Transport-Layer Security

Outline

- TCP/UDP Protocols
- TLS Security Protocol
- DTLS Security Protocol
- QUIC Security Protocol

TCP/UDP

The protocol where two applications can talk to each other

TCP/UDP: Story and Motivations (1/4)

Imagine two countries, Country A and Country B, separated by a single path that only allows one postman to deliver parcels at a time. Each parcel cannot be larger than 1x1x1 meter. In Country A, there is a person named Alice, who wants to send items to Bob, who lives in Country B.

- How can the parcel be delivered correctly between any two persons in Country A and B, respectively?
- What happens if the item is too big and exceeds the parcel size limitation?

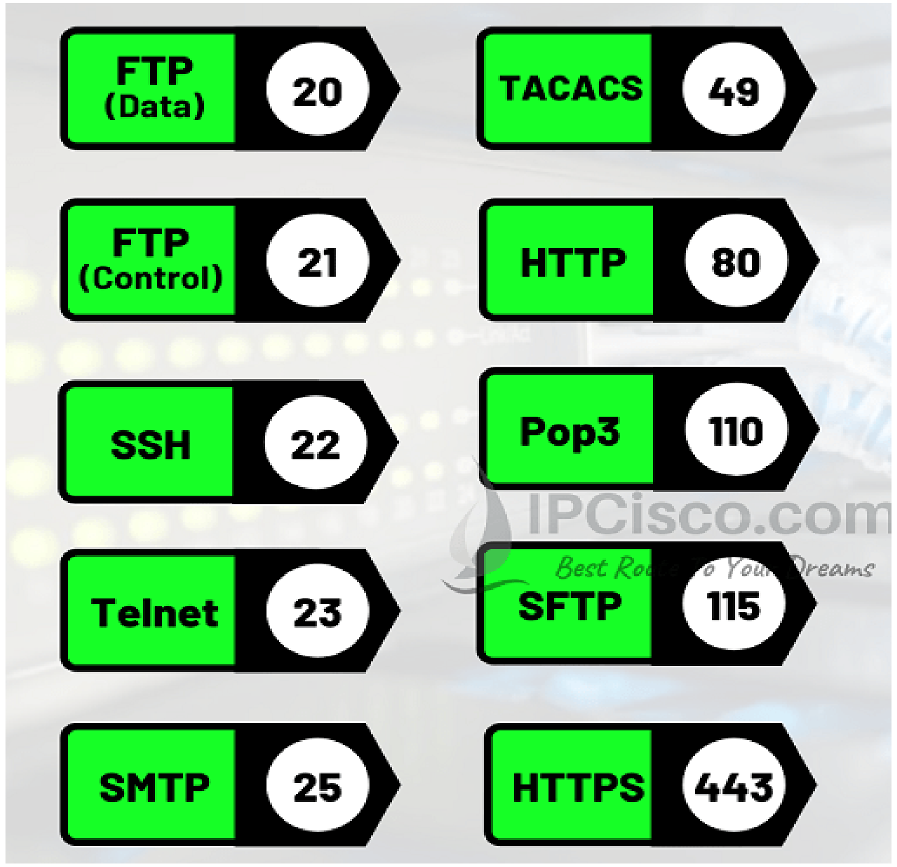
TCP/UDP: Story and Motivations (2/4)

Imagine two computers, Computer A and Computer B, connected by a single data cable that only allows one data packet to be sent at a time. Each data packet cannot be larger than 1K bytes, representing the maximum size allowed for each transmission.

On Computer A, there are multiple applications, just like people wanting to send messages. One application wants to send data to a corresponding application on Computer B. Now, two key questions arise:

- How can the data packet be delivered correctly between any two specific applications on the respective computers?
- What happens if the data packet is too big and exceeds the size limitation?

