AN ABSTRACT OF THE THESIS OF

Hima Gotlur Muralidhar for the degree of Master of Science in Computer Science presented on September 17, 2019

Title: Enabling Graceful Software Upgrades in Distributed Systems

Abstract approved:

Jesse R. Walker

In today's world, we are highly dependent on software systems together with devices for almost every task in our day to day life. Software system upgrades are released whenever it is necessary to accommodate the ever-changing user's needs. The devices we use to run the software systems might be of different configurations, and we might be running different versions of these software systems on the devices. In such events, it is not possible for everybody to continue to run the same version of the software, nor can these billions of people using a software system update their software at the same time or even on the same day. Therefore, it has become necessary to design systems to maintain uninterrupted communication among the devices irrespective of software versions and device configurations. One of the key characteristics to achieve uninterrupted communication among the software systems is interoperability. In this thesis, we study techniques to achieve graceful software upgrades in distributed systems by implementing interoperability techniques. It is also observed that multiple software systems, like Bitcoin, failed to facilitate communication between versions.

This thesis identifies the necessary elements to support graceful upgrade, i.e., the principles that are recommended for an upgrade to avoid interruption of user's work or data. These elements were discovered by closely studying multiple widely popular distributed technologies that successfully achieved graceful upgrades. This thesis also analyzes the Bitcoin protocol and identifies the factors in its design that undermine the possibilities to have graceful upgrades in this cryptocurrency. A coding experiment was conducted to illustrate the efficacy of these principles and demonstrate graceful communications across three different versions of the same software.

Enabling Graceful Software Upgrades in Distributed Systems

by
Hima Gotlur Muralidhar

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented September 17, 2019
Commencement June 2020

ACKNOWLEDGMENTS

Foremost, I would like to express my deepest appreciation to my advisor, Dr. Jesse Walker for providing outstanding support for my research endeavors, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. As my teacher and mentor, Dr. Jesse Walker has taught me more than I could ever give him credit for here and I could not have imagined having a better advisor.

I am incredibly grateful to Dr. Rakesh Bobba, and Dr. Yeongjin Jang for their competent remarks, suggestions, and support. Their advice on my research has been invaluable.

I would like to extend my sincere thanks to Dr. Zhaoyan Fan for his insightful comments and encouragement.

Last, but not the least I owe all of this to my loving parents and grandparents, Mr. G.S. Muralidhar, Ms. Vinutha, Mr. Raghavendra Setty, and Ms. Rathnamma, and other family members, without whom my master’s journey would not be possible. Their unconditional love and support has made this journey a dream come true. I am also grateful to my friends who have supported me along the way.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

Page

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Motivation

There is a constant need for updates in any software, based on the needs of the users, and it is very important that software is upgraded responsibly. Though an upgrade process is one of the most important aspects to be followed, it is often overlooked by the software organizations and not everyone is aware of the design principles that must be conducted to support successful upgrades. These principles are rarely addressed in the engineering curriculum. A Software should be designed to allow upgrades without disrupting services users depend upon.

Upgrades become tricky in a distributed environment, where there might be millions or even billions of machines using a single software system at the same time, but it is impossible to upgrade the software on every system simultaneously; upgraded clients must still operate correctly with un-upgraded servers and vice versa. The configurations of these machines are mostly different from each other, and they are owned by different entities, who will each enforce select different times to perform upgrades. Even though these applications might be running on these machines on different versions, the users should be able to interact with each other, if needed. Since many developers assume a Flag-day, where everyone is required to stop the application and upgrade its software at the same time. But this is impossible to achieve because of the upgrade policies of the distributed ownership, and because of regulations in some industries, which does not allow for downtime.

Whenever a newer version of a software is released, a user's activity should not be interrupted, and it should be user-friendly; that is, the experience should be seamless for the user. As we shall see, Bitcoin is a distributed software system that was not designed keeping upgrades in mind, and a change to its block structure in March 2013 caused a hard fork: non-upgraded Bitcoin clients and miners would no longer accept bitcoins from upgraded clients and blocks published by upgraded miners [1]. This is not a good practice. We need to design our software in a way that each version can understand and communicate with each other successfully, this is to achieve interoperability.

The objective of this thesis is to investigate and propose design techniques to support graceful software upgrade. This is an engineering study more than computer science, as it explores techniques which have proved useful to solve the problem in real systems, and then demonstrates them in example code and an examination of Bitcoin. This study would be helpful to the software community to identify the techniques needed to make any version of the software user-friendly by making it interoperable.

1.2 Background

The aim of any version of the software is to carter the needs of the users for a long time. On the contrary, user needs are not constant over time, and this necessitates changes in the software. Therefore, we need a methodology to change a version of the software without disrupting the user experience or voiding resources users have accumulated by using prior versions.

Bitcoin is an example of a distributed system (DS). This study analyzes Bitcoin and identifies elements in its design that undermine the ability to upgrade its functionality. It also proposes techniques which would have overcome those shortcomings if they had been used by the Bitcoin designer. Few of the issues Bitcoins failed to address are what if the signature algorithm or hash function is broken? What if better hash or proof of work (PoW) algorithms are invented? What if the block size must be increased? What if new features must be implemented? People who designed Bitcoin might not have thought of these issues. A major constraint that needs to be achieved while implementing the necessary changes is that the system should not be broken, i.e., the system containing these new changes should be able to communicate with other versions successfully and know how to parse the fields that it doesn't recognize.

The solutions for these problems can be obtained by studying a few of existing distributed system protocols that did grapple with upgrade issues, such as migration from Bootstrap Protocol (BOOTP) to Dynamic Host Configuration Protocol (DHCP), working of X.509 and 802.11i. One of the main lessons learned from migration from BOOTP to DHCP is that you need to have reserve bits for future use [2]. X.509 points out the importance of extensions and how the signature algorithm in the certificates can be changed if the previous one is broken [3]. 802.11i illustrates the technique for a server to support clients implementing different versions of the protocol. These systems illustrate how to achieve interoperability in a distributed system and how the existence of features like version number, reserved bits, extensions, etc. makes a great impact in making the upgrade process successful.

1.3 Problem Overview

Bitcoin, a distributed system, failed to deliver a successful upgrade capability to its users. When a new version of Bitcoin was released after an increase in the block size of data structure, old clients would not recognize transactions published in the new larger blocks. That is, the older version of Bitcoin software was not able to communicate with the newer version. Due to this, the new software had to be abandoned in order to maintain a single community-wide view of the Bitcoin economy. A flag day is not possible, i.e., it is impossible that all the users around the world would halt Bitcoin trading simultaneously and update their software at the same time on the same day before resuming the Bitcoin economy. This failure illustrates that it is essential that every software version should be able to communicate with other versions of the same software irrespective of the version they are using.

1.4 Research Question

The primary research question of the work presented in this document is how to gracefully migrate from a prior version of software to a newer version, i.e., a software upgrade that does not break the environment or user data in a distributed system? The primary research question can be further broken down into the following sub-questions:

Sub-Question 1:	What are the factors in a system design that enable graceful upgrades?
Sub-Question 2:	What are the principles needed to achieve interoperability in a distributed system in the presence of software upgrades?
Sub-Question 3:	Can we illustrate these principles using real software?
Sub-Question 4:	Does Bitcoin fail to illustrate these principles? And how?

1.5 Thesis Outline

The first chapter starts with an introduction to this research, along with a brief background, finally laying out the problem, i.e., interoperability between different versions of software, this research aims to solve. The second chapter describes how a few existing protocols or software, like BOOTP – DHCP, X.509, and 802.11i, have been successful in achieving smooth communication between versions. Chapter two also provides a detailed working of Bitcoin. After understanding the functionality of the Bitcoin protocol, Chapter three has a detailed analysis of the problems hindering smooth or successful upgrades in Bitcoin. Furthermore, chapter three contains the demonstration of graceful upgrades achieved using all the necessary elements identified in this search so far. Finally, chapter four lists all the requirements that support graceful upgrades in distributed systems. Chapter 5 summarizes and concludes the research conducted in this study.

2 Literature Review

This chapter reviews multiple protocols in distributed systems that witnessed successful upgrades.

2.1 Upgrade from BOOTP to DHCP

2.1.1 BOOTP [2]

BOOTP is an Internet Protocol (IP)/ User Datagram Protocol (UDP) used by a server to assign IP addresses to the diskless devices in the network. A diskless workstation is a computer that requests the server on its network to load the required operating system. The bootstrap process can be categorized into two phases. In the first phase, the IP address of the client and the boot-file selection is determined. Here, the diskless workstations use BOOTP to obtain an IP address, along with other necessary information like the subnet mask, address of the router, address of the Domain Name Server (DNS) server, and the name of a Trivial File Transfer Protocol (TFTP) server. The second phase of the bootstrap process is where a file transfer occurs, and this phase is beyond the scope of the study presented in this document. The bootstrap protocol operates over UDP. The UDP port 67 is used for the server to receive the client's messages/requests, and UDP port 68 is used for the client to receive the server's responses.

The way the layers are stacked is: a BOOTP message is set as a payload inside a UDP datagram, UDP is a payload inside an IP datagram, IP is a payload inside an 802.2 frame, which in turn is inside an 802 (Ethernet or Wi-Fi) frame. When messages are exchanged between BOOTP client and BOOTP server, it has to go through all these different layers, i.e., each layer verifies the message and then passes it to the next layer. A receiving node processes as the following at each level of the protocol stack.

BOOTP performs a single packet exchange between a client and a server. Both client and server use the same packet layout with fixed-size fields to simplify the parsing. The client broadcasts a “BootRequest” containing its hardware address. The server then answers using a “BootReply.” The two packets are differentiated using an “op” field, which has two distinct values for each of these types. When a client doesn’t know its IP address, the server fetches the IP address of the client corresponding to the hardware address provided by the client in its BootRequest packet. A client can optionally provide the name of the server it wants to communicate with and receive replies from that server. In a given network if the server is not directly connected using a physical cable, the packet is forwarded to the neighboring gateways cooperating with each other. The packet is again forwarded based on the value in the “hops” field.

The BOOTP packet, along with client and server values to the corresponding fields, is given in the Table. 2.1.

Table 2.1: BOOTP Packet Format

BOOTP Header format	**Description**	**Length in bytes**	**Client Transmission (BOOTREQUEST)**	**Server Transmission (BOOTREPLY)**
Op	Operation code or opcode – indicates the message type 1 = BOOTREQUEST 2 = BOOTREPLY (0, 3-255 reserved)	1	Sets to 1	Sets to 2
htype	Specifies the Hardware Address type; Ethernet 10mb is the htype usually used, its value is 1.	1	Sets to 1	Sets the same as in BootRequest
hlen	Specifies Hardware Address Length of the type specified in the	1	Sets to 6	Sets the same as in BootRequest

	above field; value for 10 MB ethernet is 6			
Hops	Used for cross-gateway booting by gateways. It should be incremented on each forwarding.	1	Sets 0	If a gateway is used, fills this with the appropriate number to reach the intended server
Xid	A random 32-bit transaction ID; this is used to match the request and the response corresponding to it.	4	Generates a random number and sets it to that.	Uses the same ID while responding to the BOOTREQUEST
Secs	Used by the Client to indicate the time since it started trying to boot. Servers might use this to decide the order in which it wishes to respond	2	Sets to the number of seconds since it started to request for booting	Might use this to decide on the order of response to be given to the requests

Reserved bits/unused	Saved for future use	2	N/A	N/A
ciaddr	Client IP Address; if the client already has an IP address it fills this value with that address otherwise it sets to 0	4	0	N/A
yiaddr	"Your" IP Address; after allotting the IP address to the client, the server fills this field with the client's IP address	4	N/A	Sets to the Client's IP address if this was previously 0; this is to inform the client about its new IP address that it can use
siaddr	Server IP Address; this is the address of the server that is replying to the BOOTREQUEST	4	Sets to 255.255.255.255; since it is a broadcast message at the start as the client does not know who the server is	Sets its own IP address; may be used for further communication

giaddr	Gateway IP address; Used to indicate their IP address in cross-gateway booting by gateways; this is an optional field	4	N/A	The gateway address is used if the server and the client are not on the same subnet
chaddr	Indicates the client’s hardware address, i.e., the MAC address of the machine	16	Sets its own hardware address	Sets the same as in BootRequest
sname	Server Host Name; this is an optional field; the null-terminated name of the server the client wants to restrict the request to; it can be a nickname or a fully-	64	Might set it some desired server name if required	Fills in with current server’s address

	qualified DNS domain name of the server			
File	File Name of the required Boot file to be loaded into the client device by the server; it has to be a null-terminated name; it can null for default boot file; it can be a fully qualifies directory path; it can be a generic name like ‘Unix.’	128	Might specify either of the three formats as mentioned in the description	If the specified file name exists corresponding to the IP address of the client, then it replaces the ‘file’ with fully-specified pathname.
Vend	Vendor Extension. Optional. This contains the vendor information	64	Vendor related information can be present if required	Vendor related information can be present if required

Before setting up the packet for the first time, the entire buffer is cleared and is set to all zeros. The BOOTP client then sets the fields with the values as mentioned in Table 2.1 and broadcasts it in the network. As the client does not have an IP address yet, it cannot send unicast IP datagrams. The two most vital fields in a BootRequest message are the transaction id and the client's hardware address.

After receiving the client's request, the BOOTP server checks for the following conditions to decide whether to accept and process the request or discard it:

a) If UPD destination port doesn't match the BOOTP server port; then discard the packet.

b) If sname field is null or is specified, check for the match; if a match exists, continue with packet processing.

c) If sname does not match name or nickname, the packet is simply discarded. If the sname is present on the same physical network, discard the packet, as that server will pick this request and process it. If sname is not present on the same subnet, check for giaddr to see if the gateway is specified to forward the request on to a different network. If giaddr is zero, either fill it with the current server's address or a gateway that can be used to get to that network and then forward it.

d) If ciaddr is zero, then look for the client hardware address (chaddr, hlen, htype) in the database. If there is a match, fill yiaddr with the match found in the database. If no match is found, discard the packet or forward the packet on to other subnets.

e) If file is not null, lookup for a file name or fully-specified pathname along with client’s IP address, if a match is found, replace the 'file' field with the fully-specified pathname of the boot file.

f) If file is null, fill with the default Boot file

g) If the value in the file does not match any value in the database, then discard the packet.

After performing the above checks, the server populates the packet as displayed in Table 2.1. It sets the “op” to 2, “xid” to the same value to match the request, “siaddr” to its IP address, “yiaddr” to the client’s IP address, “giaddr” to the respective gateway address, “sname” to specify the DNS server, “siaddr” to the its (current server’s) IP address, and replaces “file” to the corresponding fully qualified directory-path.

When a client receives this BootReply packet, it does the following checks:

a) If Opcode is set to 2. If yes, the packet is processed. Otherwise, it is discarded.

b) If the xid matches the xid from BootRequest, if yes, process the packet. Discard the packet if xid doesn’t match.

c) As the client doesn’t know its IP address, it checks to see if the chaddr in BootReply match the client’s hardware address mentioned in its BootRequest. If it is not its hardware address, the packet is discarded. Otherwise, the packet is accepted.

After making these checks, if the packet is accepted, the client now has an IP address, which is the value in yiaddr, a server address (siaddr), a gateway address (giaddr) and a fully qualified path to the boot file.

When the request server is not on the same physical network, the packet should be forwarded to other networks using gateways. A gateway listening to the broadcast BootRequests takes its time to decide when it is appropriate to forward these packets. As a part of its configuration, a gateway might have a list of other networks to receive the broadcasted BootRequests. The gateway could wait until the "secs" or decide to forward the packet immediately. If the gateway decides to forward, it first checks the "giaddr" field. If the field is empty, it fills this field with its own IP address and then forwards the packet, so that other servers receiving this packet know which network or which server to contact to reach to that client. The "hops" fields can also be used along with giaddr to control how far the packet should be again forwarded.

2.1.2 Dynamic Host Configuration Protocol [4]

DHCP is an upgrade of BOOTP. DHCP is fundamentally interested in a dynamic IP stack configuration, while BOOTP is concerned about diskless workstation configuration, as discussed in the previous section 2.1.1. Despite this difference, BOOTP had the right infrastructure needed for DHCP to reuse BOOTP packet formats and infrastructure to achieve an entirely new function. For example, BOOTP gateway becomes the DHCP forwarding agent (discussed later in this chapter). That is, the BOOTP gateway infrastructure can be reused to forward messages between unconfigured nodes on one subnet and BOOTP/DHCP servers on another, thus allowing the TCP/IP stack configuration to be centralized at a site.

DHCP offers two services. First is a protocol where servers deliver host-specific configuration parameters to a host, and second is a mechanism to dynamically allocate network address or IP address to a host for a finite time. To provide the first service, DHCP needs to have persistent storage of its network clients. Each client entry is stored as a key-value. For each client, a key can be a single unique identifier, such as, transaction ID, hardware address or even a pair of identifiers, like, IP-subnet-number and hostname or IP-subnet-number and hardware address, within the subnet that make the combination unique and value is its configuration parameters, like, the lease during, etc. To provide the second service, DHCP can allocate temporary or permanent network addresses when requested by its clients. The period over which an IP address is allocated to a client is called "lease." Once the IP address is assigned to a client, it is not reallocated to any other client before the lease ends. These DHCP servers have a limited number of IP addresses to allocate among clients. Therefore, non-leased IP addresses can be reused when a new device joins the network. There might be cases where a client can ask for a permanent IP address; for examples, devices like printers, servers, routers can require unchanging IP addresses. In such cases, the DHCP server allocates a lengthy but not infinite lease, as a client can retire and not inform the server about it, statically using the BOOTP subset of the DHCP protocol. Figure 2.1 presents the DHCP message format.

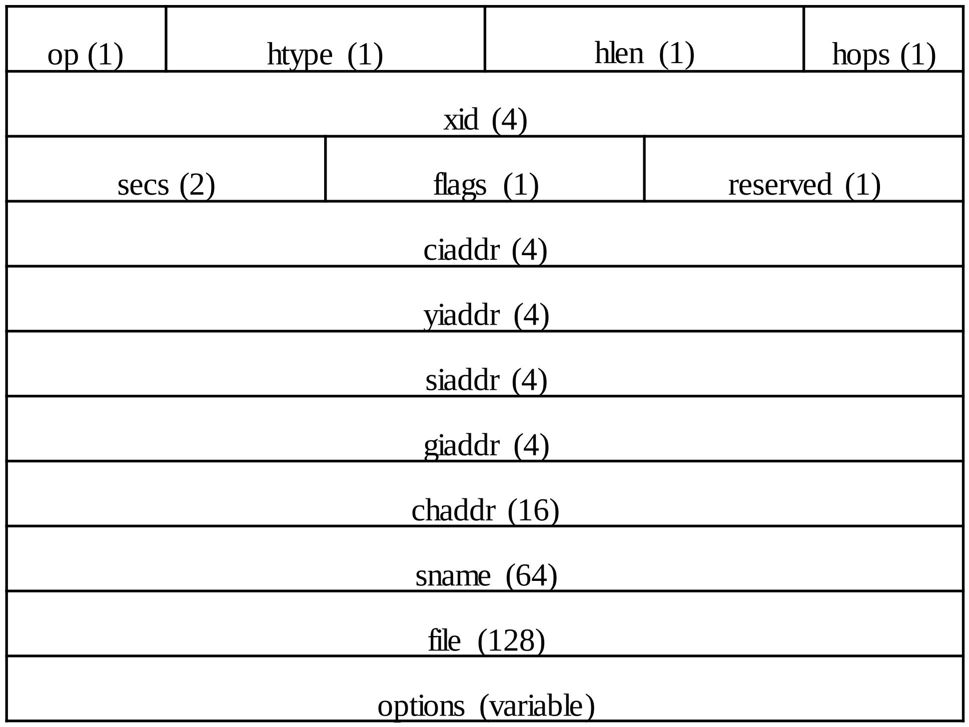

Figure 2.1: DHCP Message Format

The two main differences in the functionality of DHCP over BOOTP is that: First, DHCP assigns a network address to its clients for a finite lease. Second, a client can receive all the IP configuration parameters required to operate. A few modifications that are made to the existing BOOTP implementation to achieve DHCP are as follows:

a) Vendor Extensions (vend) field is renamed as Options. Multiple new option types are defined.

b) A new “client identifier” option was introduced. This “client identifier” option acts as a key and must be unique to that client within the subnet. Previously in BOOTP, chaddr was used for both hardware address and as a client identifier. This “client identifier” option must be unique to that client within the subnet.

c) The “siaddr” is now used to return the IP address of the next server to use in the next bootstrap service. To provide the IP address of the current server, a new “server identifier” option was introduced.

d) One of the reserved bits in BOOTP was used. As an alternative to the client who cannot receive unicast messages, a new field “flags” was introduced. The leftmost bit of this field is defined as the BROADCAST (B) flag. If this flag is set, it provides a hint to the server to broadcast any messages to the client.

