be transmitted. The client then decompresses the chunks and proceeds to reconstruct the original files.

VI. Conclusion

The proposed idea provides a method to cut the size of data needed to be transmitted when synchronizing data from clients to the server. The usage of Merkle tree allows the client to compare the local version with the version on the server. In this proposed idea, the client is able to determine which small chunks of data are needed to be redownloaded by the comparison of the trees. Provided that preprocessing and reconstruction of the data needs to be done in order to gain the best result of the proposed idea. The preprocessing is done by creating a bulk of data by appending the files about to be synchronized and then dividing the bulk data into small chunks with a fixed size. The reconstruction is done by first overwriting the downloaded files on top of the existing local bulk data by using the metadata provided by the server and then dividing back the bulk data into separate files by using the information provided by the server.

VII. Acknowledgment

Alhamdulillahirabbil 'Alamin. First of all, the author would like to express praises and the most gratitude to Allah SWT. for the blessings and chance for the author to complete this paper. Furthermore, the author would like to thank the parents of the author, who have allowed the author to explore the interests of the author and to study and develop. The author would also like to thank Dr. Ir. Rinaldi Munir, M.T. as the lecturer of IF4020 Kriptografi for the knowledge shared and taught in the class despite the current ongoing COVID-19 pandemic which causes the lectures to be held online through various platforms. Finally, the author would like to thank the friends of the author who have helped the author in this exploration and the writing of this paper, also the authors of the references that have helped the author in the writing of this paper.

References

[1] Tanenbaum, A. S., & Wetherall, D. J. (2014). *Computer Networks*. Harlow, England: Pearson Education Limited.

[2] Munir, R. (2020). *Bahan Kuliah IF4020 Kriptografi*.

[3] Bashir, I. (2017). *Mastering Blockchain*. Birmingham, UK: Packt Publishing Ltd.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 21 Desember 2020

[illegible]

Aidil Rezjki Suljztan Syawaludin - 13517070