TCP/UDP: Story and Motivations (3/4)



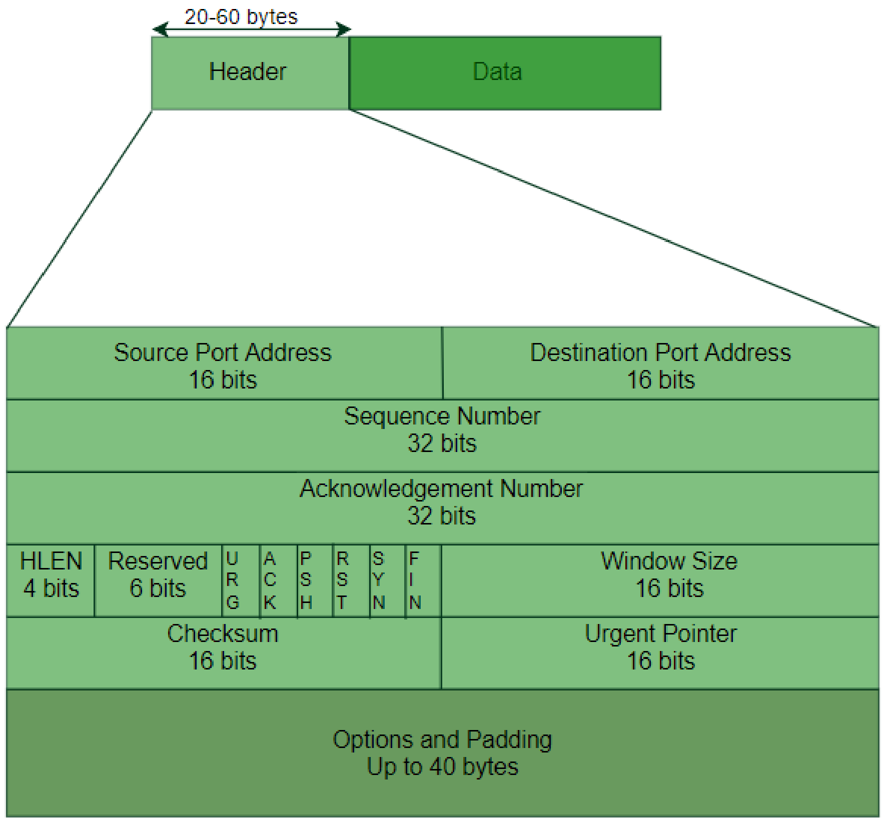
(Fuchun: No need to remember these numbers)

The solution to delivering data packets correctly between specific applications on two computers lies in using port numbers. Each computer has multiple applications running, and each application is assigned a unique port number, which acts as an address within the computer.

When data is sent:

1. The data packet includes the destination computer and the port number of the specific application.
2. Upon reaching the destination computer, the system looks at the port number in the packet to determine which application should receive the data.

TCP/UDP: Story and Motivations (4/4)



Segments of TCP Protocol

What happens if the data packet is too big and exceeds the size limitation?

- In network communication, if a data packet exceeds the maximum size allowed by the network (known as the Maximum Transmission Unit or MTU), it is broken down into smaller chunks called segments. Each segment is sent individually, and the receiving computer reassembles these segments into the original data. This process ensures that even large amounts of data can be sent efficiently over the network, despite size limitations.