DHCP is a two-phase commit protocol to make sure that clients and servers a synchronized view of the server’s dynamic database. So that the clients can depend on the state of the database they are aware of. This suggests that a client can believe that the IP address it is allocated to is not used by anybody else. Two-phase commit is required as UDP packets can be lost. In the two-phase protocol, the first phase is the request phase, where a client requests for an IP address, and the server responds with an available IP address. The second phase is called the commit phase. Upon receiving the IP address from the server, if the client agrees with the server’s decision, the client votes a “yes,” and the server commits the IP address into the database.

DHCP uses BOOTP messages to communicate with each other. A client initiates the process by requesting an IP address. The messages from a client to a server use BootRequest and the messages from a server to a client use BootResponse. Typically, four messages are exchanged between a client and a server to complete an agreement, two from each client and server. Since the opcode of the two messages from client or server is going to be the same, i.e., both messages from the client use BootRequest and both messages from the server use BootReply, a new option, “DHCP message type,” is defined which tells which of the four DHCP message it is. This option must be included in every DHCP message. Just like BOOTP, DHCP uses UDP ports for communication. It uses port number 68 for the client to send requests and port number 67 for the server to respond to those requests.

The interaction between the client and the server using the four message happens as follows:

1. “DHCPDISCOVER”: This message is broadcasted by the client on its local physical network request the servers for an IP address. The client must include the transaction ID (xid), its hardware address (chaddr) and DHCP message type option. The client may include the desired server, IP address, IP address lease time, and client identifier in its options. The DHCP forwarding agents may broadcast this request to the servers on other subnets.

2. “DHCPOFFER”: The servers willing to assign addresses sends a DHCPOFFER in response to the DHCPDISCOVER. The server must include the xid, available IP address in the “yiaddr” field, lease duration option, server identification option, and DHCP message type option. It may also include other configuration options. For the client on a different subnet, DHCP forwarding agents can transmit the DHCPOFFER messages. In this case, the message must contain the subnet mask, DNS server, gateway address, and the IP address of the next server to use in “siaddr” field. Since there can be multiple servers at a site, a client can receive multiple offers. The client picks the offer in which most of the desired configurations match. At this point, the offered IP address is not reserved for this client; it can still be allocated to other clients. Therefore,

clients must not start using the IP address offered in this message and wait till all the four messages are exchanged successfully.

3. "DHCPREQUEST": The client can broadcast this message to achieve any of the three purposes. First, to accept the offer from one server and inform other servers about it by implicitly declining their offers in response to the DHCPOFFER. Therefore, the DHCPREQUEST also acts as a notification to inform the other servers that the client did not select them. Second, to confirm the previously allocated address after a system reboot or any such situation. Third, to renew or extend the lease duration. A DHCPREQUEST must include the xid, the server identification options to imply which serve the client has selected, and the requested IP address option must contain the IP address (the network address in "yiaddr") offered in the DHCPOFFER. If the client chooses a server in other subnets, the DHCP forwarding agents broadcast DHCPREQUEST message to other subnets.

4. "DHCPACK": Servers use this message in response to confirm DHCPREQUEST. After receiving DHCPREQUEST, the server selected by the client checks if it can still offer the parameters it promised in the DHCPOFFER. If yes, the server commits the IP address to that client in its database and responds with a DHCPACK message. The combination of "chaddr" or "client identifier" option and the assigned IP address becomes the unique identifier for that client. Both client and server use this unique identifier to identify the lease. The DHCPACK must contain all the information (must not conflict with the parameters offered in DHCPOFFER) provided by the server previously in DHCPOFFER. After receiving the DHCPACK, the client

must do a final check to see if all the parameters are correct, and then the client is configured.

The client is responsible for retransmission of these messages. So, if the client does not receive either the DHCPOFFER or the DHCPACK from the server, the client should retransmit DHCPDISCOVER or DHCPREQUEST by setting the desired value in “secs” field until it receives a reply from the server.

Along with the above four messages, there are additional messages used in DHCP, which are discussed in the following paragraphs.

“DHCPNAK”: If the server has moved to another network or the network address has already been allocated, the server uses DHCPNAK to inform the client that the server is currently unavailable to satisfy the DHCPREQUEST message. After receiving a DHCPNAK message, the client restarts the configuration process.

“DHCPDECLINE”: After receiving the DHCPACK, if the client detects that the IP address it is allotted it is already in use, the client must use DHCPDECLINE message to inform the server about this and restart the configuration process.

“DHCPRELEASE”: If there is still time left for the expiration of the lease, but the client wants to end the lease, the DHCPRELEASE message is sent to the server by the client to inform that the client wishes to quit the current lease and the server can make this IP address available to other clients.

“DHCPINFORM”: When a client is already configured with an IP address, but it requires other local configuration parameters, the client can send a DHCPINFORM message. The server then replies with a DHCPACK message with the requested configuration parameters. However, the server should not assign a new IP address and should respond with a DHCPACK message.

2.2 X.509 [3]

X.509 is a standardized protocol used by public key infrastructure (PKI) to provide electronic identities. It is a standard determining the format for public key certificates (PKC). A certificate contains a public key and an identity, which can be a hostname, or an organization, or an individual, and is signed by either a certificate authority or can be self-signed. A self-signed certificate is a certificate that is signed by the same entity whose identity this certificate certifies. A PKC provides an authenticated data structure to communicate the public key used in asymmetric cryptography between entities in an open network. It binds the public key to an assigned identity identifying the entity that holds the corresponding private key. A PKC is considered valid and trustworthy when it is signed by a certificate authority (CA) that is accepted as valid and trustworthy, as the CA's signature on the certificate attests that the named entity holds the private key corresponding to the public key in the certificate. Thus, someone holding a certificate signed by a trusted CA, or validated by other means, can rely on the public key it contains to establish secure communications with another party or validate documents digitally signed by the corresponding private key. Some people believe self-signed certificates are trustworthy as well, but they assume that the certificate verifier is received directly from an already entity.

X.509 certificates are used for authentication. The authentication process is often started by an initiating entity when it sends its X.509 certificate to a responder. The X.509 certificate conveys the sender's public key and assigned identity, certified by a

CA. If the responder recognizes the CA, it believes the details in the certificate. Next, the responder generates and sends a random challenge. The initiator then signs a response that includes the random challenge sent by the requester, along with other protocol-specific information. Only the initiator who received random challenge will be able to produce a signature which can be verified as correct using the public key in the certificate. By doing this, the initiating entity proves its identity and can be now trusted as the party identified by the certificate. This, in turn, proves that the key received using this certificate is, in fact, the legitimate key. The protocol into which this exchange has been embedded can also be designed to guarantee that nothing was changed during the exchange of these messages.

2.2.1 The X.509 architecture

The X.509 architecture consists of the following entities:

Subjects and End Entities: A subject identifies the end entity whose public key is present in the certificate.

Issuer: An issuer is called a CA. Issuers can be categorized into root CAs and subordinate CAs. A root CA is a CA whose certificate is self-signed and well-known, i.e., end entities that trust it believe they have somehow securely obtained the root CA's self-signed certificate. Since a root CA is well-known, it is usually difficult for an attacker to forge a root certificate. A subordinate CA is a CA whose certificate is signed by another CA. One could have two subordinate CA's sign each other's' certificates, but this is not done; ultimately there has to be a chain of certificates from a root CA to a subordinate CA, with each certificate in the chain being signed by a subordinate CA above it, except for the root CA at the beginning of the chain, which is a self-signed certificate. Issuers create certificates for both subordinate issuers and end entities. An issuer that signs its own certificate is called a root certificate authority, and its certificate is well-known, i.e., publicly known by many other entities; since it is well-known, it is usually difficult for an attacker to forge a root certificate.

Certification Chains: The issuer of an end entity's certificate is usually a subordinate CA; the subordinate CA normally also has a certificate issued by another subordinate CA, and this process continues until the final subordinate CA has a certificate issued by a root CA. This sequence of certificates is called a certificate chain. As an example, an OSU student or faculty computer might be an end entity and be certified by a certificate issued by OSU's CA dedicated to issuing end entity certificates. This OSU CA would be certified by OSU's root certificate, which might itself own a certificate issued by a well-known public CA such as Verisign's CA for signing enterprise certificates; VeriSign's enterprise CA's certificate would be signed by VeriSign's root CA. Thus, the entire certificate chain in this example would be as shown in Fig. 2.2.

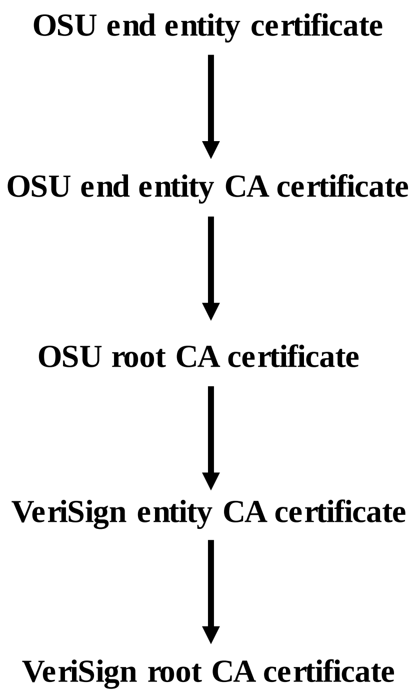
OSU end entity certificate
OSU end entity CA certificate
OSU root CA certificate
VeriSign entity CA certificate
VeriSign root CA certificate

Figure 2.2: An Example Illustrating a Certificate Chain

End entities usually only trust certificates issued by root CAs a priori, and so a certificate chain is a mechanism to recursively extend trust to entities unknown to the end entity relying on certificates: the end entity trust certificates issued by the root CA, so trusts certificates issued by the subordinate CA, etc.

Certificate Revocation List (CRL): It is a list of the serial numbers of digital certificates that have been revoked by CAs and should no longer be used. A certificate is revoked if the private key is believed to have been compromised. Revocation is irreversible.

Registration Authority (RA): A registration authority is an entity that receives and responds to certificate requests on behalf of an issuer. A certificate request asks for the RA's CA to issue a certificate to the requester. The X.509 architecture uses registration authorities for several purposes. First, requiring end entities to interact with RAs instead of CAs makes it somewhat harder to directly attack and subvert issuers. Second, RAs allow for more efficient operation of the public key infrastructure by batching requests to the CA. Finally, this architecture separates concerns, with the checking of the certificate requestor’s identity being performed in the RA and moving the mechanics of issuing certificates to the CA. It is believed this separation of concerns reduces the vulnerability of the entire PKI to attack.

Online Certificate Status Protocol (OCSP) [5]: This is a protocol used to access revocation status of X.509 certificates, to avoid the cost to end entities of fetching CRLs, which can grow quite large (large = megabytes or greater). It is a request-response protocol running over HTTP. An OCSP response can be either good, revoked, or unknown. If the server is unable to process the request, it sends an error message.

Lightweight Directory Access Protocol (LDAP) [6]: It is used to access a certificate repository: It is a protocol to enable accessing and maintenance of the distributed directory information services over an IP network. Using this any information like the public keys, email address, etc. can be obtained. It was derived from the X.500 DAP and simplified the X.500 standard. An LDAP directory is organized in a simple "tree" hierarchy consisting of the following levels: The root directory is the starting point or the source of the tree, which branches out to Countries, each of which branches out to Organizations, which branch out to Organizational units (like divisions, departments), which branches out to Individuals, which includes people, files and so on. Each entry is identified by a unique identifier called a distinguished name. This works in request-response format. Few of the requests that can be made on the items in the directory are:

a) StartTLS – use the LDAPv3 Transport Layer Security (TLS) extension for a secure connection

b) Bind – authenticate and specify LDAP protocol version

c) Search – search for and/or retrieve directory entries

d) Compare – test if a named entry contains a given attribute value

e) Add a new entry

f) Delete an entry

g) Modify an entry

h) Abandon – abort a previous request

i) Unbind – close the connection (not the inverse of Bind)

2.2.2 *Certificate Life-cycle*

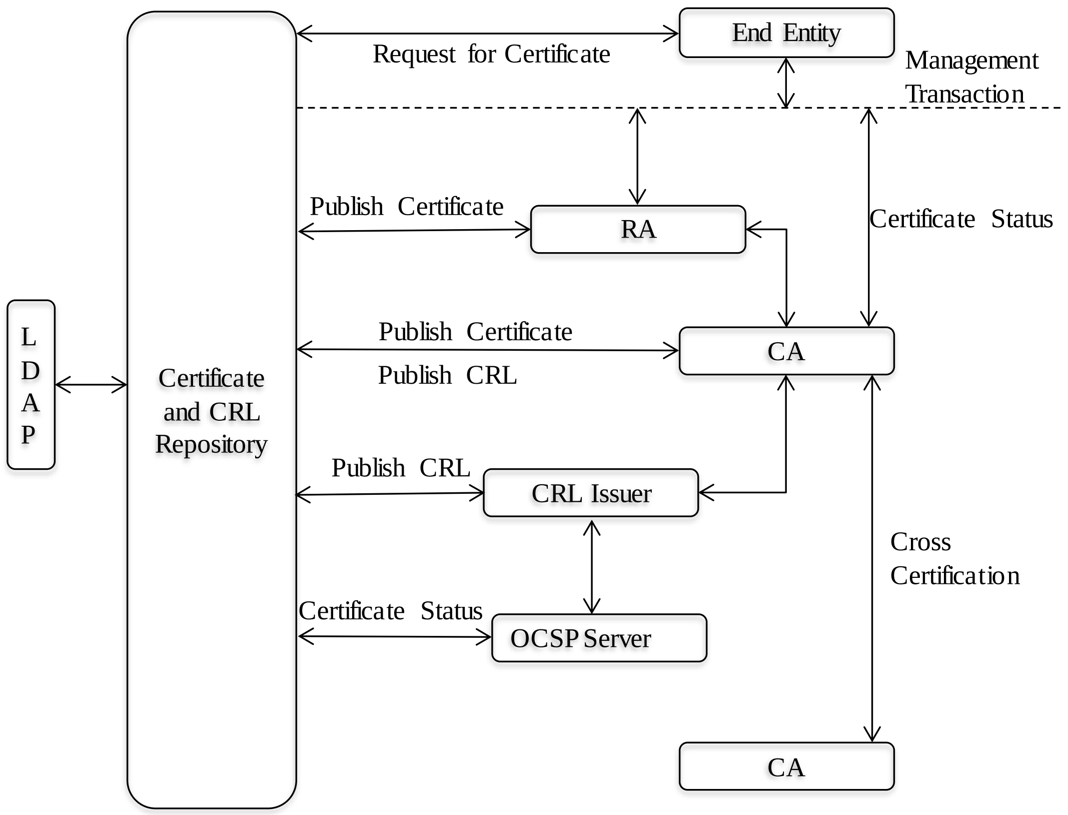

Figure 2.3: Certificate Life-cycle

Figure 2.3 depicts the interactions involving certificates, CRL, end entities, and the X.509 infrastructure. These interactions are explained below:

1. The end entity can directly communicate with the CRL server to request a certificate's status. The CRL determines whether the certificate has been revoked by checking with LDAP for each CA in its certificate chain. LDAP responds to the request by searching for that certificate in its directory. If the certificate is not revoked after checking with the CRL, the certificate is sent to the end entity.
2. If an RA receives the request, the RA may check whether the information provided in the request, and if so, ask its CA to issue the requested certificate. Finally, the RA responds to the end entity's request with a valid certificate or a denial of the request. An RA can also use LDAP to insert the newly issued certificate into the certificate directory.
3. If CA receives the request, the CA can insert certificates it issues into the LDAP directory itself, rather than relying on its RAs to accomplish this function for it. It is more common for an RA to insert the certificates issued by a CA into the certificate directory than the CA.
4. Cross Certification: In this process, a certificate is issued by one CA to another CA. The details in one certificate of one CA can then be trusted by a user who trusts the certificate of the other CA. Two CAs establish and specify the terms of trust and issue certificates to each other, entities within the separate certificates can now interact subject to the policies specified in the certificates. End entities can't know about every possible root CA, so can't necessarily trust certificates issued by these unknown CA. The way X.509 attempts to alleviate this problem is through cross-certification. If the

end entity trusts certificates from, e.g., Symantec, but doesn't know about GoDaddy, then it can trust certificate chains rooted in GoDaddy if Symantec cross certifies the GoDaddy root CA. In doing so, Symantec will issue a certificate with the GoDaddy root CA's certificate embedded in it, in effect saying, "Symantec hereby certifies that GoDaddy certificate chains are also trustworthy." This extends the end entity's trust in Symantec certificate chains to GoDaddy certificate chains by essentially making GoDaddy's root CA a sub-CA to Symantec's. So, after a CAs verifies with the other CA, the steps in c) would be followed.

2.2.3 Management protocols

The following management protocols help in establishing the online connection between the end entities and the management entities (Few management entities are: CAs, RAs, LDAP, etc.).

a) Initialization: In this phase, all the necessary materials needed to install keys are initialized to carry out the further processes securely. This might include installing or generating the public keys on the end entity's side.

b) Registration: To register, an end entity contacts the RA function and gives it a certificate request. The RA may acquire other information from the end entity, and once it has collected "evidence" satisfying its policy, it forwards the request to the CA, which issues a certificate. The CA returns the certificate to the RA and then stores the certificate into some repository. Then, the RA delivers the certificate to the end entity. This process binds an end entity name (Subject or SubjectAltName) to the end entity's public key.

c) Key pair recovery: In order to recover from hard crashes, where an end entity is destroyed, the end entity might store the key materials like the encrypted private key of the user, into a backup system. In case, the user loses the keys, it can be recovered using this protocol.

d) Key pair update: All key pairs need to be updated from time to time. The old key pairs will be replaced with a new key pair, and new certificates will be issued. This is accomplished by requiring the end entity to randomly generate a new key pair and registering it with an RA, with the request signed by both the old and the new key.

e) Revocation request: If evidence exists that a key might be compromised, this function is used by the CA to revoke that certificate.

2.2.4 Abstract Syntax Notation One (ASN.1) [7]

ASN.1 is a standardized data description language for defining machine-independent data structures. It also supports serialization and deserialization of these abstract data types in a cross-platform way. That is, an ASN.1 compiler can compile modules into libraries of code that decode or encode the data structures. It is mostly used in telecommunications and computer networking, and especially in application protocols. ASN.1 is both human-readable and machine-readable. X.509 certificates are written in the ASN.1 language. ASN.1 uses a type-length-value format. Types and values can be assigned names using the assignment operator (::=). The four types that ASN.1 supports are:

1. Simple types: These are atomic and have no components. Few of the simple types used in X.509 certificates are:

a) BIT STRING, an arbitrary string of bits (ones and zeroes).

b) INTEGER, an arbitrary integer.

c) OBJECT IDENTIFIER, an object identifier, which is a sequence of integer components that identify an object such as an algorithm or attribute type. OBJECT IDENTIFIERs play the role of tags in ASN.1.

d) OCTET STRING, an arbitrary string of octets (eight-bit values).

e) UTCTime, a "coordinated universal time" or Greenwich Mean Time (GMT) value.

2. Structured types: These consist of components. Structured types can have optional components, possibly with default values. Few of the structured types are:

a) SEQUENCE, an ordered collection of one or more types. Only this is used in X.509 certificates.

b) SEQUENCE OF, an ordered collection of zero or more occurrences of a given type.

c) SET, an unordered collection of one or more types. Note that a SET can include multiple objects of the same type.

d) SET OF, an unordered collection of zero or more occurrences of a given type.

3. Implicitly and explicitly tagged types: Tagging helps in distinguishing types within an application, or even within a structured type. Tags help in avoiding ambiguity. There are two ways to tag a type:

a) Implicitly tagged types: These tags are derived from other types and are obtained by changing the tag of the underlying type. These tags are identified by the keyword “IMPLICIT” present after the tag number.

b) Explicitly tagged types: These tags are derived from other types by adding an outer tag to the underlying type. These tags are identified by the keyword "EXPLICIT" present after the tag number. If only the tag number is present, it is considered as an explicit tag type.

4. *Other types*: These tags include

a) CHOICE: This type denotes a union of one or more alternatives from which one type should be chosen.

b) ANY: This type denotes an arbitrary value of an arbitrary type, where the arbitrary type is possibly defined in the registration of an object identifier or integer value.

2.2.5 X.509v3 Request Format [3]

An X.509 v3 certificate request has the following formats:

```
CertRequest         ::=             SEQUENCE {
                certReqId          INTEGER          --ID for matching
                                                    request and reply
                certTemplate       CertTemplate
                controls           Controls
                                   OPTIONAL}


CertTemplate        ::=             SEQUENCE {
              version          [0] Version                OPTIONAL
              serialNumber     [1] INTEGER                OPTIONAL
              signingAlg       [2] AlgorithmIdentifier    OPTIONAL
              issuer           [3] Name                   OPTIONAL
              validity         [4] OptionalValidity       OPTIONAL
              subject          [5] Name                   OPTIONAL
publicKey       [6] SubjectPublicKeyInfo     OPTIONAL
issuer ID       [7] UniqueIdentifier         OPTIONAL
subjectUID      [8] UniqueIdentifier         OPTIONAL
extensions      [9] Extensions}              OPTIONAL
```

```
Controls    ::=    SEQUENCE SIZE (1..MAX) OF AttributeTypeAndValues
```

Next, a CertReqMessage is created, containing CertRequest. The client formats and sends a CertReq to an RA. The RA eventually sends a response, either giving the client the certificate it has requested or a denial of the request. This request message contains CertReqId field which contains a random number that is sent to the CA.

```
CertReqMessages        ::=            SEQUENCE SIZE (1..MAX) OF
                                                CertRequest
```

After receiving this request, CA checks if the certificate can be granted. Whether this is the case is outside the scope of the X.509 specification, as different CAs will enforce different policies about how the requests are authenticated. The request is discarded by the CA if the requirements do not match. If CA agrees to grant the certificate, it signs the certificate along with the random number sent by the end entity. By doing this, two things are proved:

A signature over the random number proves that certificate was generated after receiving the request and not before. Random number proves the liveness of the certificate.

The identity of the CA can be proved because only the legitimate CA can know the private key used to sign the certificate.

2.2.6 X.509 v3 Certificate Format [3]

The X.509 v3 certificate basic syntax, which is represented in the ASN.1 structure. A certificate is basically a header and is a sequence of three required fields, as shown below:

```
Certificate          ::=                     SEQUENCE {
                     tbsCertificate          TBSCertificate
                     signatureAlgorithm      AlgorithmIdentifier
                     signatureValue          BIT STRING}
```

tbsCertificate: Contains all the required information about the certificate. It contains the names of the subject, issuer, a public key, the validity period of the certificate, etc.

signatureAlgorithm: Contains the identifier for the cryptographic algorithm used by the CA to sign this certificate. The algorithm identifier is used to identify a cryptographic algorithm, and its structure is as follows:

```
AlgorithmIdentifier  ::=            SEQUENCE {
                     Algorithm      OBJECT IDENTIFIER
                     Parameters     ANY DEFINED BY
                                    algorithm OPTIONAL}
```

The OBJECT IDENTIFIER component identifies the algorithm (such as DSA with SHA-1). The contents of the optional parameters field depend on the algorithm used.

signatureValue: Contains the digital signature computed over the ASN.1 encoded tbsCertificate. This value is encoded as BIT STRING. By computing the signature, CA validates the information present in the tbsCertificate and certifies the binding between the subject and the public key mentioned in the certificate.

The following ASN.1 structure presents the components that store all the necessary information of a certificate

```
TBSCertificate        ::=        SEQUENCE {
                 Version             [0] EXPLICIT   Version DEFAULT v1
                 serialNumber                        CertificateSerialNumber
                 signature                           AlgorithmIdentifier
                 Issuer                              Name
                 Validity                            Validity
                 Subject                             Name
                 subjectPublicKeyInfo                SubjectPublicKeyInfo
                 issuerUniqueID      [1] IMPLICIT   UniqueIdentifier
                                                     OPTIONAL
                                                     -- If present, version
                                                     MUST be v2 or v3
                 subjectUniqueID     [2] IMPLICIT   UniqueIdentifier
                                                     OPTIONAL
                                                     -- If present, version
                                                     MUST be v2 or v3
                 extensions          [3] EXPLICIT   Extensions OPTIONAL
                                                     -- If present, version
                                                     MUST be v3
                 }
```

Version: This field is to identify the version of the X.509 certificate. Currently, version 3 certificates are used, as it supports extensions. The value for version 3 is 2. If the field - “UniqueIdentifier” is present, then the version of the certificate is 2, and the value of this version is 1. If only basic fields are needed in the certificate, version 1 is used, and the value of this is 0.

```
Version        ::=        INTEGER        {v1(0), v2(1), v3(2)}
```

Serial number: It is a positive long integer, up to 20 octets, assigned by the CA. It must be unique for each certificate issued by a given CA, i.e., the issuer name and serial number identify a unique certificate.

```
CertificateSerialNumber        ::=        INTEGER
```

Signature: This field contains the algorithm used by the CA to sign the certificate. It is the same as the field “signature algorithm” mentioned above.

Issuer: This field contains the information of the CA or the RA who issued the certificate. The type of this field is Name, and this field should contain a non-empty distinguished name. The attributes of the name are composed hierarchically. The structure of Name defined in ASN.1 is as follows:

```
Name                         ::=             CHOICE {
                             rdnSequence     RDNSequence}

RDNSequence                  ::=   SEQUENCE OF RelativeDistinguishedName
RelativeDistinguishedName    ::=         SET SIZE (1..MAX) OF
                                         AttributeTypeAndValue
```

The value for Name is selected from the list of the RelativeDistinguishedName mentioned in the RDNSequence. Only one value is chosen from the sequence.

```
AttributeTypeAndValue        ::=             SEQUENCE {
                             Type            AttributeType
                             Value           AttributeValue}

AttributeType                ::=             OBJECT IDENTIFIER
```

```
AttributeValue      ::=      ANY -- DEFINED BY AttributeType
```

The type of AttributeValue is decided by the value chosen for the field AttributeType.

```
DirectoryString          ::=                   CHOICE {
                    teletexString        TeletexString (SIZE (1..MAX))
                    printableString      PrintableString (SIZE (1..MAX))
                    universalString      UniversalString (SIZE (1..MAX)),
                    utf8String           UTF8String (SIZE (1...MAX)),
                    bmpString            BMPString (SIZE (1..MAX)) }
```

All the above decisions are based on the value present in DirectoryString, which recommends the type of value Name can have. The values in DirectoryString can be a country, organization, state, common name, surname, given name, etc.

Validity: This field specifies the validity period of the certificate granted by the CA, who signed the certificate. This field as a sequence of two dates which specifies the beginning of the certificate validity period and the time after which the certificate is not considered valid. The validity period should be represented in UTC or generalized time. Before the validity period begins and after it ends the certificate is considered invalid, and any information in it is untrusted.

```
Validity            ::=             SEQUENCE {
                  notBefore             Time
                  notAfter              Time}


Time                ::=             CHOICE {
                  utcTime               UTCTime
```

```
                                generalTime                GeneralizedTime}
```

Subject: This field identifies the entity whose public key the certificate contains or the entity bound to the public key. The subject can be either present here or in the extension – "subjectAltName." A certificate can be self-signed; in this case, the issuer and the subject are the same (the certificate of a root CA is always self-signed). Therefore, the subject field should contain a non-empty distinguished name that matches the contents of the field Issuer and should be encoded in the same manner as present in the Issuer field. If it is not a self-signed certificate, then this field should a non-empty distinguished name which would not match the Issuer field. For all the other name formats, the extension – "SubjectAltName" is used. If there is no value present in the subject field, SubjectAltName is made critical, which makes it mandatory to have either of the field populated at all times. The issuing CA assigns the SubjectAltName for the end entity that generated the certificate request. The CA usually constructs this by prepending some form of a name requested by the end entity to a string in a namespace controlled by the issuer.

Subject Public Key Info: This field contains the subject's public key used in the certificate and the algorithm used to derive the key. The AlgorithmIdentifier used here is the same used in signatureAlgorithm.

```
SubjectPublicKeyInfo        ::=                 SEQUENCE {
                          Algorithm             AlgorithmIdentifier
                          subjectPublicKey      BIT STRING}
```

The subjectPublicKey BIT STRING is parsed according to the algorithm specified.

Unique Identifier: This field contains the subject's and issuer's unique identifier. Because the names of the subject or the issuer might get reused later, this field should be present only in version 2 and 3 and not in 1.

Extensions: This field provides a way of adding new information other than the existing fields. These are present only in version 3 certificates.

```
Extensions    ::=        SEQUENCE SIZE (1..MAX) OF Extension
    Extension            ::=            SEQUENCE {
                         extnID         OBJECT IDENTIFIER
                         Critical       BOOLEAN DEFAULT
                                        FALSE
                         extnValue      OCTET STRING}
```

One of the important extensions is the Subject Alternative Name. This field is used as an extension of the "subject" field. The name form options this field should accept are Internet electronic mail address, a DNS name, an IP address, and a Uniform Resource Identifier (URI). If an alternative name form is used to identify the subject, the Subject Alternative Name should be populated, and the subject field should be empty. If the subject field is empty, the CA marks the Subject Alternative Name as critical and fills the subject identity that is bound with the keys.

2.2.7 Comparison of X.509 v2 and X.509 v3 [3], [8]

In X.509 v2, the specifications of the field "Subject" were ambiguous. "Common name" is a type defined for the Subject field. Since the ambiguity in the specifications of this field made X.509 v2 vulnerable to attacks. There were multiple instances where two implementations of the X.509 system interpreted the same input differently. This led to a CA signing a certificate that it viewed as being granted one privilege, while the client-side application (the web browser) parsed the same input differently. This vulnerability caused “Null terminator attack.” In this attack, the attacker presents a certificate signing request to the CA with the common name as “www.google.com\x00.attacker.com” and get a signed certificate for this Subject Common Name. Let us assume that the CA correctly interpreted the value in this common name field, i.e., the CA parsed this string as Pascal-style string by including \x00. However, other destinations might interpret the string as a C-style string and consider \x00 as end-of-the-string. These destinations might decode the CA’s encoding into a signed message vouching that the certificate is valid for www.google.com.

Therefore, to overcome this vulnerability, X.509 v3 introduced a new field “SubjectAltName” as an extension, replacing the “Subject” field. The “Subject” field is still preserved in version 3 to support interoperability between version 2 and version 3. However, in version 3, the Subject field is never used and is kept empty. In version 3, ASN.1 data structures were not changed from version 2; just a few extra types were defined in the extensions for SubjectAltName.

2.3 802.11i [9]

Wi-Fi is a local area networking technology that uses radio signals to communicate between the devices/mobile stations. It constitutes a wireless local area networking (WLAN) defined by the IEEE 802.11 standards. Wi-Fi is widely used in today's world, covering almost 100% of the WLAN market share. In infrastructure mode, this technology is based on the Client-Server model. The communication happens between an Access Point (AP) - which plays the role of the Server - and a Station (STA) – which plays the role of a Client. For easy understanding, let us imagine an AP as the 802.11i analog of an Ethernet switch. An AP has two interfaces, one on the radio side, which is a set of virtual STAs, and the other on the distributed system side, which is usually Ethernet. The radio interface is used to form associations with multiple STAs simultaneously. The DS interface side is used to connect the Wi-Fi WLAN with the rest of the world. The AP shuttles packets or frames between the two interfaces using switch-like technology, as it knows which MAC addresses can be reached through each of its interfaces. In 802.11i, an AP is a combination of a station and a distributed system.

2.3.1 How 802.11i discovery and association establishment work?

There are typically 6 phases in establishing a Wi-Fi connection:

1. Discovery: A STA must find an AP before it can form any connection, and this process is called discovery. Discovery can occur in two ways: First, every AP broadcasts announcements of its presence, called Beacons, periodically. Second, an STA can broadcast Probe requests asking any AP within radio range to identify itself and to share information about the 802.11i capabilities it supports, such as data rates and security posture. An AP may ignore received Probe requests, but more commonly

it usually responds with a Probe Response. An AP unicasts its Probe Response to the MAC address of the requesting STA. Like a Beacon, Probe Responses contain the AP's SSID (wireless network name), supported data rates, encryption types, and other 802.11i capabilities. If an STA receives Beacons or Probe Responses announcing 802.11i capabilities compatible with the STA, the STA can initiate the next stage of a connection, which is called Association.

2. *Association*: A STA selects a suitable AP from the broadcasted beacons and Probe response it receives. An Association Request is sent by an STA to the selected AP to establish a link. Upon receiving the request, AP decides whether to allow the connection. If the AP can support the request, it sends an affirmative Association Response to the STA. This is to inform the STA that the AP has granted access for STA to exchange MAC level messages between the correct STA and the Distribution System, which is the network behind the AP. If the AP has security enabled, it also sets up a filter to allow only the exchange of 802.1X messages between the STA and the DS.

3. *Authentication*: This is skipped if the AP does not have security enabled. If security is enabled, 802.1X authentication between the STA and the DS is required. Typically, 802.1X authentication is facilitated by an Authentication Server (AS) somewhere within the DS, and an 802.1X proxy running within the AP, which knows how to route 802.1X messages between the STA and the AS. The IEEE 802.1X protocol is also called EAPoL, to denote it uses EAP over LAN, to effect authentication between the STA and AS. 802.11i security assumes that as a side effect of 802.1X authentication a secret Pairwise Master Key (PMK) is established at both the STA and

AS. The PMK is an authorization key whose possession proves the holder is authorized to send and receive messages on the association between the AP and the STA. On successful authentication, the AS sends the PMK to the AP and then deletes its own copy, so the AS cannot impersonate the AP to the STA nor vice versa. Both AP and STA then use the PMK in the next step of the connection life cycle to confirm that they are indeed the legitimate parties authorized to access the association. This next step is called the 4-Way Handshake.

4. *4-Way Handshake*: The 4-Way Handshake is an authentication process based on 802.1X and used to prove (a) the STA and the AP are both authorized to send and receive messages over the association between them and (b) derive cryptographic keys from the PMK to restrict access to the channel between them. This protocol is named the 4-Way Handshake since it requires four messages to achieve its goals. Mutual authentication requires an exchange of at least three messages to prove identity and liveness, but the use of 802.1X, which is a request/response protocol, means a 4th message is used. The second, third, and fourth 4-Way Handshake messages include a Message Integrity Code (MIC) to detect unauthorized forgeries and changes to the information transmitted in these messages. MIC is an alternate name given in this protocol for Message Authentication Code because the 802 community already used the acronym MAC to mean Medium Access Control. MIC is usually constructed over the entire message so that the receiver can check if the message was tampered before it received the message. The protocol also uses nonces, defined as 32-byte (256-bit) random number, that is used in these messages to prove the liveliness of a session. By the Birthday paradox, the number of bits of security you get from randomness is half

the number of random bits, so a 256-bit random number yields 128 bits of security. In this case, security means reuse resistance, so the use of 256-bit numbers, if random, requires 2^{128} guesses to find one of the generated nonces. The nonce sent by the AP is called ANonce, and that by the STA SNonce. ANonce is generated by the AP as a challenge to the STA to prove its responses in the protocol were not generated before the AP sent its ANonce. And, similarly, SNonce is the random number generated by the STA to challenge the AP to prove its responses in the protocol were not generated before the STA sent its SNonce. ANonce and SNonce are used to generate a Key Confirmation Key (KCK) from the PMK so that a valid MIC value proves the liveliness of this session. The MIC computed using this KCK also binds the information exchanged to this session since information or nonces sent in a different session will result in a different KCK and hence lead to an invalid MIC with very high probability. The four messages exchange is depicted below:

a) *Message-1(M1)*: The AP generates a random ANonce and sends a unicast EAPOL message along with ANonce to the STA as soon as it gets PMK from AP after 802.1X authentication. This message informs the STA that AP is waiting for a reply.

   EAPOL-key (Response Required, Unicast, ANonce)

b) *Message-2(M2)*: After receiving M1, STA responds to AP by generating a random SNonce and sending a unicast EAPOL message along with its SNonce and includes an RSN information element (RSN IE) that selects among the security options the AP advertised by including an RSN IE in its Beacon or

Probe Response. The M2 MIC is computed over the entire message, along the ANonce received from M1.

EAPOL-key (Unicast, SNonce, MIC, STA RSN IE)

c) *Message-3(M3)*: After receiving M2, AP computes the KCK using the PMK, the ANonce from M1, and the SNonce from M2, and then checks whether the MIC value sent by the STA in M2 is valid, using the KCK. AP checks whether the RSN IE in M2 selects from the security options, the AP advertises in its Beacons and Probe Responses, and discards the message, aborts, the protocol, and ends the association if not. If, however, the RSN IE selections are valid, the AP generates and sends M3. Message M3 includes the RSN IE it advertises in its Beacons and Probe Responses. It also reiterates ANonce. The AP computes the MIC over this information and adds it to M3, sends M3, and then computes a set of cryptographic keys called the Pairwise Temporal Key from the PMK, ANonce, and SNonce as described in the section 2.3.3.3.

EAPOL-key (Response Required, Install PTK, Unicast, ANonce, MIC, AP RSN IE)

d) *Message -4(M4)*: On the arrival of M3, STA checks for the MIC sent in M3. It also checks that the included RSN IE is identical to that included in the Beacons and Probe Responses received from the AP. If not, the STA abandons the association and reenters discovery; this check defends against a “down-grade attack, where an attacker forges Beacons or Probe Responses that include a security policy different from that specified in the RSN IE the AP includes in its discovery messages. If so, the STA computes the PTK and installs it to begin encrypting the channel with the AP. Finally, the STA encodes M4, reiterating SNonce and including a MIC computed using the KCK over the rest of M4. The STA sends M4 as a confirmation message to the AP.

EAPOL-key (Unicast, MIC)

If anytime the checks fail, the connection is terminated right away. Temporal or session key is installed on both sides, i.e., STA and AP. AP takes the 802.1X filter away, and any message can be now exchanged.

5. Data Transmission: As the 802.1X filter is removed and STA and AP successfully prove their identities to each other, all types of messages are exchanged. All of the messages exchanged during this stage of the connection are encrypted and authentication using the scheme selected by the RSN IE exchange in the 4-Way Handshake.

6. Disassociation: When the STA and AP no longer wish to exchange messages, either can send a Disassociation message to terminate the connection. Both parties delete the PMK and all derivative cryptographic keys upon disassociation.

2.3.2 *Robust Security Network Information Element (RSN IE)*

Element ID	**(In Octets) 1**
Length	1
Version	2
Group Cipher Suite	4
Pairwise Cipher Suite Count (m)	2
Pairwise Cipher Suite List	4-m
AKM Suite Count (n)	2
AKM Suite List	4-n
RSN Capabilities	2
PKMID Count (s)	2
PMKID List	16-s

Figure 2.4: RSN Information Elements

The RSN IE contains the Security policy an AP offers to its client or the selection of features from this policy, a client will use over a session, as shown in Fig 2.4. It conforms to the tag-length-value concept to distinguish it from other 802.11i information elements, but its subfields do not; thus, the expectation is that if later versions have to change the RSN IE format, these changes will be compatible with the existing version of the element. Element ID is the tag, Length has the length of IE, and the rest of the fields are the values. It is present in Beacons, Probe response and Message 2 and 3 of the 4-way handshake. Group Cipher Suite contains the cipher suite selectors used by BSS to protect broadcast messages. The count of all the pairwise cipher suite selectors contained in the RSN IE and their list is present in the next two fields. AKM (Authentication and key management) Suite List contains all the algorithms used to provide Authentication and key management, and AKM suite count had the count of all these algorithms. All the advertised or requested capabilities are present in RSN Capabilities field. The last two fields contain the count and the list of Pairwise master key identifiers that STA feels it is suitable for the AP.