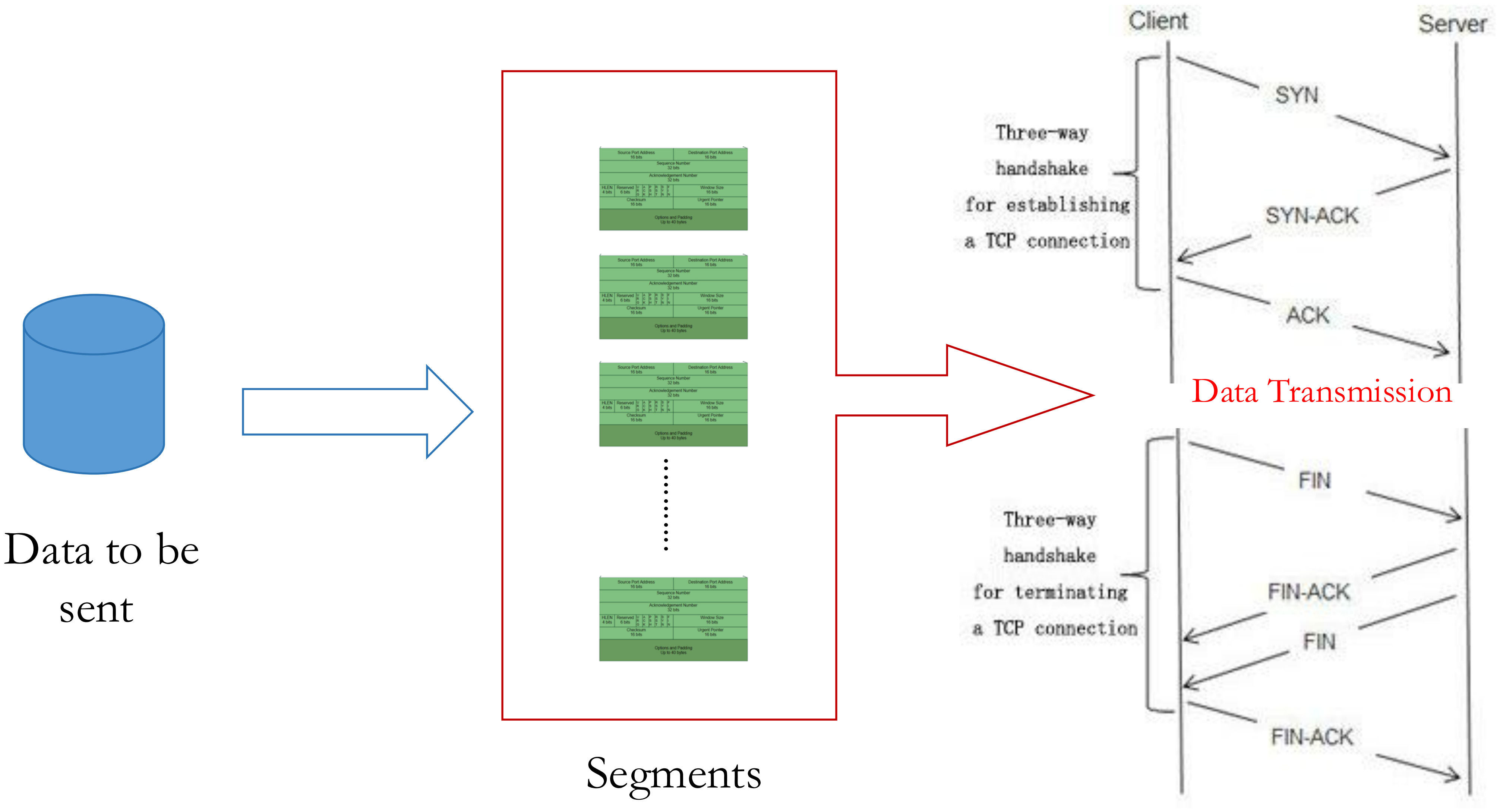
TCP/UDP: TCP (1/6)

TCP (Transmission Control Protocol) is a reliable, connection-oriented protocol used for data transmission over networks. It ensures that data is delivered accurately and in the correct order between two devices.

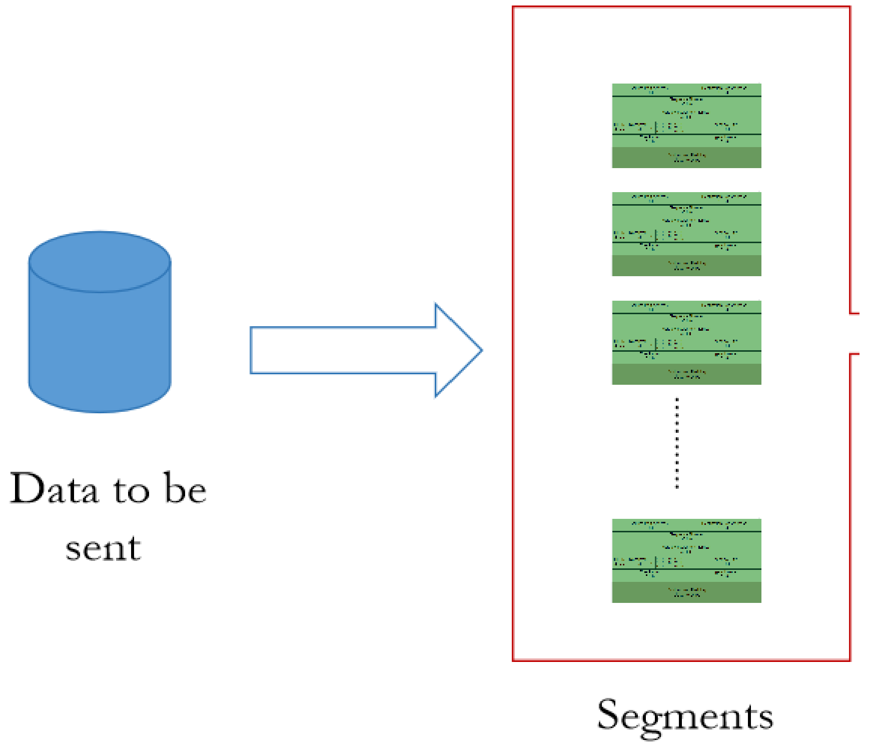
Key features of TCP include:

- Reliable Delivery: TCP ensures that data is received without errors, and if any packet is lost, it retransmits the missing packets.
- Orderly Delivery: Packets are sent and reassembled in the same order they were transmitted.
- Connection Establishment: Before data transmission, TCP establishes a connection through a process called the three-way handshake.
- Flow Control: TCP regulates the rate of data transmission to avoid overwhelming the network.

TCP/UDP: TCP (2/6)


Data to be sent
Segments
Client
Server
Three-way handshake for establishing a TCP connection
SYN
SYN-ACK
ACK
Data Transmission
Three-way handshake for terminating a TCP connection
FIN
FIN-ACK
FIN
FIN-ACK

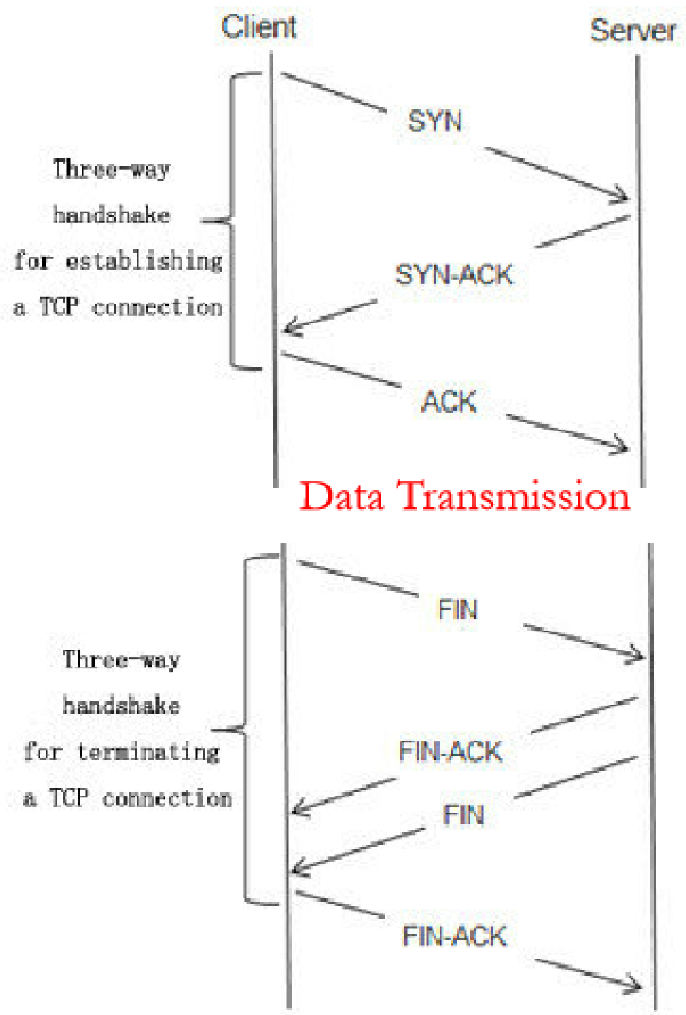
TCP/UDP: TCP (3/6)



When data is broken down into segments, it's essential to consider

- how the receiver can reassemble the segments correctly, and
- how to handle any potential data errors.

TCP/UDP: TCP (4/6)