2.3.3 *Key Management*

2.3.3.1. Extensible Authentication Protocol over LANs (EAPoL) [10]

Protocol Version – 1 Octet	Packet Type – 1 Octet	Packet Body Length – 2 Octets
Descriptor Type - 1 Octet		
Key Information - 2 Octets	Key Length - 2 Octets	
Key Replay Counter - 8 Octets		
Key Nonce - 32 Octets		
EAPoL Key IV - 16 Octets		
Key RSC - 8 Octets		
Reserved - 8 Octets		
Key MIC - 16 Octets		
Key Data Length - 2 Octets	Key Data - n Octets	

Figure 2.5: EAPoL message format

Extensible Authentication Protocol is a transport or encapsulation for the authentication protocol. EAP was introduced into the Point-to-Point Protocol (PPP) to support authentication for dial-in Internet service [11]. More importantly, EAP was introduced to allow enterprise IT departments to reuse their existing authentication infrastructures. This reuse was an economic necessity and was one of the things that made dial-in and Wi-Fi successful. Since 802.1X is based on the EAP authentication model, EAP assumes an authentication server, an authenticator, and a client. The authenticator acts as an agent between the client and the authentication server to relay EAP messages between them. The Authentication Server drives the authentication protocol using a request/response protocol. It sends the EAP message to the client, and then the client must respond to it or abort the protocol. Therefore, there is always even number of messages that are exchanged in an EAP protocol. EAPoL is another name for 802.1X, and each EAPoL message carries a single EAP message. EAPoL is just an EAP message and the 802 header needed to send the EAP message over an 802 LAN without a network or transport layer. It is needed because TCP/IP cannot initiate itself until after EAP authentication completes.

EAPoL can transport the cryptographic keys between AP/AS and STA in the message format as shown in Fig 2.5; the only key so transported is the AP’s broadcast key, called the Group Temporal Key or GTK< and which is sent in message M3 of the 4-Way Handshake; in this message the AP encrypts the broadcast key under another key, the Key Encryption Key (KEK) derived from the PMK, ANonce, and SNonce, so only the intended STA can decrypt the key. The field Descriptor Type is a value identifying the IEEE 802.11i key descriptor. The Key Length field is an unsigned binary number representing the length of PTK in bytes or octets. Key Replay Counter is an unsigned binary sequence number which is set to 0 initially when the AP establishes the GTK after a reboot. Client and server keep track of the replay counter to detect replayed messages. In particular, without the latest counter value, an attacker could replay any encrypted broadcast message sent by the AP before the STA’s association, and the STA would otherwise accept as valid; with the current counter value, the STA will drop these out-of-date replays if received. EAPoL-Key IV contains the IV concatenated with KEK while deriving the encryption key used to encrypt the Key data field. Key RSC is the receive sequence counter used in 3rd message of the 4-way handshake to synchronize the IEEE 802.11i replay state. Key MIC is a value generated by a cryptographic function after including all the elements in the EAPoL message frame. Initially, Key MIC is set to 0, and the EAPoL-key MIC is computed. If the key descriptor version in key information is set to 1, then it uses HMAC-MD5 to compute the MIC, and if the key descriptor version is set to 2, HMAC-SHA1-128 is used. Key Data Length represents the length of the Key Data and is an unsigned binary number. Key Data includes all the extra details that are required for key exchange, which are not already present in other fields. It can include information elements like RSN IE or cryptographic encapsulation KDEs like group key ID or Pairwise key ID.

2.3.3.2 802.1X Authentication

802.1X is the transport for EAP over LANs. 802.1X has to act like EAP because EAP messages drive what it does. 802.1X authentication is a process that facilitates the exchange of Pairwise Master Key between a station and an access point with the help of the authentication server, as shown in Fig. 2.3. The AP requests the STA for its information using the EAP Request Identity message. To reply to the identity request, STA sends over the EAP Response Identity containing all the required details about itself. Upon receiving the response, the AP sends this information to the AS using a RADIUS message. The AS then checks how legitimate the identity of the STA is. The

STA can check the authenticity of the AP as well. If either of them cheated about their identity, the process is aborted without going any further. If the STA and AS are successful in authenticating each other's identity, an EAP mutual authentication is exchanged between the STA and AS. After this, the PMK is derived on both STA and AS. There is supposed to be only one copy of the PMK on the infrastructure end and the client end for security reasons. Therefore, the AS sends a RADIUS Accept message, along with the PMK, to the AP and deletes its own copy of the PMK. Finally, after receiving the PMK, the AP sends an EAP success message to the STA informing that it has received the PMK and that the STA and the AP can start the 4-way handshake.

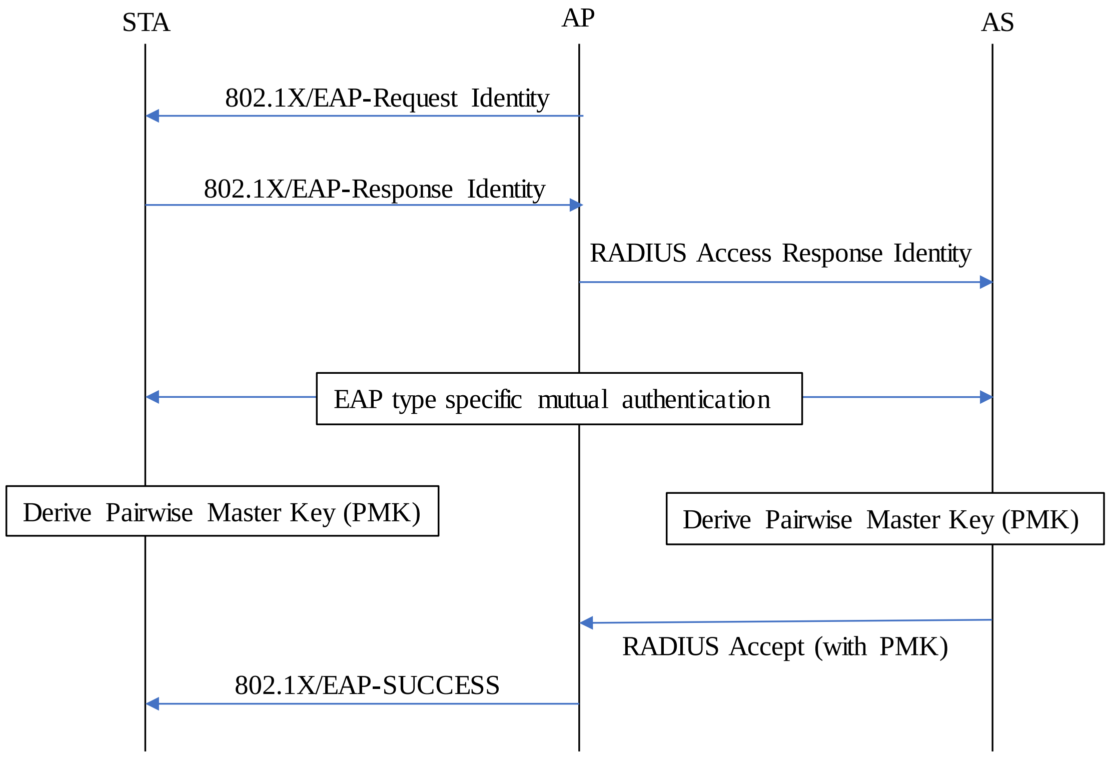
STA
AP
AS
802.1X/EAP-Request Identity
802.1X/EAP-Response Identity
RADIUS Access Response Identity
EAP type specific mutual authentication
Derive Pairwise Master Key (PMK)
Derive Pairwise Master Key (PMK)
RADIUS Accept (with PMK)
802.1X/EAP-SUCCESS

Figure 2.6: 802.1X Authentication Overview

2.3.3.3. Key Generation

It is a challenge to generate, manage, and distributed keys. 802.11i uses fresh keys for each session to prevent replay attacks. 802.11i follows the key hierarchy scheme, shown in Fig 2.3, to manage the keys. The AS and STA somehow derive a pairwise master key; how this is done is outside the scope of 802.11i. The AP and STA then use the PMK to derive a pairwise temporal key or session key. The pairwise transient key is derived using the PRF function, which includes PMK, Access Point's Mac address, Station's MAC address, ANonce, and SNonce as the parameters. The MAC addresses bind the derived keys to the AP and STA. If they are indeed chosen randomly, then ANonce and SNonce guarantee that the derived keys for this session are independent of keys derived for other sessions (the probability of a collision with any other key is $\frac{1}{2^{512}}$). PMK has a PMK Identifier (PMKID) which is derived using HMAC -SHA1 -128 (PMK, "PMK Name" || AP MAC Addr || STA MAC Addr). The label "PMK Name" guarantees that other keys derived from the same PMK are independent of the temporal keys. From the PTK three types of keys are derived; Key confirmation key provides data origin authenticity, Key encryption key provides data confidentiality, and temporal keys are valid only for a particular session which will be mostly discarded when the session expires. It is sufficient only to preserve Master key and not the others. The parties can derive the other keys anytime they want if they have the master key. This also saves memory. The pairwise keys are used in unicast messages to preserve the message content. Similarly, group keys are used in broadcast messages.

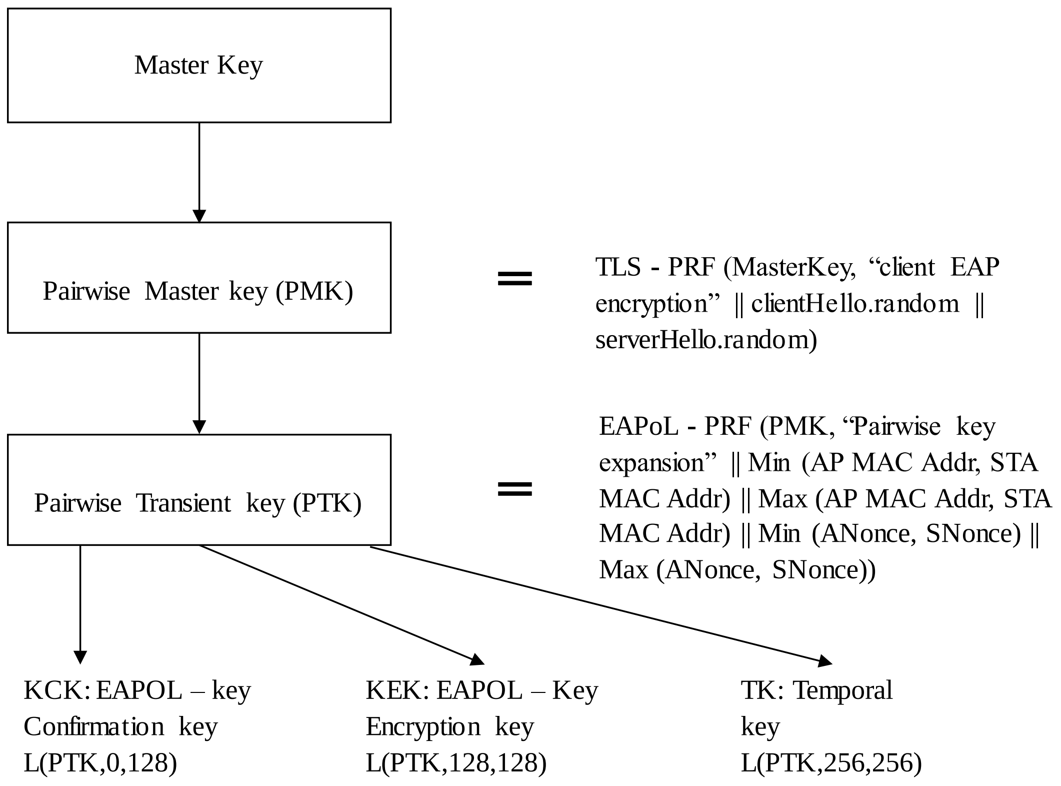
Master Key
Pairwise Master key (PMK)
=
TLS - PRF (MasterKey, "client EAP encryption" || clientHello.random || serverHello.random)
Pairwise Transient key (PTK)
=
EAPoL - PRF (PMK, "Pairwise key expansion" || Min (AP MAC Addr, STA MAC Addr) || Max (AP MAC Addr, STA MAC Addr) || Min (ANonce, SNonce) || Max (ANonce, SNonce))
KCK: EAPOL – key Confirmation key L(PTK,0,128)
KEK: EAPOL – Key Encryption key L(PTK,128,128)
TK: Temporal key L(PTK,256,256)

Figure 2.7: Key Hierarchy Scheme

2.3.4 How 802.11i reuses 802.11's original security architecture

2.3.4.1 Working of 802.11

An 802.11 frame format is mainly divided into two parts – a header and a payload, as shown in Fig. 2.5. There is no 4-Way Handshake in 802.11 to protect the data. Instead, there is a field named "encrypt flag" whose value decides if the payload is encrypted or not. If the flag is set to 0, the payload is unencrypted. And, if the flag is set to 1, the payload is encrypted using Wired Equivalent Privacy (WEP) algorithm. A WEP header has a WEP key ID and an Initialization Vector (IV) as shown in Fig 2.6. The key ID has four values – 0,1,2 and 3; with byte values between 4 to 255 being reserved. A per-packet key used to encrypt the payload is generated by appended the 24-bit IV with the WEP key ID value.

When an 802.11 client receives an 802.11i packet, as it doesn't know anything about RSN IE and since 802.11 follows tag-length-value format, 802.11 parser skips that field and moves on to the next field.

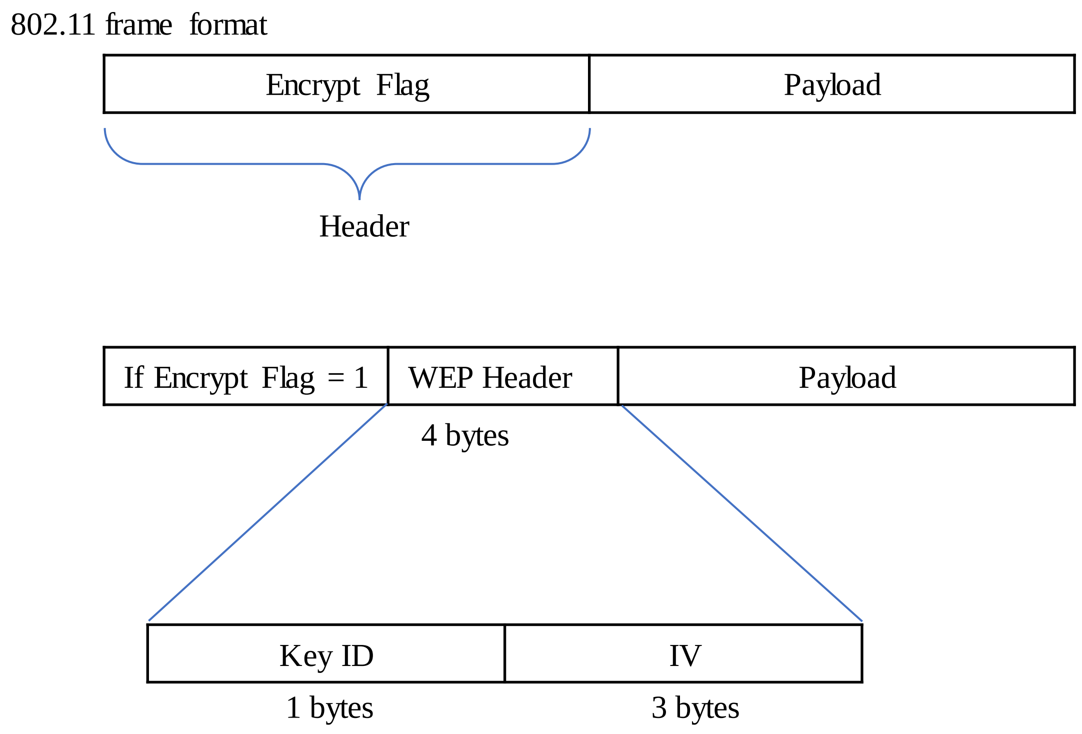

Figure 2.8: 802.11 Frame Format

2.3.4.2 How 802.11i reuses 802.11

Dr. Jesse Walker broke the WEP protocol in July 2000. The 3 bytes (24 bit) IV is too small to avoid key reuse, and the WEP per-packet key is too weak to protect the WEP key from being learned from analyzing encrypted packets; this second fact was demonstrated by Scott Fluhrer, Itsik Mantin, and Adi Shamir in the spring of 2001 [12]. It is also feasible to construct valid WEP encrypted forgeries from captured WEP encrypted packets without the encryption key by modifying a previously encrypted WEP packet. Since 802.11i had to use the first generation WEP hardware used by 802.11, all the previous fields from 802.11 were kept intact, as shown in Fig. 2.6. The encrypt flag is always set to 1, and the WEP header is replaced by an 802.11i header. Along with the WEP fields - “key ID” and “IV,” there are two more fields that were added in 802.11i to increase the security - “Extended key ID” and “Extended IV,” as shown in Fig 2.9. The protocol was designed in such a way that the hardware would read only the WEP key ID and the WEP IV and not the extended key ID and IV. However, the key to encrypt the payload is generated using all the four fields.

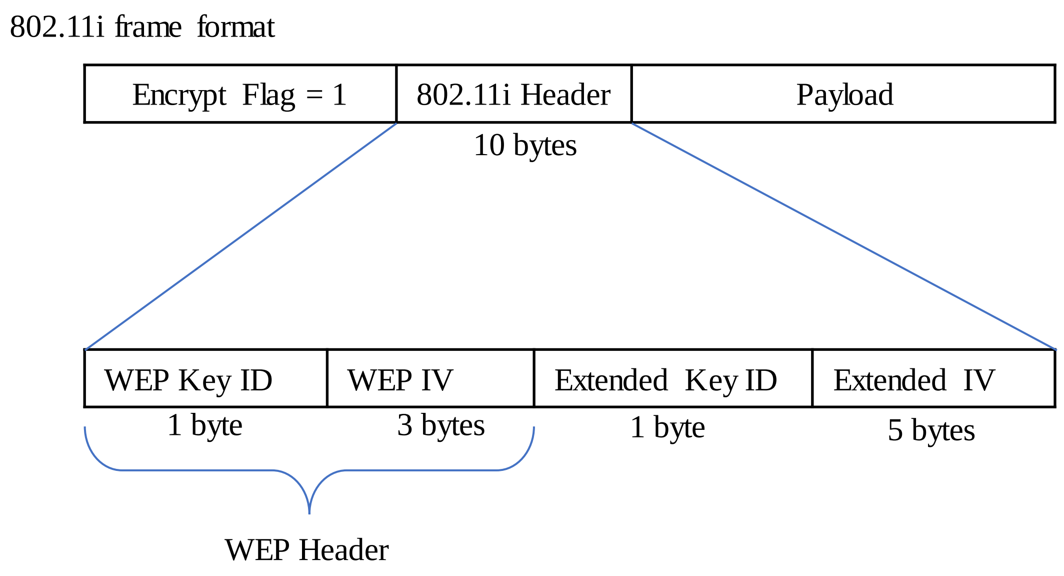

Figure 2.9: 802.11i reusing WEP header

3 Hypothesis

3.1 Lessons learned from prior work

3.1.1 How DHCP reused BOOTP [2], [4], [13]

Two main learnings from this BOOTP to DHCP upgrade is that:

1. Having reserved fields and reserved bits are important concerning upgrades. Since the BOOTP design reserved bits for future use, new features could be added without changing the meaning of the existing fields.

2. Extensions also play an important role in bringing in new changes according to the new needs of the population. New changes were brought in by successfully retaining old features and using reserved fields ignored by BOOTP entities to convey new features. Therefore, clients who are just BOOTP compatible can communicate using DHCP messages, as BOOTP functionality is fully maintained by DHCP. This is an example of interoperability: all the devices on a network should be able to communicate with each other irrespective of the protocol version, and upgraded messages still being interpretable by devices that haven't been upgraded.

3.1.2 How X.509 supports upgrades [3], [7]

X.509 can support upgrades because of the following features.

1. It contains a version number. Version number helps us identify which fields should be populated or present and which shouldn't. Based on that it also specifies how to interpret these fields correctly, i.e., it indicates the semantics attached to each field.

2. X.509 follows ASN.1 encoding rule. Thereby, following tag-length-value format in which each field has a unique tag. Based on each version, if the tag is not recognized, it can be skipped, based on the length field.

3. The certificates accommodate adding new fields in the form of Extensions. Therefore, as the situation demands, new attributes can be added without breaking the system.

4. By marking the fields critical, it mandates the future versions to keep this field. Therefore, criticality can enable interoperability between versions. In particular, the inclusion of a critical extension can guarantee that an entity which supports only an

earlier version will reject a certificate containing a critical field about which it does not know, thus enforcing a security policy that requires process the critical field.

3.1.3 How 802.11i supports upgrade [9]

802.11i supports upgrade in the following ways:

1. 802.11i RSN information element includes a version number. Having a version number helps in introducing new changes and keeping track of the changes. Version number also helps the interpreter parse the packets correctly. The version number indicates how each of the RSN IE fields are to be interpreted.

2. As described in section 2.3.2, RSN IE follows Tag-length-value. This mechanism facilitates interoperability between various versions of the software. If a version doesn't recognize the tag, it reads the length, skips that value and moves on to the next tag.

3. Elements in RSN IE includes a list of items it supports. This facilitates adding new items in future versions

3.2 Requirements to support successful upgrades

There is no way to make a product perfect that meets all the requirements/needs of the users and without making mistakes or misjudgments. Even if this is somehow achieved, the ever-changing needs of software users make upgrades inevitable. We need to consider and accommodate these situations while designing software. Hence, we do work to explicitly enable upgrades. There are numerous principles that are needed to enable graceful upgrades. After studying and analyzing various examples of systems that successfully supported upgrade, this research has the identified design features that seem to make them work. There are few requirements that are to be implemented in the carriers of the information or communication mediums, like messages or certificates, and a few requirements to be considered crucial while designing the software or hardware, like parsers or formatters, receiving the information through these carriers. These negotiations or decisions help in easy transition among various versions, which in turn makes the communication and understanding of these messages effortless. This section will be talking about these requirements that are necessary while designing the data structures of the messages or designing the protocols at the nodes receiving or sending the messages.

1. Version number: It is always necessary to include version numbers in the protocol messages. Each version has its own rules that will be implemented by the formatters or parsers of that version. Therefore, the version number informs the implementation about which rules to follow in order to properly interpret the message and data structure fields supported by that version. A version number by itself is not sufficient to support upgrades. For example, as we will see later on, although Bitcoin has a version number, it does not support changes to its basic data structures (transactions and blocks in the blockchain) without imposing a hard fork on the entire Bitcoin community. [14]. Another example, there is no way to change the hash algorithm or the PoW algorithm. Whereas, in X.509, since the ASN.1 for the "subject" field was ambiguous and led to attacks, the tag-length-value structure of X.509 data elements allowed a new field "subjectAltName" defined using unambiguous ASN.1 to be introduced in version 3 to replace the "subject" field for version 3 X.509 implementations [3].

2. Preserve old features in new versions: While upgrading from one version to another, the newer version might have its own parser and formatter. However, along with the new implementation, it is very important to the preserve and support the old parsers and formatters to support the older versions of the protocol. For instance, let's considering upgrading from version 1 to version 2, it is very important to keep all the existing fields/features present in version 1 while adding new fields/features required version 2, even if the old fields are not required in the new version. This decision might differ if the requirements explicitly specify to remove old functionality. Maybe there are sufficient versions of the older software to justify supporting the old field. For example, as we will see later, in the coding experiment conducted for this research, versions 1 and 2 supported a "name" field, even though version 2 did not need this. It was preserved for interoperability between version 1 and version 2. Version 3 removed the "name" field from its messages because we made an arbitrary decision to no longer require interoperability of version 3 with version 1. Another example, while upgrading from BOOTP to DHCP, DHCP is built on the BOOTP utilizing the reserved fields and not removing any existing fields, so the messages could still be properly interpreted by the BOOTP infrastructure [2], [4]. Using the "Reserved/Unused" bits from BOOTP, DHCP utilizes one bit to build a new field "BROADCAST message," and the remaining bits are still reserved for future use. Therefore, preserving old features in new version helps in unbroken communication between different versions.

3. Tag-length-value: Using tag-length-value implementation for the message formats informs the parser and the formatter implementations about the representation of the information present in that field just by reading the tag. In this type of encoding,

the version number specifies which tags are valid, and the tag for a field tells how to parse the field's value, i.e., it specifies the semantics to apply to the serialized bit string representing the value: will it remain a bit string, become an integer, a compound data structure, etc. Length gives the size of the value this tag represents, and finally, the value is where the actual intended value of the field is present. Each tag value should be unique across version numbers. A version which uses a tag value can correctly format and parse the field; a version that does not use this tag value can use the length field to skip over the field in messages it receives from an implementation that does use the field. This thus differentiates fields by version number and tags. By doing this doing, the version 2 parser can differentiate its native tags from the version 1 tags. This is useful because version 2 or any later versions can skip the tags they do not recognize. The way this works is, the parser of a version first reads the tag ID, if it doesn't recognize this tag, the parser moves on to the next bit and reads the length field, and then skips over those many bytes. This way, old fields can be preserved, and the next versions can just skip over those fields if not recognized. If the old fields are no longer required, they can be removed. And later version can add new features. That is, fields can be added or removed before and after a tag-length-value without affecting how this field gets processed. Therefore, Tag-length-value format provides maximum flexibility.

4. Reserved values: The main reason for having different variants of software is because of the changing needs of the users or exploring new requirements that can be added to the software to accommodate the user needs. If we want to add new requirements to the existing software, we need to save some space for future use

because the future is unpredictable. New parsing and formatting rules can be introduced by having reserved fields and reserved tag values in the data structure of the protocol messages. In practice, this means incrementing the version number and allocating new tag values or fields reserved in prior versions. These reservations are important so that others do not use them for their personal use. All the reserved items should follow the intended standards. For example, in version 1 of the coding experiment, tag numbers till 6 were used and the rest was reserved; the tags 7 and 8 were used in version 2 to define new fields first name and last name. If we did not reserve bits, we would not be able to add these fields. If either the reserved version numbers or the tags number are used deviating from the standards, it will cause confusions. An example for this is 802.11i standards has reserved bits, using this China created a variation of their own called the "WAPI" which did not follow the 802.11i standards and used these reserved bits for their own [15]. WAPI was the alternative for WI-FI. This caused interoperability problems when Wi-Fi entered China's market.

5. Preserve the semantics: Though each version has its own formatting and parsing rules, the semantics of the messages should be preserved. This makes it easy to maintain standards throughout the organization or a hierarchical level and helps to make it easy for the future developers and other team members to understand the code and make changes. In the experiment conducted for this research, when we added new version, we left the old versions of code to handle the prior versions, and we added new code to handle the new versions. Changing the old code increases the complexity and the bug count. It is always recommended to make a copy of the old code and add the new code without changing the old code. The coding style, the naming conventions,

alignment of code, keeping related code together, commenting style, and other necessary semantics were kept in mind while adding new changes. If the new versions change the semantics of any older versions, the other parsers would not recognize these semantics for that field and would result in discarding the message. Therefore, all the messages received from this version will be discarded by other parsers. This fails the main cause of this research, which is to achieve interoperability between versions.

6. Critical flag: It is sometimes useful to extend the tag-length-value construction to include a criticality flag. This is useful when, by policy, we explicitly do NOT want an older software version to process a received data structure when it does not already understand a critical field. The semantics of critical fields in the messages indicate and force the parsers and the formatters to drop the packets if they are not aware of these fields marked as critical. X.509v3 includes a flag for its extension fields, in order to enforce security policies that older versions that do not understand but must not violate [3]. Using the reserved bits, new fields can be added in the later versions. If tag-length-value encoding is not followed, extensions can be used here. Creation of new fields can also be used in an effective way to correct the mistakes made in the past. Just like how extensions are used in X.509. In version 2 and 1, the way "subject" field was encoded, caused a lot of confusions while parsing, which were exploited by the attackers [8]. To solve this problem, in version 3, a new field named "subjectAltName" was created as an extension, with more specific encoding rules, and the original field "subject" was not used anymore. Things could go wrong if either of these did not have values. A rule was also made to which indicated that if "subject" field is not populated with some value, the field "subjectAltName" should have value. To achieve this, the concept of

critical fields was used. That is, in version 3, “SubjectAltName” was made mandatory by marking it critical. The way critical fields work is, the fields marked are to be always used while constructing a message. If any of the parser receiving the message doesn’t know how to decode it, then the packet should be discarded. If the parser knows to decode this field but has not received it in the packet, even in this case the packet should be discarded. They are a little different from tag-length-value here because critical fields cannot be just skipped over if they are not recognized. Critical fields are important in financial type transaction and are useful in messages containing policy-driven information.

There are also special rules for versioning in protocols that support negotiation.

1. Include a list of supported version numbers when possible: By including the list of version number a device supports in its message, the device is making it easy for the parsers or formatters at the receiver’s end, to decide whether the receiver supports one or more versions from the list and select the most favorable versions or features from among those advertised. Since large distributed systems consist of software with different versions, it is necessary to discover the best version common to sender and receiver whenever possible. This is usually the highest version number supported by all members of a group of communicating implementations. When the system dynamics makes it feasible to do so, it is recommended that communicating implementations negotiate a favorable version number and then communicate using the data structures and messages for that version. For example, an 802.11i access point publishes in its beacons the minimum version and the maximum version number it supports. 802.11i clients then select the highest version number they share with the

access point to adopt the most advanced security features [9]. This is an effective way of eliminating unsuitable peers. 802.11i server also broadcasts the security implementations it supports in its beacons and probe responses in its RSN IEs. In X.509, the list of extensions that version 3 supports is presented [3].

2. Negotiations: To achieve all the above-mentioned requirements, it is often useful to have some sort of negotiations while agreeing on the tags, versions, parsing rules, encoding rules, and formatting rule. Wi-Fi uses negotiation to establish version and features [9]. This is possible because the AP and STA must agree on version and features to operate correctly, and so Wi-Fi uses an interactive request-response protocol for this purpose. Only the receiver and sender should agree on these rules. As we will see later, the broadcast communication protocol in Bitcoin does not support negotiations [16]. Instead, every member of the Bitcoin network must support the same messages, since Bitcoin is based on a non-interactive broadcast protocol.

3.3 Interoperability Trade-offs

There are situations where choosing interoperability would not be the best decision. The protocols or the applications we have studied so far in this thesis also encountered these circumstances where other tradeoffs were chosen upon interoperability. Therefore, it is necessary to break interoperability to achieve other goals. Few of those examples are listed below:

1. As a security measure: For security reasons, X.509 decided to break interoperability among its versions. Version 3 was designed not to communicate with version 1 and version 2. As discussed in section 2.2.7, the common name type in the subject field had a vulnerability. To prevent downgrade attacks, X.509 v3 certificates

included the concept of criticality into their certificates. The “SubjectAltName” was made critical in version 3. If the older versions received this certificate and if they did not support the critical fields, they simply discarded the packets. Therefore, to prevents bugs in the older version onto the newer versions, interoperability can be broken.

2. To force an upgrade: It is economically difficult for the vendor to support many versions of the software. Therefore, all the users of that software might be forced to upgrade to a certain version. As we will later in section 4.3, the coding projects forces the users of version 1 to upgrade at least to version 2. Version 1 and version 2 of the project are interoperable, and version 2 and version 3 are interoperable. However, version 1 and version 3 are not interoperable as version does not support the “name” field.

3. Hardware compatibility: The RSN IE of 802.11i do not follow tag-length-value throughout. Instead, few of the fields follow just length-value format. This is because tag-length-value format takes up too much bandwidth, and it becomes too expensive to use hardware that would support flexibility to this extent. Though using length-value format does not provide complete interoperability when compared to tag-length-value, it is advised to follow this format to ease the implementation of the parser when hardware parsing is used instead of software.

4. Network bandwidth limitation: Using tag-length-value format might facilitate flexibility and efficiency at the software implementation level, but having a tag and a length field for every field in the message increases the size of the message. So, tag and length are extra information that is to be transmitted for the parser to understand the message and might take up bandwidth more than a communication link would tolerate.

In some cases where the bandwidth is low, it is easier to have fixed formats to transfer limited data.

4 Testing the Hypothesis

4.1 Bitcoin [16]–[19]

Bitcoin is a cryptocurrency, a version of electronic cash. It is an alternative currency to the existing traditional physical currencies. Cryptocurrencies enable users to exchange value digitally without the involvement of without an infrastructure of accredited brokers or governments. Bitcoin is a billion-dollar economy, and there are millions of users participating in the Bitcoin community.

Since Bitcoin is a distributed system, it is a P2P protocol, meaning all the nodes running the Bitcoin protocol are each other's peers, and they have no hierarchical relationship among them. All nodes have equal access to all Bitcoin activities. These nodes of the Bitcoin ecosystem are spread across the world and are not controlled by any centralized authority, which makes Bitcoin a decentralized distributed system. Any node can join the Bitcoin Network and do the operations without registering or requiring any approval from any peer. No node is enrolled for any special treatment.

A unit of currency in Bitcoin is a bitcoin. It can be represented using these symbols BTC, XBT, and ₿. The smallest fraction of a bitcoin is 10^{-8} bitcoins and called as a satoshi, i.e., a single bitcoin can be divided up into 100 million satoshis.

Nodes or the clients in the Bitcoin network can acquire bitcoins in only two ways. One is by mining bitcoins, and the second is when a bitcoin owner transfers some number of satoshis to a peer. There are two types of Bitcoin functionality: a wallet and a miner. Wallets are used to store, receive, and send bitcoins. A miner is an entity which verifies these exchanges of bitcoins among wallets.

A wallet can send bitcoins to another node in the Bitcoin ecosystem using transactions. These transactions contain all the necessary details about who is sending, how many bitcoins, and to whom. Since there is no centralized authority to verify these transactions, Bitcoin uses the Nakamoto consensus protocol, described later, to approve these unverified transactions. This protocol is used to establish consensus among all the peers of the Bitcoin community about which are the valid transactions.

A miner collects valid unverified transactions and forms them into what is called a block. PoW, a challenging computational puzzle, is a lottery that decides which miner gets to publish a block of verified transactions. After winning this puzzle, the blocks are published by the miner into a back-linked list called the blockchain. The blockchain is publicly available to all peers. Miners add new blocks to the blockchain. Once a miner adds a new block to the blockchain, the community assumes it cannot be edited or deleted by anyone. Hence, the transactions in a published block in the blockchain are considered confirmed. On an average, a block is added, or in the Bitcoin jargon, mined every 10 minutes.

4.1.1 A Brief History

Bitcoin is an online P2P decentralized cryptocurrency. A domain with the name "bitcoin.org" was registered in August 2008, its initial software released as open-source software in January 2009, and a paper with the title "Bitcoin: A P2P Electronic Cash System" was posted to a security mailing list called the Cypherpunks. All this was issued under a pseudonym, Satoshi Nakamoto.

A main intent of the Bitcoin protocol was to make transactions between the Bitcoin nodes untraceable. This intent was not realized, although it took a number of years to realize; and the misconception encouraged various black markets, like illegal weapons sales, money laundering, copyright pirating, human trafficking, etc., used bitcoins for their transactions. This illicit activity started when bitcoins were used in Feb 2011 by a very popular black-market website named Silk Road. Silk Road was a platform used for selling illegal drugs. This website was seized and shut down by the FBI in Oct 2013. The notion of untraceable payments is a misconception in Bitcoin because every Bitcoin transaction is recorded in the public blockchain and hence can be traced. However, it might take more time to trace Bitcoin transactions when compared with the time taken to trace transactions that use traditional currencies which are monitored by some centralized authority.

An essential element of Bitcoin is the public ledger known as the blockchain. Auditable electronic cash was proposed in the late 1990s, where the banks maintained a public database to detect double-spending. This design served as a good inspiration to the Bitcoin community to fight against double-spending (described in section 4.1.7.4).

4.1.2 The blockchain

The base or the initial block of the blockchain is called the genesis block and was created in 2009. The genesis block creates 21 million bitcoins, all that will ever be, and all these bitcoins will be transferred to miners by the year 2140. The genesis block is the common ancestor of all the blocks. The genesis block was mined in 2009 for the first time. The Bitcoin currency was used in 2010 for the first time to buy a pizza in exchange for 10,000 bitcoins. Just like any other traditional physical currency, the Bitcoin currency can be converted into most of the traditional currencies used across the world.

In a system that is governed by a central authority, the verification of any transaction is done by this authority and everybody trusts and agrees to the decision taken by this authority. Since Bitcoin is a decentralized distributed system, there is no central authority to verify the transactions and make decisions. A key building block of Bitcoin is the challenging computational puzzle used to verify the transactions. These are called the PoW puzzles and were proposed in the early 1990s to fight email spam; the PoW puzzle will be described later. In Bitcoin, PoW demonstrates to all the Bitcoin nodes that a miner has confirmed a set of transactions and thereby enables the community to reach consensus about which transactions are valid. Each block contains a PoW challenge which the miner who publishes the block containing this transaction must choose to solve this PoW puzzle.

A miner who solves the PoW puzzle publishes its block into the blockchain by broadcasting it to the entire Bitcoin network. Other miners may notice this new block and try to add their own blocks onto this one, thus, extending the blockchain. Since more than one miner might try to add a block to the blockchain at the same time, the blockchain could begin to grow a tree-like structure instead of being a chain of blocks. However, miners overcome this problem by adding new blocks to the longest sub-chain of which they are aware, and eventually one chain grows longer than the rest, and all blocks of this longest one become obsolete. Once this block is published into the blockchain and has been extended six times, all the nodes on Bitcoin network have reached consensus, i.e., all the nodes are bound to agree to this decision of adding this block into the blockchain.

We will describe the blockchain in more detail in section 4.1.7.

4.1.3 Key Management

4.1.3.1 Private and Public Keys

Ownership of bitcoins is determined using a pair of keys – a public key and private key – in the EC-DSA signature algorithm on the elliptic curve secp256k1. A pair of keys consists of a public key which is derived from a private key and a generator point on the curve using elliptic curve point addition. The private key is a 256-bit (32 bytes) random number between 1 and 2^{256} picked randomly by the end user's Bitcoin software. The bitcoins in a wallet are some number of satoshis bound to the public key. A wallet can have multiple key pairs. Therefore, it is very important to back-up the private keys in a secure location so that they can be recovered if lost. If the private key is lost and cannot be recovered, then the bitcoins associated with this private key are lost forever, i.e., these bitcoins cannot be used ever again. These bitcoins are called the "zombie coins." The public key is derived from this private key using elliptic curve addition of the fixed generator point *G* for the curve

$$K = k \times G,$$

Where *k* is the private key, *G* is the constant point on the curve, called the generator point because every point on the curve is of this form for some *k*, and *K* is the resulting public key. When the curve and the generator point is carefully chosen, it is computationally infeasible to calculate the private key *k* using the public key *K* and generator point *G*.

4.1.3.2 Bitcoin address

Bitcoin uses addresses to identify public keys to send and receive bitcoins. A Bitcoin address is a 160-bit number (20 bytes) generated from the public key using SHA256 and RIPEMD160:

address := RIPEMD160(SHA256(public-key))

SHA256 and RIPEMD256 are one-way cryptographic hash algorithms. The results of these functions are hashes. Hashes are like fingerprints, unique, and uncorrelated with their inputs. An address is used to represent the recipient's identity similar to an email address or a beneficiary name on a cheque. Bitcoin encodes addresses in Base 58 to make them human-readable and to avoid digits, and letters than appear similar in form, i.e.., 0, O, and o, and 1, l, and I have been removed from the usual base 64-character set.