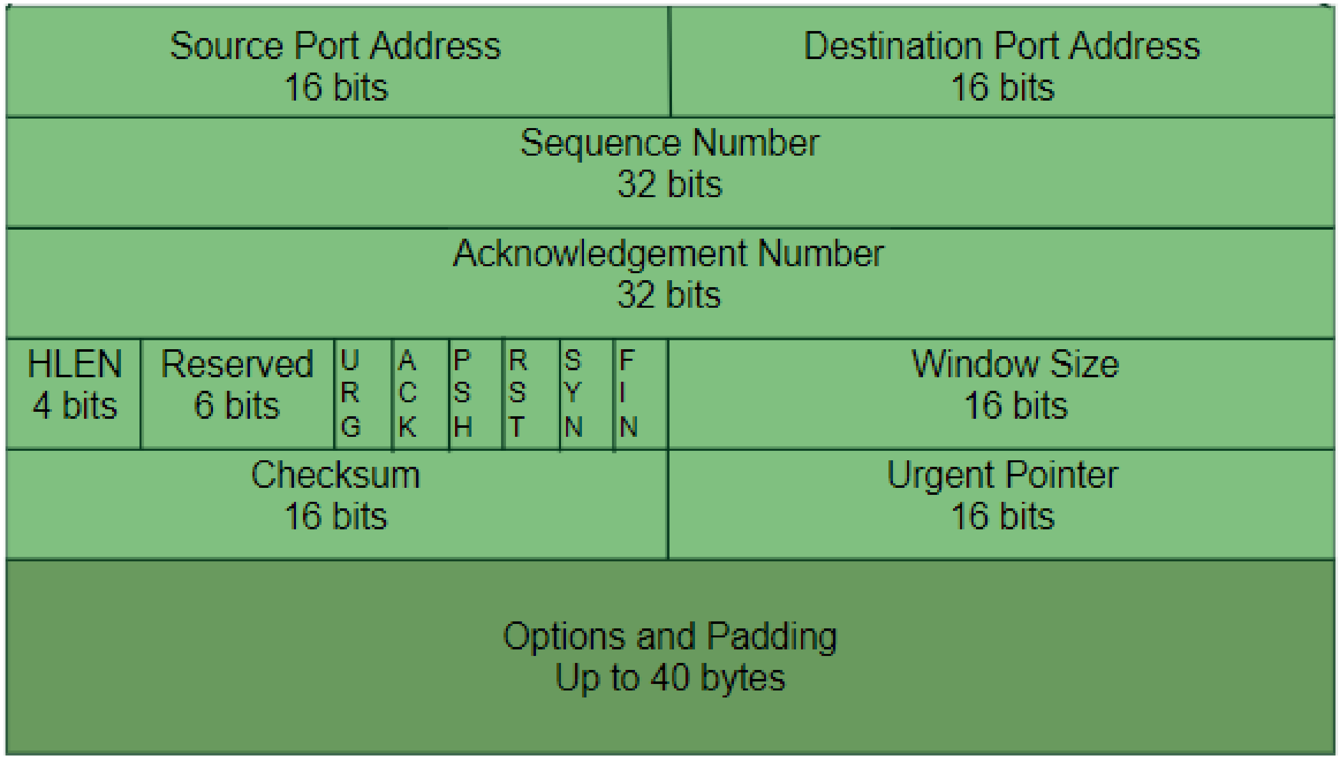

Client
Server
SYN
SYN-ACK
ACK
Three-way handshake for establishing a TCP connection
Data Transmission
FIN
FIN-ACK
FIN
FIN-ACK
Three-way handshake for terminating a TCP connection

1. Handshake for Establishing a Connection: The handshake process establishes a reliable connection between the sender and receiver before any data transmission occurs. This ensures that both parties are ready for communication and agree on the parameters of the session

2. Data Flow: This component manages the reliable transfer of data between the sender and receiver after the connection is established. It ensures that the data is sent, received, and acknowledged in an orderly manner.

3. Termination: This component cleanly ends the TCP connection between the sender and receiver, ensuring that all data has been transmitted and acknowledged before closing the connection.

TCP/UDP: TCP (5/6)



Sequence Number (32 bits): This field indicates the sequence number of the first byte of data in this segment (or the index of segment).

Acknowledgment Number (32 bits): This field indicates the next expected byte from the sender. It tells the sender that all bytes up to (but not including) this number have been successfully received by the receiver.

Flags (6 bits): Includes control flags such as SYN, ACK, FIN, etc. Each flag serves a specific communication purpose in the communication process

TCP/UDP: TCP (6/6)

Key features of TCP include:

- Reliable Delivery: It uses mechanisms like checksums to detect errors in the transmitted segments and uses acknowledgments (ACKs) to confirm the successful receipt of segments.

- Orderly Delivery: Each segment is assigned a sequence number, allowing the receiver to order the segments properly.

- Connection Establishment: It use a three-way handshake process (SYN, SYN-ACK, ACK) before data transmission begins. This process ensures that both the sender and receiver are ready for communication and that they can synchronize their sequence numbers.

- Flow Control: It employs flow control mechanisms (specifically the sliding window protocol, called window size in header) to manage the rate of data transmission .