4.1.3.3 Wallet(s):

Wallets are the structured, allegedly secure, personal databases to store keys (private and public key pairs) and the number of bitcoins associated with each of these keys. There is no need to store a Bitcoin address, as the public key can be efficiently computed from the private key and the address from the public key. When someone wants to join the Bitcoin community, they can download the wallet application. One such well-known wallet application is the "Satoshi Client" available on bitcoin.org. A wallet can generate the key-pair using the key generator that comes along with an elliptic curve package implemented by the wallet and the private key (32-byte random number). These wallets can then be used to send and receive bitcoins while maintaining a count of the bitcoins bound to each public key in the wallet. When receiving the bitcoins, it checks to see if the transaction containing those bitcoins is confirmed and then it updates its public keys with those bitcoins.

The keys are stored online in these kinds of wallets. The wallets are prone to attacks, as adversaries often want to steal the bitcoins stored in a wallet by gaining access to the private keys it contains and transferring any available bitcoins to themselves. It is the user's responsibility to keep their keys safe and protected.

4.1.4 Bitcoin transaction lifecycle

A Bitcoin transaction goes through the following steps:

1. Creating transactions: The lifecycle of the transaction starts by creating the transaction. This is known as the origination phase. A transaction specifies the number of satoshis being transferred from one address to another. This number of satoshis mentioned in the transaction cannot exceed the number of satoshis bound to the wallet address of the sender of this transaction. As described in section 4.1.5, a transaction contains all the necessary details about the sender and receiver of the coins, along with the number of bitcoins that are being transferred. Once a transaction assigns bitcoin from a sender key (sender) to a receiver key (receiver), the sender's private key must sign the transaction, as this is the only way to authorize the transaction

2. Broadcasting the transaction: After signing the transaction, the wallet broadcasts it on to the entire Bitcoin network. Since Bitcoin is TCP-based, broadcast here means that each transaction is flooded to each neighbor recursively, until it has traversed the entire network. However, while propagating these transactions, the nodes check if they have already seen the transaction so that they do not propagate any transaction multiple times. Each node maintains a list of recently-seen transaction it

has come across. Since the transaction does not contain any private or confidential information, and since the transaction signature(s) protect it from modification, it can be broadcasted on any insecure network like Wi-Fi, Bluetooth, text message, skype or through web forms. The transaction is broadcasted to make every Bitcoin node aware of this proposed transaction so that one or more miner receives the transaction and can then attempt to verify it. However, before propagating each node validates the transaction independently. Therefore, an invalid or infected transaction should not propagate to more than one node. To prevent spamming or Denial of Service (DoS) attack each node checks if it has already propagated this transaction.

3. Mining: Once a miner receives the transaction, the miner verifies the bitcoins in the transaction have not already been spent by the sender, and, if so, the miner adds the transaction to a pool of unverified transactions. A miner picks from 1 to about 2^{11} unverified transactions from its pool. The motivation for miners behind mining is incentives they receive for publishing a block of transactions. The miner has to solve the computational puzzle to win the competition to publish the block. To win the race, the miner chooses a random number and then computes the hash of the random number together with the selected unverified transactions. If the hash has a prescribed number of leading zero bits, then the miner broadcasts the block to the entire Bitcoin network; otherwise, the miner picks another random number or modifies the list of selected unverified transactions (or both) and tries again. If instead, a miner receives a published block from another miner before it completes this process, it discards its selected list, removes all the transactions verified in the new block from its pool of unverified transactions, and starts the entire process over again. The PoW algorithm is so-named

because it demonstrates that a miner solved the computational puzzle. In particular, every node in the Bitcoin network can easily verify the PoW, but, since the output from hash functions are uncorrelated with their inputs, the PoW algorithm is a random search which on average requires the generation of 2^n different random numbers to generate a hash whose first n bits are zero – the miner must do substantial work to win.

4. Confirming a transaction: A transaction in a published block is considered verified. Because published blocks propagate across the Bitcoin network at a finite speed, more than one miner can publish different blocks before anyone of them propagates across the entire network. This kind of race condition is resolved by considering no published block as valid until it has been extended by a chain of at least six blocks. According to the protocol specification, this race condition is resolved by requiring miners to add their blocks to the longest chain of blocks of which they are aware. When some chain becomes six blocks longer than another chain, all the blocks of the shorter chain up to the ancestor block common to both chains are deleted. The miner who published the first block on the longest chain beneath this common block then receives a reward in bitcoins mined from the Genesis block and a transaction fee from the sender of each transaction in the published block.

4.1.5 Bitcoin transaction

Transactions are the heart of the Bitcoin system; everything else is designed to support transactions. A transaction is a data structure or message that transfers bitcoins from the source of funds, called input, to the destination, called an output. A transaction can contain multiple inputs and multiple outputs. Since a single user can have multiple pairs of private and public keys, and each public key has bitcoins bound to it, a user can have individual input for each of these public keys. As a single user can transfer bitcoins to multiple recipients or multiple bitcoin addresses, a user’s transaction can have multiple outputs. The minimum size of a transaction is approximately 256 bytes, with one input and two outputs and the average size of a transaction would be about approximately 280 bytes with one input and three outputs. A transaction contains a version, number of inputs, the list of transaction inputs, number of outputs, the list of transaction outputs and Locktime as shown in Fig 4.1. Section 4.1.5.1 contains an explanation of the structure of inputs and outputs.

Size	**Field**	**Description**
4 bytes	Version	Specifies the formatting and parsing rules the transaction follows
1-9 bytes	Input Counter	Number of inputs included in a transaction represented as a variable-length integer
Variable	Inputs	One or more transaction inputs
1-9 bytes	Output Counter	Number of outputs included in a transaction represented as a variable-length integer
Variable	Outputs	One or more transaction inputs
4 bytes	Locktime	Block Number or Unix timestamp. It indicates the earliest time a transaction can be added to the blockchain. It is usually set to 0 to indicate immediate execution.

Figure 4.1: Structure of a Bitcoin Transaction

The inputs and the outputs are length-value formats as shown in Fig 4.2, i.e., before the actual input or the output field, the length of the input or the output field is provided.

This format supports having multiple inputs and outputs. However, it does not provide the level of flexibility provided by tag-length-value format. The length of these can be encoded in 1 to 9 bytes varying the value of the length field as shown in Fig 4.2. If the length of an input or an output is less than 253, it is encoded as 1-byte unsigned integer. If the length is less than 65535, then it is encoded as a 3-byte unsigned integer. The first byte of the 3 bytes contains 0xFD to indicate the parser that the value is more than 253, and there are two more bytes to read. Next, if the length is more than 0xFD and less than 4294967295, it is encoded as a 5-byte unsigned integer. The first byte of the 5 bytes contains 0xFE to indicate the parser that there are four more bytes to be read. Finally, for any value higher than 4294967295 is encoded as a 9-byte unsigned integer with the first byte containing 0xFF.

Value	**Storage Length**	**Format**
<0xFD	1	unit8_t
<= 0xFFFF	3	0xFD followed by the length as uint16_t
<= 0xFFFF FFFF	5	0xFE followed by the length as uint32_t
-	9	0xFF followed by the length as unit64_t

Figure 4.2: Input and Output Length Encoding

4.1.5.1 Transaction outputs and inputs

A transaction in the Bitcoin network is called unspent transaction output (UTXO). Each transaction has a set of inputs and outputs to indicate who is transferring the bitcoins to whom respectively. That is, a transaction can have multiple inputs and outputs. For example, if a node doesn't have enough bitcoins bound to a single public key, then the node has to combine multiple public keys to add up to an amount exceeding the required number of bitcoins, where each public key would constitute an input. A transaction can have multiple outputs if a node needs to send the bitcoins to multiple destinations.

Transaction input contains a transaction hash, output index, unlocking script, and its size and sequence number. All the details of a transaction input are depicted in Fig 4.3. Since, there are only two ways to obtain bitcoins: either by mining them from the genesis block or by receiving bitcoins from other users; every transaction input is a reference to an output from a previous transaction, meaning, the bitcoins used in this input were received previously as a transaction output in some earlier transaction. Therefore, it contains the hash of that previous transaction in the field "Transaction hash," which is a double SHA256 computed over that entire transaction. Since there can be multiple outputs in a transaction, the field "Output index" is used to indicate the specific output of this input in the referenced previous transaction. The input script – "*ScriptSig*" provides the public key of the receiver of the bitcoins and a signature of the sender over this public key. The public key provides the bitcoin address of the recipient, and the signature over this public key eliminates the chance of false claiming of bitcoins by editing the public key. That is, if an adversary tries to change the public key and replace it with his public key to redirect bitcoins to their bitcoin address, the adversary would not be successful because the signature provided in the input script would be over the public key provided by the sender and not the public key changed by the adversary. The input script is explained in section 4.1.5.2 There is also a sequence number field which is usually not used.

Size	**Field**	**Description**
32 bytes	Transaction hash	Pointer to the to-be accepted UTXO
4 bytes	Output index	Index number of the to-be UTXO
1-9 bytes	Input-script size	Length of the unlocking script in bytes
Variable	Input-script	The input script is called ScriptSig. It contains the public key of the recipient: which computes to the bitcoin address of the recipient; and a signature made by the sender, using their private key, over this public key.
4 bytes	Sequence number	Currently disabled

Figure 4.3: Structure of an Input Transaction

Next, a transaction output contains the number of satoshis that are being transferred in the transaction, an output script and its size. The structure of output is shown in Fig 4.4. Output script called the "*scriptPubKey*" contains the address of the recipient and opcodes to perform various operations to verify the signature and public key present in the input script against the bitcoin address provided in the output script (explained in later section 4.1.5.2).

Size	Field	Description
8 bytes	Amount	Bitcoins, denominations in Satoshi
1-9 bytes	Output-script size	The size of the locking script in bytes
Variable	Output-script	This field consists of a script known as ScriptPubKey. It consists of some opcodes and a public key address. The opcodes say the number of satoshis in the output transaction to this public key. There can be different opcodes to perform different operations to verify the receiver of these bitcoins

Figure 4.4: Structure of an Output Transaction

A transaction typically contains a minimum of one input and three outputs. Let's consider the example where a node needs to send *j* satoshis. Suppose the sending public key has some number, say *i*, of satoshis bound to it. Assuming *i* is the nearest denomination to *j* and $j < i$. As we know, the miners get incentives for publishing a block (explained later). One of the types of incentives is the transaction fee a sender needs to pay the miner for verifying his transaction, i.e., publishing the block containing this transaction. Therefore, one of the sender's output should send *m* satoshis to some miner. This is also called the Coinbase transaction (explained later). Next, the rest of the satoshis, i.e., $i - m - j$, should be addressed back to the sender as change. If the sender wants to receive the change, the sender must specify an output as one of its own addresses. Then the input contains the information of the sender. Therefore, there are three outputs, one to the receiver, one to the miner, and one to the sender himself. A second example uses multiple inputs, i.e., $i < j$. Suppose a sender needs to send *j* bitcoins, but none of his public keys has at least *j* bitcoins individually. Since a sender can have multiple public keys, the node may select from the available public keys containing various denominations of bitcoins to form the amount greater than or equal to the required transaction amount. In this way, a transaction can have multiple inputs.

4.1.5.2 Scripts [20]

A Script is a list of instructions describing how a node verifies the transfer of satoshis in the input and output scripts. Typically to validate a transaction, Bitcoin relies on two types of scripts: input script and output script. The input of a transaction contains the input script called the *ScriptSig*. It contains the public key (<pubk>) of the recipient receiving the bitcoins and a signature (<sig>) by the sender or the current owner of these bitcoins over this public key (<pubk>). These parameters are provided so that the recipient can verify and redeem the bitcoins in the output of the transaction.

<sig> <pubK>

The output of a transaction contains the output script called the *scriptPubKey*. It contains the following parameters:

OP_DUP OP_HASH160 <PubKHash> OP_EQUALVERIFY OP_CHECKSIG

The output script contains the opcodes: OP_ DUP, OP_HASH 160, OP_EQUALVERIFY and OP_CHECKSIG and the recipient's bitcoin address, which is the hash of public key of the recipient (<PubKHash>). These opcodes are to verify the public key and the signature provided in the input script alongside the bitcoin address present in the output script. This verification process is explained later.

A transaction is verified by executing the input and output script simultaneously but separately, and the stack is transferred between the two executions. It is done this way because of a vulnerability that allowed infected unlocking scripts to push data on the stack (the same stack was used for both executions), allowing it to corrupt the locking script.

4.1.5.3 Scripting Language

Bitcoin's scripting language uses pushdown automata (PDA) to verify the contents in the input and output scripts. Pushdown automata interpret context-free languages which can accommodate an infinite amount of information temporarily using a stack. A PDA has three main components: an input tape, a control unit, and a stack. An input tape in case of Bitcoin are the elements of the Bitcoin input and output script. The input tape is divided into symbols, and each symbol is read from left to right at a time. A control unit has a pointer which points the current symbol, which is to be read from the input. The size of a stack is infinite. Each of the symbols from the input tape is fed into the stack using the control unit. A PDA stack does two operations push and pop. Push adds items on to the top of the stack and pop reads and removes the item or symbol at the top of the stack. Since a pushdown automata stack follows last-in-first-out the Bitcoin scripting language executes the script from left to right. This language was designed to carry out very basic operations on the stack. These symbols in the scripts are either constants or opcodes. These opcodes specify the type of operation to be performed on these constants. The opcodes used in Bitcoin scripts are OP_DUP: duplicates the top of the stack, OP_HASH 160: the hash is computed over top of the stack, OP_EQUALVERIFY: checks if the top two items of the stack are equal, and OP_CHECKSIG: checks the signature for the top two items of the stack.

4.1.5.4 Pay-to-Public-Key-Hash (P2PKH)

Bitcoin uses Pay-to-Public-Key-Hash(P2PKH) script to verify the inputs and their respective outputs of a transaction. The input script, *ScriptSig*, contains the public key (<pubk>) of the recipient and the sender's signature (<sig>) on this public key. And the output script, *scriptPubKey,* contains OP_DUP OP_HASH160 <PubKHash> OP_EQUALVERIFY OP_CHECKSIG. To follow the instructions and redeem the coins, the following Table 4.1 illustrates the checking process. If at any point any of these checks fail, the transaction is marked invalid and is discarded.

Table 4.1: P2PKH checking process

Stack	**Script**	**Description**
EMPTY	<sig> <pubk> OP_DUP OP_HASH160 < PubKHash > OP_EQUALVERIFY OP_CHECKSIG	scriptPubKey and scriptSig are loaded on to the stack
<pubk> <sig>	OP_DUP OP_HASH160 < PubKHash > OP_EQUALVERIFY OP_CHECKSIG	Constants are pushed to the top of the stack
<pubk> <pubk> <sig>	OP_HASH160 < PubKHash > OP_EQUALVERIFY OP_CHECKSIG	Top of the stack id duplicated
<pubHashA> <pubk> <sig>	< PubKHash > OP_EQUALVERIFY OP_CHECKSIG	Hash is computed over top of stack
<pubKHash> <pubHashA> <pubk>	OP_EQUALVERIFY OP_CHECKSIG	Constant is pushed to the stack

<sig>		
<pubk> <sig>	OP_CHECKSIG	It is checked if the top two items of the stack are equal. If equal, they are popped out of the stack
TRUE	EMPTY	Signature is checked for the top two items of the stack. If it is true, then the transaction is verified successfully.

4.1.6 *Bitcoin Mining Process*

Bitcoin mining is a process of adding transactions on to the public ledger or the blockchain. Transactions transfer bitcoins (satoshis) between wallets. The mining process decides which of these transfers are accepted by the Bitcoin community. The mining process consists of miners, PoW, and incentives. Miners are the end-entities driving the mining process with the help of advanced computing devices called the mining rigs. To decide which transaction should be added to the Blockchain, along with the signature verification done in P2PKH (explained in section 4.1.5.4), the miners use the PoW algorithm to verify the transactions to check and eliminate any malicious activities. Miners get incentives in the form of transaction fees and rewards. The senders or the creators of the transactions pay the miners transaction fees, called the Coinbase transactions, for verifying and adding their transactions on to the Blockchain. In addition to the transaction fee, miners get rewards, generated from the genesis block for the same. All these concepts are explained in later sections.

Miners have mining rigs specially built for this purpose. These mining rigs consist of high-performance CPUs with more cores than regular or special crafted ASICs. Mining rigs cost and consume a great amount of resources like electricity, money, hardware components, etc. because of which miners sometimes steal the resources. The Bitcoin design motivates miners to perform the verification function by presenting the winner of the lottery to publish a new block with a reward in the form of bitcoins received from the genesis block. Anybody in a Bitcoin network can be a miner, but only a limited number of Bitcoin network participants can afford to run a mining rig.

A miner is always listening to the proposed transactions broadcast by the wallets. As soon as a miner becomes aware of a newly broadcasted transaction, it verifies that the input bitcoins have not already been spent, and, if not, adds the transaction to a temporary pool of unverified transactions maintained locally by the miner. The miner picks from 1 to about 2^{11} transactions from its pool and puts these into a buffer representing a proposed new block. A miner verifies the transaction to check against double-spending, i.e., the miner checks to see if the bitcoins included in this transaction were ever used previously, refer section 4.1.7.4 for more information. Only one miner out of all of them gets to publish a block at a time. To compete with other miners and to get the incentives or rewards, a miner should solve the PoW faster than others.

To publish a block, the miner has to win a race by solving a challenging puzzle called the PoW. To solve a PoW, a miner has to pick a nonce, which is a random number to match the target, which is used to set a threshold value. A target is a 256-bit number. The lower the target, the more difficult it is to compute the right hash. Harder the difficulty target, the more computing resources it takes to solve the PoW. The

Bitcoin automatically maintain a target for the PoW. The target is to treat the hash of a block as a little-Endian 256-bit number smaller than a threshold value. The threshold value is chosen so that the binary representation of numbers smaller than it will have a specified number of leading bits equal to zero. Since SHA256 acts like a random mapping, finding a block of unverified transactions and a random number whose hash results in a value smaller than the threshold is essentially a random search, for which there is no shortcut; if a shortcut did exist, this could be used to break SHA256 collision resistance and preimage resistance. Thus, being a random process, the PoW algorithm requires every Bitcoin miner to selected trillions of random numbers and computing the resulting hash to see whether the right number of leading bits are all zero. On the other hand, any Bitcoin peer can verify that the winning miner did this work by computing a single hash of a published block and seeing that the hash has the required number of leading zeroes. Therefore, the algorithm has the property to prove that the winning miner did the work needed to win the lottery. The number of miners and the mining rigs used by them is not constant. If, it becomes easier for a miner to solve the PoW puzzle with multiple mining rigs than with fewer since he can search a larger space with more rigs. It is expected that over time, along with the advancement in technology, the miners and the computers would increase. To cope up with these external factors, the Bitcoin network adjusts its difficulty, called the target, to keep the block generation time at about 10 minutes, regardless of the amount of computational resources brought to bear on PoW. This means the difficulty target is set to whatever mining power will result in a 10-minute block interval. This difficulty is adjusted automatically by the Bitcoin protocol on every node for every 2016 blocks, i.e., for every two weeks. The miner chooses a nonce (a random number) and then computes the hash of the random number together with the selected unverified transactions. If the hash has a prescribed number of leading zero bits i.e., the hash should be less than or equal to the difficulty target, then the miner signs the block and broadcasts it to the entire Bitcoin network; otherwise the miner picks another random number – a new nonce is selected until the desired hash is achieved - or modifies the list of selected unverified transactions (or both) and tries again. Meanwhile if any other miner is successful in solving the PoW and publishes a block, this miner receives the published block, the miner checks his selected list or his potential candidate block, that he/she was trying to publish, to see if any of the transactions in the published block were included in his block. If there is a match, these already published transactions are removed from its pool of unverified transactions and the candidate block. The miner then starts by recreating his candidate block with other unverified transactions from the pool and starts the entire process of solving the PoW all over again.

On average, a new block is mined every 10 minutes. Once the block is published and added to the blockchain, it is considered to be confirmed once. It takes six confirmations for a block never to be altered. At this point, all the nodes of the Bitcoin network have reached consensus. Everybody agrees that the block with six confirmations is legitimate and trustworthy.

Bitcoin automatically maintains the 10-minute rate by using a difficulty target to regulate the rate of new block production by miners statistically. Bitcoin adjusts the number of leading zeroes in the hashes produced by the PoW algorithm to maintain this rate; since finding a hash digest with n leading zeroes takes about 2^{n-1} invocations of the hash algorithm, adding or removing the number of leading zeroes required doubles or halves the length of time the computation takes; thus if the amount of computing resources dedicated to PoW is doubled or halved, the time to successfully complete the PoW and publish a new block remains about the same.

As discussed before, the miners are awarded two types of incentives, transaction fee, and a reward, for publishing the blocks. The sender has to pay transaction fees to the miners for each transaction. This fee is calculated as the difference in the sum of bitcoins in the input and the sum of bitcoins in the output. So, when a transaction is created, and the number of bitcoins that are to be transferred is specified, the sender should make sure to add the transaction fee to this number of satoshis.

$$Transaction\ fees = sum(Inputs) - sum(outputs)$$

As discussed before the size of each transaction is approximately 256 bytes. Transaction fees are calculated based on the size of the transaction in kilobytes. Currently, the transaction fee threshold is about 0.0001 bitcoin per kilobyte per transaction. Using a higher transaction fee increases the chances of this transaction to be picked by a miner. Most miners ignore proposed transactions that do not include a transaction fee, as there is no incentive for the miner to invest resources verifying such transactions.

Along with the transaction fee, a fixed number of bitcoins is given to the miner as a reward. It started with 50 bitcoin per block. The reward is reduced by half every four years. It is currently reduced to 12.5 bitcoin per block, and by 2021 it is further going to be depreciated to 6.75 bitcoin per block.

Since the sender of the transaction does not know which miner would publish his transaction, the transaction fees and the reward are delivered to the miner using a special type of transaction called the Coinbase transaction [21]. The miners create Coinbase transaction to themselves. A miner while creating a candidate block to publish, creates the Coinbase transaction. This transaction is the first transaction in a block. Figure 4.5 depicts a Coinbase transaction. This special type of transaction has only one input, which is blank. Since this transaction is not referencing any existing transaction, the transaction hash of the input is filled with 0s, and the output index is filled the maximum hexadecimal value, i.e., all fs. The ScriptSig can contain any information, as this is a black input. Just as the input, since there is only one output in a Coinbase transaction, "output counter" is 1. The output of the transaction contains the number of satoshis, received as incentives, and the Bitcoin address of the miner. As a block contains multiple transactions and as there is a dedicated transaction fee in each of these transactions, all these transaction fees added up, plus the reward amount becomes the value for the "Amount" field in the Coinbase transaction. The miner enters the Bitcoin address of the miner in ScriptPubKey script. Since Sequence Number and Locktime are not used currently they are filled with default values.

Transaction Fields	Input/Output Fields	Value (Little-Endian / Hexadecimal)
Version		01000000
Input Counter		01
Input	Transaction Hash	0000000000000000000000 0000000000000000000000 00000000000000000000
	Output Index	ff-ff-ff-ff
	ScriptSig Size	45
	ScriptSig	03ec59062f48616f4254432f53756e204368 756e2059753a205a6875616e67205975616 e2c2077696c6c20796f75206d61727279206 d653f2f06fcc9cacc19c5f278560300
	Sequence Number	ff-ff-ff-ff
Output Counter		01
Output	Amount	529c6d9800000000
	ScriptPubKey Size	019
	ScriptPubKey	76a914bfd3ebb5485b49a6cf1657824623 ead693b5a45888ac00000000
Locktime		00000000

Figure 4.5: Coinbase transaction

4.1.7 The Blockchain

Blockchain is a global and publicly available ledger. It is a data structure which is an ordered back-linked list of blocks of transactions. Every node/user is supposed to maintain a local copy of the entire Bitcoin blockchain. Each block has a reference to its previous block, which makes is back-linked. The previous block to which it refers is known as the parent block of the current block. The hash of this parent block is present in the block header, which is the back-link. It is an append-only list; new blocks can only be added. The blocks in the blockchain cannot be edited or deleted once added to the blockchain. This back-linking creates a chain going back all the way to the genesis block. When a node receives a block, it checks the previous block hash field, if it something that is already present in the copy of the blockchain it holds, then it knows it is up to date. Otherwise, that is if it has never seen this hash value, it verifies by checking the previous block hash this new last block contains, if it is same as the block hash of its last block, it adds this block into its local copy.

Every block can be identified using a unique identifier and its block height. An identifier is a unique and an unambiguous 32-byte cryptographic hash obtained by hashing the block header twice using the SHA256 algorithm. This is not created by each block to store in its data structure. Instead, this is created by a node when a block joins the blockchain and has to compute the hash of its parent block. Block height is the block's position in the blockchain, as the position cannot be altered after confirmation. The height of the genesis block is 0 and as blocks are added the numbers increase. Block height is dynamically calculated when a block is received; it is not pre-stored.

A block can have only one parent, but there can be a possibility where it has multiple children. Since Bitcoin is a big network, multiple miners might mine their proposed blocks at approximately the same time – that is, before the announcement of their proposed block propagates to all the other miners, and these blocks can choose the same parent. Hence, there can be multiple children for the same parent. This situation is called a “soft fork” and is a temporary situation. At this point, two blocks fighting for the same position might reference the same block height for a while. When publishing new blocks, miners add their blocks to what appears to them to be the longest soft-forked sub chain. Statistically, one of these sub-chains will be extended faster than the other, and all blocks from all the sibling sub-chains will be deleted and removed from the blockchain. Experience shows that a block has been safely linked into the blockchain and all of its siblings removed once it has been extended six times. A soft fork is backward compatible with existing clients. Whereas a “hard fork” is irreversible. A hard fork in Bitcoin is a radical change to the protocol that makes previously invalid blocks/transactions valid or vice-versa. This requires all nodes or users to upgrade to the latest version of the protocol software. Therefore, a hard fork is a permanent divergence from the previous version of the Bitcoin, and the newest version will no longer accept nodes running previous versions. This essentially creates a fork in the blockchain: one path follows the new, upgraded blockchain, and the other path continues along the old path. Generally, after a short period, those on the old chain will realize that their version of the blockchain is outdated or irrelevant and quickly upgrade to the latest version.

However, two blocks fighting for the same position (when a fork happens) might reference the same block height for a while. The height of the genesis block is 0 and as blocks are added the numbers increase. Block height is dynamically calculated when a block is received; it is not pre-stored.

4.1.7.1 Structure of a block

A block is a data structure which stores multiple transactions and blocks join to form the public blockchain. The maximum size of a block can be 500k bytes. Therefore, a block can contain up to roughly 2000 transactions, as the size of each transaction is approximately 256 bytes. A block contains the size of the block- which depends on the number of transactions a block has, a block header – contains important information about the block (discussed in the section 4.1.7.2), a transaction counter to indicate the number of transactions it has and finally, all the transactions. These components are shown in Fig 4.6:

Size	Field	Description
4 bytes	Block Size	Size of the block
80 bytes	Block header	Contains important block information
1-9 bytes	Transaction Counter	To indicate the number of transactions this block contains
Variable	Transaction	List of all the transactions approved by this block; the size varies based on the number of transactions a block stores

Figure 4.6: Structure of a Block

4.1.7.2 Block header

A block header contains minimum, but yet very important information to provide an idea of all the information a block contains. If a user doesn't want to use too much space on their device to store the entire blockchain, storing just the headers is an option. For example: If someone is using their smartphone to operate Bitcoin, smartphones do not provide enough space to store the blockchain. Hence just the headers can be stored, as a header takes up only 80 bytes. Along with its version, a block header is made up of three sets of block metadata, as shown in Fig 4.7. First, it contains a pointer to the hash of its previous block in the blockchain in the field “Previous block hash.” A “Merkle tree,” which is a data structure used to summarize all the transactions a block contains (discussed in section 4.1.7.3). Finally, a few details related to the mining process, “Difficulty Target,” “Timestamp” and a “Nonce.”

Size	Field	Description
4 bytes	Version	A version number to specify the formatting and parsing rules; Also, to track the protocol upgrades
32 bytes	Previous Block Hash	A reference to the hash of its parent block (previous block) in the blockchain
32 bytes	Merkle Root	A hash of all the transactions present in this block.
4 bytes	Timestamp	The approximate time when this block was created; in seconds
4 bytes	Difficulty Target	The difficulty target is that challenging value set in the PoW algorithm for this block (as explained in the section 4.1.6)
4 bytes	Nonce	This is the random number picked to solve the PoW. That is, this nonce helps the miner find a value lose to the Difficulty target. (as explained in the section 4.1.6)

Figure 4.7: Structure of a Block Header

4.1.7.3 Merkle Tree and Simplified Payment Verification

A Merkle tree is a type of binary tree data structure. In this data structure, the content of each parent node is the cryptographic hash of the parent’s child nodes. They are usually used to efficiently summarize and verify the integrity of large data sets. In

Bitcoin, Merkle trees are used to summarize all the transactions a block contains. A Merkle tree of a block produces an overall digital fingerprint of all the transactions. In Bitcoin, the algorithm used to produce the hash is double-SHA256. All transactions are nodes here and act as the leaves in this context. A Markle tree is constructed by hashing nodes recursively in pairs until only one hash is obtained, called the root. A Merkle tree can be constructed only even number of leaves. If there are an odd number of transactions, the last transaction is duplicated. It is constructed bottom-up. For example, if we have two transactions, T_a and T_b, the Merkle tree for this would be first applying the hash on these two leaves individually as shown in the equation below.

$$H_A = SHA256\big(SHA256(T_a)\big) \text{ and } H_B = SHA256\big(SHA256(T_b)\big)$$

And next concatenating these two hashes and double hashing the result as shown in the equation below. Here "+" is used to represent string concatenation.

$$H_{AB} = SHA256\big(SHA256(H_A + \ H_B)\big)$$

Similarly, this process is followed for all the n transactions in a block.

Storing the entire Bitcoin blockchain on a device to verify an incoming transaction is very burdensome. Therefore, Simplified Payment Verification (SPV) helps in verifying if a newly received transaction is included in a block using a Merkle tree. SPV loads just the latest Merkle tree of the blockchain and verifies if the new block was actually included by looking at the Merkle tree. If the transaction was actually included in the block, SPV adds this new transaction into the client's copy as it is currently not present. A user/client needs to always keep an updated copy of the blockchain or the Merkle tree because if it relies on its neighbors or has an outdated copy, the client node is prone to get compromised and might lose all the coins the client owns.

4.1.7.4 Double Spending

Double-spending means spending the bitcoins twice. For example, if a person “X” uses $5 in cash to buy a coffee at Starbucks, that cash is not owned by Starbucks. “X” cannot use the same $5 bill to make another purchase. But bitcoin is virtual cash and not physical cash. There is a possibility that bitcoins can be spent twice by its owner. To solve this, Bitcoin proposes PoW and public ledger. Once the transaction is verified and added into the blockchain, it becomes computationally impractical for an attacker to change this transaction to use the bitcoins present in this transaction again. Double-spending in Bitcoin can be possible if a group of miners try to control more than 50% of a network’s mining power or computational power. This type of attack is called the 51% attack. Miners in control of such computation power can edit the blockchain - that is add the attacker's blocks in the existing blockchain; and double-spend bitcoins. However, this situation of owning more than 51% of the mining power is unachievable under normal circumstances, i.e., under the assumption that no miner has cheated in acquiring more resources than other miners on the network to achieve the target faster.

There are multiple reasons why an attacker cannot be successful in extending his chain in comparison with the honest chain; under the same assumption as above, that all miners have balanced amount of mining resources to publish a block. Let’s consider the scenario where an attacker and is trying to generate an alternate chain faster than the honest child’s chain. The attacker has to hit the target and publish a block. As discussed earlier, on average, only one block can be published every 10 minutes. For this block to be extended, it has to traverse through the network to reach the majority of the nodes. Next, after receiving a block, the node takes few milliseconds to check the validity of this block. Therefore, the chances of multiple blocks being extended simultaneously vanish, which leaves only one winner. Along with the above reasoning, the original paper “Bitcoin: A P2P Electronic Cash System” says, “The race between the honest chain and an attacker chain can be characterized as a Binomial Random Walk. The success event is the honest chain being extended by one block, increasing its lead by +1, and the failure event is the attacker's chain being extended by one block, reducing the gap by -1”. Hence, the probability of an attacker catching up with the honest chain becomes lower as the number of confirmations increases. Since a block takes an average of 10 minutes to be mined. It becomes more challenging for an attacker chain to catch up with the honest chain because of the 10-minute interval. Two blocks can’t be mined in the 10 minutes window. Only one chain extends faster than the other. Therefore, after considering all these circumstances, it is believed by the Bitcoin community that six confirmations protect the block against double-spending. Also, a block cannot be reversed or changed after six confirmations. All of this stands true when no miner is cheating.

4.1.8 Bitcoin Network [16], [17], [22]

Bitcoin is a P2P network, which means, all nodes or the users of the protocol network are each other's peer, and there is no hierarchal topology of these nodes. All nodes are afforded equal access to all Bitcoin transactions and to the Blockchain, no node is treated special, i.e., the transactions or the request to send bitcoins from all the nodes have an equal chance of being picked and confirmed. A bitcoin network is where a group of nodes are running the Bitcoin protocol. A bitcoin node is a collection of the following four functions:

a) Network Routine Node: This node contains all the necessary routine functions to connect and participate in the network, validate, and propagate transactions.

b) Full Blockchain: The Bitcoin specification assumes that each Bitcoin node maintains a copy of the complete Blockchain. However, since each block is 512K bytes, and a block is added to the blockchain every 10 minutes, the Blockchain grows by 72 MB every day, or about 26 GB each year. Since Bitcoin is about 10 years old, this means its full Blockchain requires about 260 GB of storage today. In practice, this means few nodes can afford to maintain a copy of the entire Blockchain themselves, and so many nodes rely on others for verifying the transactions.

c) Mining Node: These nodes are called the miners. They create new blocks, add transactions to the existing or new blocks and work through the PoW algorithm. Since a copy of the entire blockchain is needed for this process, a miner node also acts as the full blockchain node by maintaining a local copy. Whenever a miner gets a block extended at least 6 times on the Blockchain, it earns a reward of some number of bitcoins from the Genesis block.

d) Wallet: These nodes are usually used with desktop or smartphone bitcoin clients. They help the user set up all the necessary preliminary components like keys, bitcoin address, etc.

It is not necessary for a node to contain all these four functions. However, A full node contains all the four functions. It requires 260+ GB of storage to store the entire Blockchain.

4.1.8.1 Network discovery

When a new node joins the network, it needs to find some neighbor(s). There are long-running stable nodes which are implicitly trusted and available when needed, called the seeds nodes. The Bitcoin client or the wallet is built with the list of the IP addresses of the seed nodes. A new node can connect to one of these by default, and its selected seeds node will communicate a list of IP addresses of other Bitcoin nodes to which it can connect via TCP; these nodes thus become the new node's neighbors. Since Bitcoin is TCP/IP base, geographical location is irrelevant in a Bitcoin network. These are various messages supported by a Bitcoin node, few of the important ones are:

a) version: This message contains the version of the bitcoin protocol the node has along with the block count or block height it currently contains. If it contains only the genesis block, then this number is 0. It exchanged while connecting to the network at the start.

b) verack: This message is sent in exchange to the above version message. If the peer node supports the version present in the version message, it sends this message to show its willingness to connect. It sends its version and the block count it contains.

c) getaddr: This message is sent by a node to its peer requesting the addresses of the nodes they know of. This message is useful for bootstrapping, i.e., let the new node catch up with all of the activity since it was last on-line. A node connects to 3-4 peer nodes at a time. If either one goes off-line, it connects to another peer suggested by its selected seeds node.

d) addr: This message is sent in response to the Getaddr. It is also broadcasted every 24 hours. It sends the IP address of one or more nodes it knows of.

e) getblock: Request is sent to get all the blocks in a range.

f) inv: This message is sent in response to Getblock request. It sends a list of blocks it contains. A maximum of 500 blocks is sent as a response.

g) getdata: This message is sent in response to inv message requesting the hash of a single block or transaction.

h) block: sends a block in response to the getdata.

i) getheader: Request is sent to get all the block headers in the range.

j) header: This response message sends up to 2000 block headers in response to getheader message. This message is useful when just the headers are needed and not the entire blocks.

k) alert: These messages are broadcasted to all the nodes in the network in case of emergencies, such as when a high priority bug that has been identified.

l) ping: this message is sent to check if a peer is still connected.

If a previously connected node disconnects, it may attempt to reconnect to its peers once it comes back on-line, providing it saves the IP addresses of its peers and its peers accept its connection request. Alternatively, a node can start fresh each time it comes back on-line. Every node sends getblock, getdata, and inv periodically to stay up to date. As a response to these requests, when a node receives a block, it checks the previous block hash field, if it something that is already present in the copy of the blockchain it holds, then it knows it is up to date. Otherwise, that is if it has never seen this hash value, it verifies by checking the previous block hash this new last block contains, if it is same as the block hash of its last block, it adds this block into its local copy.

4.2 Analysis of Bitcoin [17]–[19], [23]

4.2.1 Upgrade problems with Bitcoin protocol

1. **Fixed transaction semantics**: Bitcoin transactions are simple scripts that unconditionally transfers some number of bitcoins from a set of sellers to a set of buyers. In particular, Bitcoin supports no notion of reversible or contingent transaction. Existing Bitcoin clients and miners hardcode the semantics of transactions, so do not support the addition of new semantics without a simultaneous upgrade of all miners and clients.

2. **Fixed block size**: Block size is fixed to 500k = 2^{19} bytes. Since Bitcoin transactions are at least 200 bytes each, each block can report at most $2^{19}/_{200} = 2621$ transactions. Since Bitcoin adds a new block to the blockchain every 10 minutes = 600 seconds, this means a Bitcoin economy can support at most $2621/_{600} \approx 4.37$ transactions/second on average, worldwide. This is a long-standing problem, as it is very slow when compared to other transaction systems like VISA, which makes around 1667 transactions per second [24]. that the Bitcoin community tried to remedy in 2014 by increasing the block size from 500K to 1M. This resulted in a hard fork of the blockchain, as clients and miners that had not upgraded their software failed to recognize the new 1M blocks as valid blocks, and the changes had to be reverted. There is no way to change the current block size without incurring a hard fork.

3. **Slow transactions**: As already noted, a block is created every 10 mins, and a block is not considered as "published" until it has been extended at least 6 times. This means it takes on average at least 65 minutes to complete a transaction. This is because the Bitcoin software automatically adjusts the cost of the PoW algorithm so that, given

the resources of all miners in the network, one miner will complete the PoW every 10 minutes. If this algorithm is changed, then miners who upgrade immediately when new software becomes available will gain an advantage over miners who cannot, e.g., because they use custom hardware or software. Bitcoin was never designed for a graceful transition to a change in this algorithm.

4. **Adding new algorithms**: The current PoW consumes an enormous amount of resources and time. What if a faster or more efficient PoW algorithm was identified? Every cryptographic algorithm eventually becomes obsolete. What if a user/client wants to sign a transaction with a different signing algorithm or hash with a different hash algorithm? While discovering the nodes during the bootstrap process, since the seed nodes are hard coded into a database that is accessible to everybody, it can be modified by a malicious user to mislead the new node. If there was a new network discovery protocol that was identified, it couldn't be used until all of the old wallets and miners had been upgraded and identifying whether all the old members have been upgraded is an unsolved problem. There is no way to change these parts of the Bitcoin protocol if the need arises without introducing a hard fork in the blockchain, which would fragment the Bitcoin economy.