TCP/UDP: UDP (1/4)

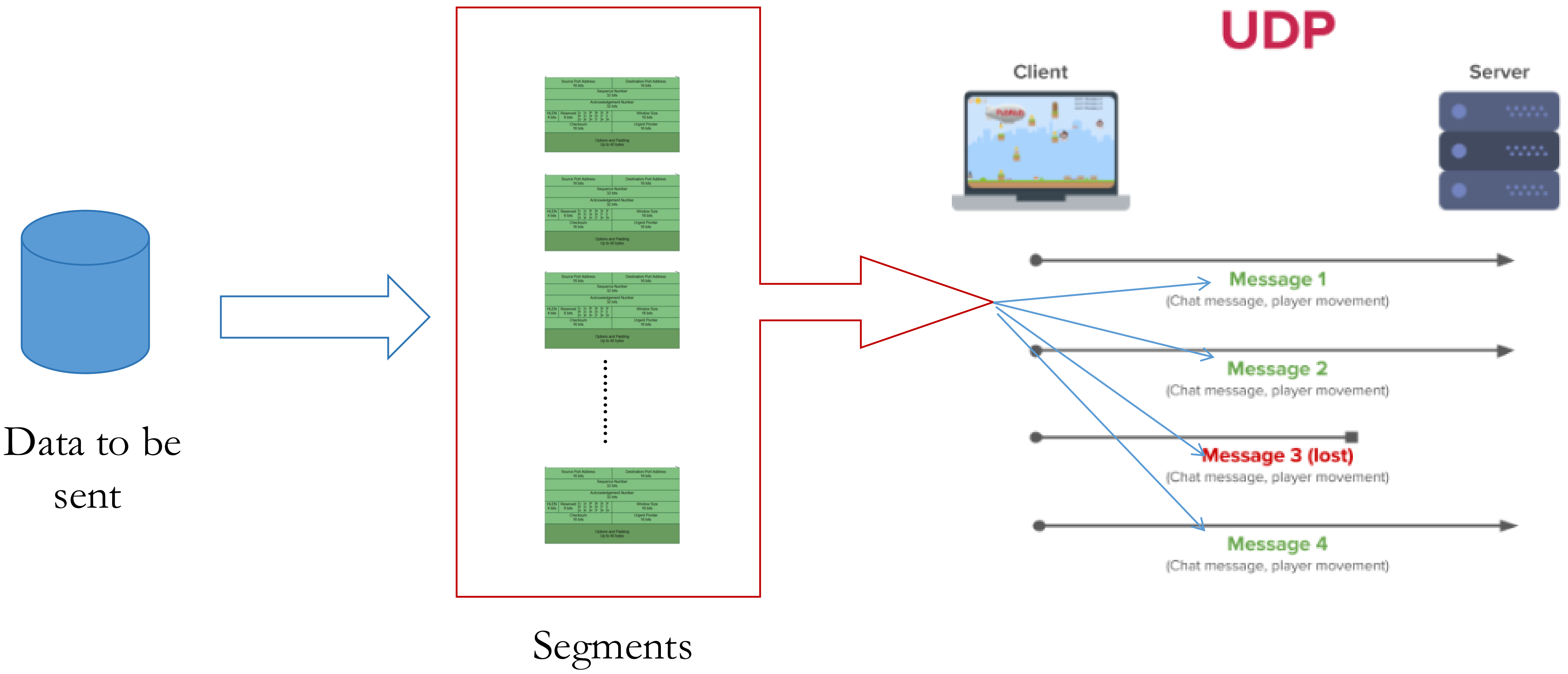
UDP (User Datagram Protocol) is a simple, connectionless protocol used for fast data transmission over networks. It prioritizes speed and efficiency, making it ideal for real-time applications where delays are more critical than occasional data loss, such as live streaming and online gaming.