These problems are symptoms of a failure by the Bitcoin designers to rely on a number of features which make the transition from one software version to another more graceful. The features that do not support graceful upgrades in the current Bitcoin design are:

1. **Hardcoded**: All the elements of all the data structures are hardcoded. This kind of encoding is satisfactory and even essential for hardware but provides insufficient

flexibility in a software system, as it makes it difficult or impossible to add new or delete old features. When the data structures are hardcoded or fixed, the arrangement of fields or data are in a fixed order or fixed location in offset. This means, the data cannot be moved to a new location nor can they be made obsolete by adding new data. Whereas, if tag-length-value encoding is used, the fields are allowed to be in any order, as they are recognized by the tag and not by their location or the order of their presence.

2. **Prioritization of hardware compatibility over software flexibility:** Hardware is very efficient at accessing a particular address in memory and an offset from it. So, the fixed-length or the hardcoded data structure formats chosen for the transaction or block data structures are easy and efficient for the hardware to parse. The fixed data structures also minimize the number of bits sent over. Therefore, these kinds of data structure were useful when dial-up connections or low-speed connections were used for communication. However, this was a poor tradeoff because Bitcoin needs high-speed internet and all the basic Bitcoin functionality, except for PoW, are implemented as application-level software, where flexibility is almost always more important than efficiency per se.

3. **Doesn’t support tag-length-value encoding**: Since all the fields of transactions, blocks, blockchain, etc. are hardcoded, neither the syntax nor semantics of the Bitcoin protocol can be changed without upgrading every client and miner simultaneously. We could achieve greater interoperability by using tag-length-value encoding since that would allow the implementation to skip over the fields that they do not recognize instead of rejecting the entire message, as Bitcoin will have to do with the original design.

4. **No significance of Version field**: Though the protocol defines a version number, describing the formatting and parsing rules, it doesn't support upgrades. If a new version of the protocol was to be defined, it should contain all the old features – which is possible to obtain in this protocol and allow the addition of new fields – which is not possible, because Bitcoin has no rules for how old clients and miners deal with unknown fields and messages. The new version should be able to define its own formatting and parsing rules. This is not allowed because the data structures cannot be changed. Therefore, the version number present in this current Bitcoin protocol doesn't support upgrades. This violates the first requirement in the hypothesis, which is to have a version number for each variant of the software. Numerous formatting and parsing rules can be supported by implementing version numbers in the design.

5. **No reserved values**: Not a single data structure in the entire Bitcoin protocol has reserved values saved for future use. This means no new features can be added. The future needs were not thought while designing this protocol. Having reserved values not only allows new values or fields to be added but also offers standardization of the protocol and avoid ambiguity, i.e., if a value is reserved by a version, v1, other versions are not allowed to change it and are compelled to follow the value established by v1. Having reserved values can be either reserving a version number or reserving a tag in tag-length-value to preserve unambiguity.

6. **List view unavailable**: Firstly, there is no scope of adding multiple items for a single field, i.e., multiple signature algorithms cannot be provided for the user to choose from. For instance, some version migrating from an old signature algorithm to a new one must necessarily include the signatures and public keys of both the old and the new

signature algorithms, or else old wallets and miners will be unable to verify the transactions secured by the new signature algorithm. Even if there was a way to upgrade and support other methodologies, there is no opportunity to provide multiple options to the users. For example, if protocol version 2 has its own new features, and it also supports version 1, protocol 2 cannot display both the features of version 1 and version2. A version 2 protocol cannot tell its peers that it supports version 1 as well. This might allow version 1 users to assume that version 2 does not support version. Having a list view is useful for nodes supporting multiple versions to agree upon a common version.

4.3 Engineering a protocol to demonstrate graceful upgrades

As discussed before, it is recommended to include a few necessary features while constructing any program in a distributed system. Prof. Jesse Walker and I developed a C code that demonstrates graceful upgrades among three different version of this software codebase using the features discussed in the hypothesis of this study. This code demonstrates that different versions of the software should be able to communicate with other versions, except when a conscious decision is made to no longer maintain backward compatibility with any earlier version. The protocol/code works as a request-response message between a client and a server. The protocol is designed such that, version 1 and version 2 protocol communicate and parse the message received from each other, version 2 and version 3 communicate and parse the message received from each other but version 1 and version 3 do not parse each other's messages. The initial database contains information about the customers, like, customer’s ID, name, address, city, and state. The client initiates the protocol by sending a request message to retrieve customer information based on a unique identifier, which is customer’s ID in this case. Next, in version 2, the name field is split into first name and last name. To support interoperability, the version 2 database contains name field along with first name and last name fields. Finally, version 3 maintains interoperability with version 2 but breaks interoperability with version 1, by including first name and last name fields and deleting the name field in its database. Therefore, based on the version, the server decides whether it can accept this request. If it can respond, the server locates the customer record by customer id from its customer database, formats the response, and returns it to the client.

Note that a version 1 server cannot necessarily respond to a version 3 client request, because we made the design decision that version 3 of the protocol is not backward compatible with version 1. It would be possible for a version 3 server to respond by formatting and returning a version 2 message, which a version 1 client can properly interpret, in response to a version 1 request, but our implementation did not do this.

We use version numbers to determine which formatting and parsing rules to apply to each version. The tag-length-format allows two versions to communicate with each other even when that version doesn’t recognize few fields. We made the design decision that an older version simply skips fields it cannot recognize with the help of the length field, as the length field indicates the amount of data to skip to reach the next field. The entire message is also formatted using tag-length-value format, with the message type being a tag, and the overall message length, of all the encoded tag-length-value fields combined. In addition to the message type and length, the message header also includes the protocol version number.

Our implementation can be found in 4 files that help in carrying out all the necessary operations: customer.c, demo.c, protocol_msg.c and msg_buffer.c.

4.3.1 Description of initial protocol code

The customer.c file defines an abstract data type, so manages all the operations that are carried out on customer information, including serialization and deserialization of the customer record fields. The initial version contains these customer record includes the customer's name, address, city, state, and customer id. Each customer record is identified by a unique identifier, cid, which is a 32-bit unsigned integer.

To facilitate all the necessary operations, allocation of a buffer as desired by each request is done in the file msg_buffer.c. This file defines the minimum, default and maximum size of the buffer that can be allocated. A default buffer size is set in the beginning. If the length of the message received is greater than the allocated default buffer size, the buffer size is set to maximum buffer size. Finally, based on this size, a buffer is allocated to store all the customer information, to conduct serialization and deserialization, and to display to customer information. This file also manages the functions used to clear and reset the buffer after its use and to skip the unrecognized tags. The way this works is when the message parser encounters an unknown tag, it deserializes (decodes) the length field and then skips that many bytes ahead to the next tag. This allows an older version to skip fields in messages from newer version peers that the older does not recognize, thus continuing interoperability across version numbers. If this was an undesired behavior for an application, we could have included a criticality flag in the tag-length-value formatting and parsing routines. In that case, the receiver's parser would abandon the message upon receiving a message with unknown critical fields.

As mentioned before, this protocol uses a tag-length-value format to represent structured data when serialized into a byte stream for communication across a network. The tags are hierarchical, and the tag values are unique for that level. In the sample code, there are two levels of tag hierarchy: message type tags and field tags. In principle, there could be more tag hierarchy if there were a need for structures within structures, but it is not required in this experiment. There are two types of messages used in this code – request message and response message. The file protocol_msg.c assigns tags to the request and response messages, assigns tags to the data fields, handles illegal tags and generates the random 64-bit unsigned integer used as xid. At the first level we have the header of a request message containing message id - which is the message tags to informs if it is a request message or a response message, xid and the data. The xid is a random 64-bit unsigned integer, serialized as an 8-byte little-Endian value. The xid is used to match the reply to the request or to know which reply message is for which request message. Little-Endian serialization stores the binary representation of any multibyte data types like int, float, etc. It stores the last byte or the least significant of the binary representation first and the first byte or the most significant byte at last. There are two main reasons why Little-Endian serialization was is generally used. First, it improves performance by achieving parallelism. As the least significant byte is communicated first, the calculation of the xid starts without having to wait for the rest of the bytes while the rest of the bytes load from memory. The second reason is that most of the systems use Little-Endian to communicate with each other. The second level defines the tags for message fields.

When a message is received, it needs to be decoded. The decoding functions utilize the deserialization functions exported by the Customer abstract data type, using the Message Buffer abstraction to access the serialized data. The code must make sure that any field occurs only once in each valid message, and that all the fields needed to populate a data structure are present and occur in the order specified by the protocol. If a field is present more than once, or if fields occur in an order different than in the protocol specification, the entire message is discarded.

Finally, the exchange of request and response messages between two peers can be demonstrated or simulated using the file demo.c. The functions in this file initiate the finding of customer details by the cid received in a request message, generates the random cid, creates a structure to initiate the response and request messages. This file is also prepared to deal with storing and deallocating the customer information in a database. To summarize, the initialization of the protocol happens in this file.

4.3.2 Upgrading from version 1 to version 2

Version 2 of this protocol has a first name and last name fields in addition to the version 1 name. Therefore, for a version 2 protocol, the fields, first_name, last_name, address, city and state fields are sufficient to communicate with the same version protocol, but version 2 maintains the name field for backward compatibility with version 1. In particular, the version 1 client parser expects the name field to be present, so to achieve backward compatibility version 2 must include the name field as well as the new first and last name fields in its messages. To get started version 1 protocol code is cloned. Most of the parts of the code in version 1 are maintained and reused in version 2. In the file customer.c of version 1, 2 new functions are added for each of these operations to accommodate the requirements of version 2 protocol. For the message that was received from version 1, containing the name field, there is a function that splits name into first name and last name based on the occurrence of the first space in it. Second, when a version 2 message is to be sent to version 1, it combines first_name and last_name to compose the name field.

In protocol_msg.c of version 1, these new functions are added – serialize_first_name, serialize_last_name, encode_first_name, encode_last_name, deserialize_first_name, deserialize_last_name, decode_first_name and decode_last_name. The functionalities of these functions are straightforward as the name suggests. To facilitate the checks in protocol_msg.c file, the fields first_name_count and last_name_count are added.

Along with these new functions, two new important functions are added to enable communication between version 1 and version 2. The function customer_name_merge_encode is used in the encoding process. This function constructs a version 1 name field from the first and last name, as required to be backward compatible with version 1 clients. The second function that is added is customer_name_split_decode. This function is used during decoding a message that is received from a version 1 protocol, to split the name field from a version 1 server into the first and last names required by the version 2 Customer record. While parsing the name field, the customer_name_split_decode function checks if the value in the name field contains a space. The value in the field “name” is split into first_name and last_name at the first occurrence of a space, which is the behavior specified in the functional requirements for the coding exercise. If there is no space, then the value in the field “name” is placed into first_name, and last_name is filled with a NULL string.

Though new functions and fields are added to the files, the functionality and working of the protocol remain the same as version 1. These fields and functions are added by borrowing the code from version 1. The semantics are preserved while adding any new piece of code. Though protocol version 2 doesn’t need the functionalities related to the field “name,” it is preserved to maintain unbroken communication between version 1 and version 2 protocols.

4.3.3 Upgrading from version 2 to version 3

Version 3 no longer supports the version 1 name field. Hence version 3 simply had to remove the name field from its processing. If a version 3 server tried to communicate with version 1 client, the message sent would lack a name field and so be rejected by the client. However, version 3 and version 2 can exchange information through messages because version 3 still supports first_name and last_name fields.

To conclude, this coding project successfully achieved interoperability between its versions by following the principles proposed in the hypothesis of this study.

4.4 Lessons Learned

It is not enough to just propose ideas. Ideas have added value when they are proved correct by some means like implementing the ideas in a real protocol. At the start of this experiment, I was very uncomfortable with coding. The last time I coded a few projects on my own was during my undergraduate days. Though I did code some basic programs in C, C++, JAVA, few other languages and a term project in C#, after a huge gap of 4 years, I was not ready to take up this magnitude of responsibility in coding.

The difference between other term projects during my master's program and this was that I owned this experiment, I had to figure out a lot of things and also learn new concepts on my own without any teammate's help. This was going to be my master's research project, which is a very big thing for me. One of the languages I was comparatively comfortable with was Python. Hence, not to be overwhelmed, I started the experiment by writing code in Python. I was successful in completing version 1 and version 2. However, one of the motives of this experiment was to get familiarized with the Object-Oriented Programming (OOP) concepts.

At the beginning of this project, I was very naïve at C programming and of course, at the concept of pointers. Pointers have always been something that frightened me as they do to most programmers. This coding project has helped me analyze a code base comprehensively and make the required changes or additions while following the semantics.

I also gained knowledge on using few code editors like Visual Studio Code, Atom; along with Code Blacks, which is an Integrated Development Environment (IDE). For this experiment, Visual studio Code was used to understand and make necessary changes in the code and Cygwin terminal was used to compile, debug and run the code. The features like outline and search (Ctrl+Shift+F) made it easy to understand the connectivity between functions in the code. After this exercise I can confidently say that I have gained adequate command on C programming, mainly OOPS concepts and I feel I am ready to make my place in the software industry.

5 Conclusion

We know the ever-changing needs of the users of distributed software systems imply they will be upgraded and have an unpredictable future. Therefore, we need to plan and design the software enabling the software to evolve to address currently unknown scenarios while still interoperating with earlier versions of the system. Since there are no guidelines established that would make the community aware of the important factors, forming these guidelines that are to be followed.

Studying a few successful and widely used distributed software, such as BOOTP, 802.11i (Wi-Fi) and X.509 identified design properties needed to enable successful upgrades. These properties facilitate smooth and successful upgrades in applications. The requirements that were discovered important in this research are:

a) Always include a version number in protocol messages. This permits a receiving implementation to apply the right semantics to any messages it receives, or even ignore entire messages whose version it does not recognize.

b) Add new features to the already existing ones without deleting the old features, i.e., preserve existing semantics. This allows implementations of newer versions to process messages sent by older version implementations that have not yet been upgraded.

c) Use a tag-length-value format whenever possible. This allows older implementations to skip fields in message versions they do not support, meaning they do not necessarily need to ignore entire messages.

d) Reserve unused version numbers and tag values for future use, which means the range of allowed tag values is explicitly defined for each version. This means new versions can allocate reserved values without fear of the values being misinterpreted.

e) Make the fields critical whenever it does not make sense for old versions to process data structures and messages containing what is to them an unknown field.

f) Define a way to negotiate the version number, so different implementations can decide on the formatting and parsing rules for data structures they exchange.

Bitcoin does not support graceful updates. This was proved when the Bitcoin community attempted to increase the size of blocks in the Bitcoin blockchain: earlier versions of the software did not accept blocks issued by miners upgraded to use the larger block size, causing hard fork in the blockchain, and the upgraded software had to be rolled back to the old block size. When this was done, the users could use the coins in this newer version that they owned in the previous version. The users had to sell their coins and buy new ones in the latest version. We can see that even though there was version number in the bitcoin code, the upgrade was not possible. Therefore, avoiding these guidelines would result in rebuilding the entire software, mostly every time changes are to be made, and as Bitcoin shows, can destroy user data.

A coding experiment was conducted to illustrate these recommended guidelines. This experiment started with a simple client/server protocol, and fields were added and removed to messages in subsequent versions. In my experiment, we exercised most of these requirements: version number was included, tag-length-value encoding was followed, new fields were added on top of the old fields, tags and versions were reserved, and the semantics of old data was preserved in each upgrade. This experiment successfully demonstrated, communication among three different versions of the software. Each version knew how to read the messages from the other two versions.

This research area has many more aspects yet to be explored. In the future, perhaps more of these requirements can be identified which could not be found in this research due to lack of time. Further, it might be possible to find novel ways to upgrade software that was never designed to make this easy, perhaps using techniques similar to how DHCP repurposes BOOTP messages and infrastructure. How to accomplish this in a complex distributed system like Bitcoin remains an open problem.

6 References

[1] A. Hertig, “In Big Block Hard Fork, Craig Wright’s Bitcoin Has Left Nodes Behind,” *CoinDesk*, 10-Aug-2019. .

[2] J. Gilmore and W. J. Croft, “Bootstrap Protocol.” [Online]. Available: https://tools.ietf.org/html/rfc951. [Accessed: 21-Aug-2019].

[3] D. Cooper, “Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.” [Online]. Available: https://tools.ietf.org/html/rfc5280. [Accessed: 21-Aug-2019].

[4] R. Droms, “Dynamic Host Configuration Protocol.” [Online]. Available: https://tools.ietf.org/html/rfc2131. [Accessed: 21-Aug-2019].

[5] “Understanding Online Certificate Status Protocol and Certificate Revocation Lists - TechLibrary - Juniper Networks.” [Online]. Available: https://www.juniper.net/documentation/en_US/junos/topics/concept/certificate-ocsp-understanding.html. [Accessed: 21-Aug-2019].

[6] J. Sermersheim <jimse@novell.com>, “Lightweight Directory Access Protocol (LDAP): The Protocol.” [Online]. Available: https://tools.ietf.org/html/rfc4511. [Accessed: 21-Aug-2019].

[7] “A Layman’s Guide to a Subset of ASN.1, BER, and DER.” [Online]. Available: http://luca.ntop.org/Teaching/Appunti/asn1.html. [Accessed: 21-Aug-2019].

[8] L. Sassaman, M. L. Patterson, S. Bratus, and M. E. Locasto, “Security Applications of Formal Language Theory,” *IEEE Syst. J.*, vol. 7, no. 3, pp. 489–500, Sep. 2013.

[9] IEEE Computer Society, “IEEE Standard for Information Technology - Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Network Specific Requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.”

[10] J. R. Vollbrecht, B. Aboba, L. J. Blunk, H. Levkowetz, and J. Carlson, “Extensible Authentication Protocol (EAP).” [Online]. Available: https://tools.ietf.org/html/rfc3748. [Accessed: 21-Aug-2019].

[11] W. Simpson, “The Point-to-Point Protocol (PPP).” [Online]. Available: https://tools.ietf.org/html/rfc1661. [Accessed: 21-Aug-2019].

[12] S. Fluhrer, I. Mantin, and A. Shamir, “Weaknesses in the Key Scheduling Algorithm of RC4,” in *Selected Areas in Cryptography*, 2001, pp. 1–24.

[13] R. Droms, “Interoperation Between DHCP and BOOTP.” [Online]. Available: https://tools.ietf.org/html/rfc1534. [Accessed: 21-Aug-2019].

[14] *Bitcoin Improvement Proposals. Contribute to bitcoin/bips development by creating an account on GitHub*. Bitcoin, 2019.

[15] “China, U.S. strike trade accord | Tech News on ZDNet,” 07-Apr-2005. [Online]. Available: https://web.archive.org/web/20050407222112/http://news.zdnet.com/2100-9584_22-5197087.html. [Accessed: 21-Aug-2019].

[16] “Protocol rules - Bitcoin Wiki.” [Online]. Available: https://en.bitcoin.it/wiki/Protocol_rules. [Accessed: 21-Aug-2019].

[17] A. M. Antonopoulos, *Mastering bitcoin*, First edition. Sebastopol CA: O’Reilly, 2015.

[18] S. Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System,” p. 9.

[19] “Bitcoin - Open source P2P money.” [Online]. Available: https://bitcoin.org/en/. [Accessed: 21-Aug-2019].

[20] “Script - Bitcoin Wiki.” [Online]. Available: https://en.bitcoin.it/wiki/Script. [Accessed: 04-Sep-2019].

[21] “Coinbase Transaction.” [Online]. Available: https://learnmeabitcoin.com/glossary/coinbase-transaction. [Accessed: 21-Aug-2019].

[22] “Satoshi Client Node Discovery - Bitcoin Wiki.” [Online]. Available: https://en.bitcoin.it/wiki/Satoshi_Client_Node_Discovery#See_Also. [Accessed: 21-Aug-2019].

[23] “How Does Bitcoin Work?” [Online]. Available: https://learnmeabitcoin.com/. [Accessed: 21-Aug-2019].

[24] J. Vermeulen, “Bitcoin and Ethereum vs Visa and PayPal – Transactions per second.” .

7 APPENDICES

7.1 Appendix A: C Source Code

1. Google Drive Link:

   https://drive.google.com/drive/u/3/folders/1_9fiso3CG7d60kJD5tZlKkqaii3Uzc16