Key benefits of UDP include:

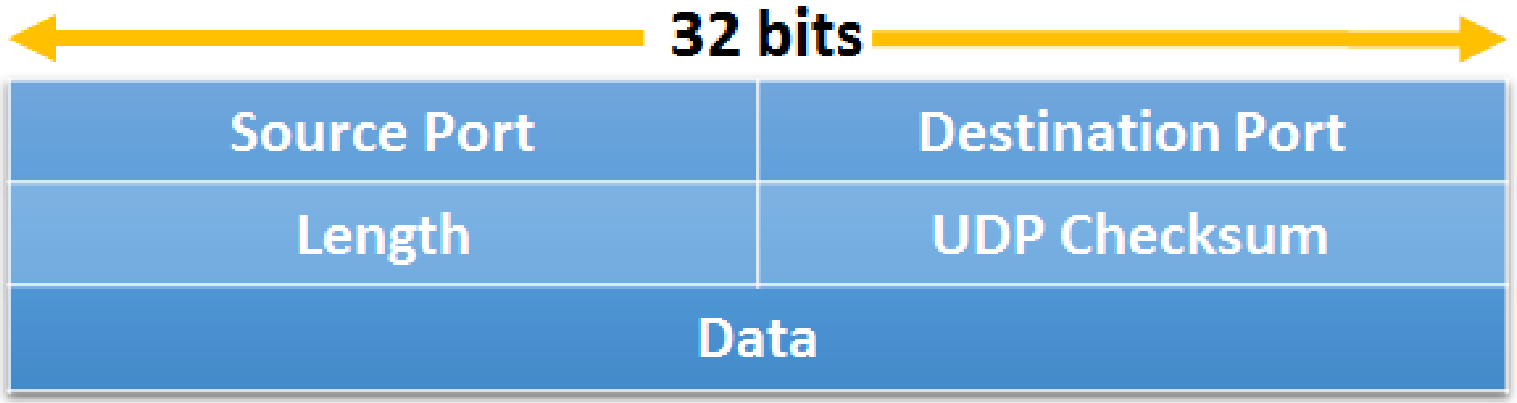
- Speed and Efficiency: UDP provides low latency and fast data transmission. This makes it ideal for real-time applications like online gaming and video streaming.

- Broadcast and Multicast Support: UDP enables efficient broadcasting and multicasting, allowing a single packet to be sent to multiple recipients simultaneously. This is beneficial for applications that need to deliver data to many users at once, such as live streaming or online conferences.

TCP/UDP: UDP (2/4)


Data to be sent
Segments
UDP
Client
Server
Message 1
(Chat message, player movement)
Message 2
(Chat message, player movement)
Message 3 (lost)
(Chat message, player movement)
Message 4
(Chat message, player movement)

TCP/UDP: UDP (3/4)



The UDP segment structure is simple compared to TCP. It consists of four fields in its header， and then payload:

- Source Port (16 bits): The port number of the sender, identifying the sending application.
- Destination Port (16 bits): The port number of the receiver, identifying the receiving application.
- Length (16 bits): The total length of the UDP segment, including the header and data.
- Checksum (16 bits): Used for error-checking (NO recovery) the header and data, ensuring data integrity.

After the header comes the data (payload), which contains the actual information being sent. UDP's simplicity makes it faster and more efficient but less reliable than TCP.

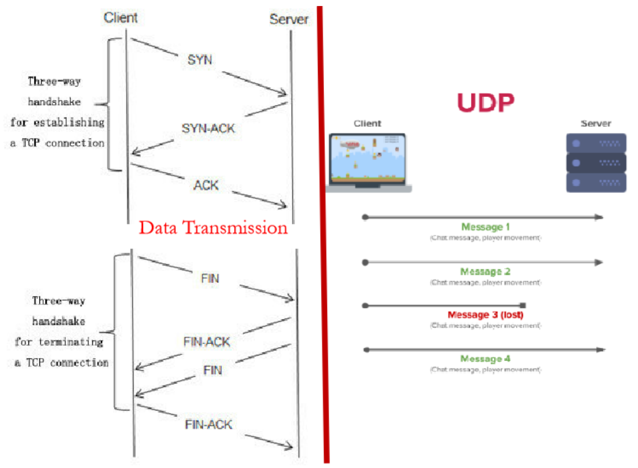
TCP/UDP: UDP (4/4)

Key benefits of UDP:

- Speed and Efficiency: UDP provides low latency and fast data transmission by eliminating overhead associated with connection establishment, acknowledgments, and retransmissions. This makes it ideal for real-time applications like online gaming and video streaming.

- Broadcast and Multicast Support: UDP enables efficient broadcasting and multicasting because it lacks the need for connection establishment, ACKs, and sequence numbers. Its simple segment structure, with minimal header overhead, allows a single packet to be sent to multiple recipients without the complexity of managing individual connections or ensuring reliable delivery, making it ideal for use cases like live streaming or online conferencing where speed and scalability are prioritized over reliability.

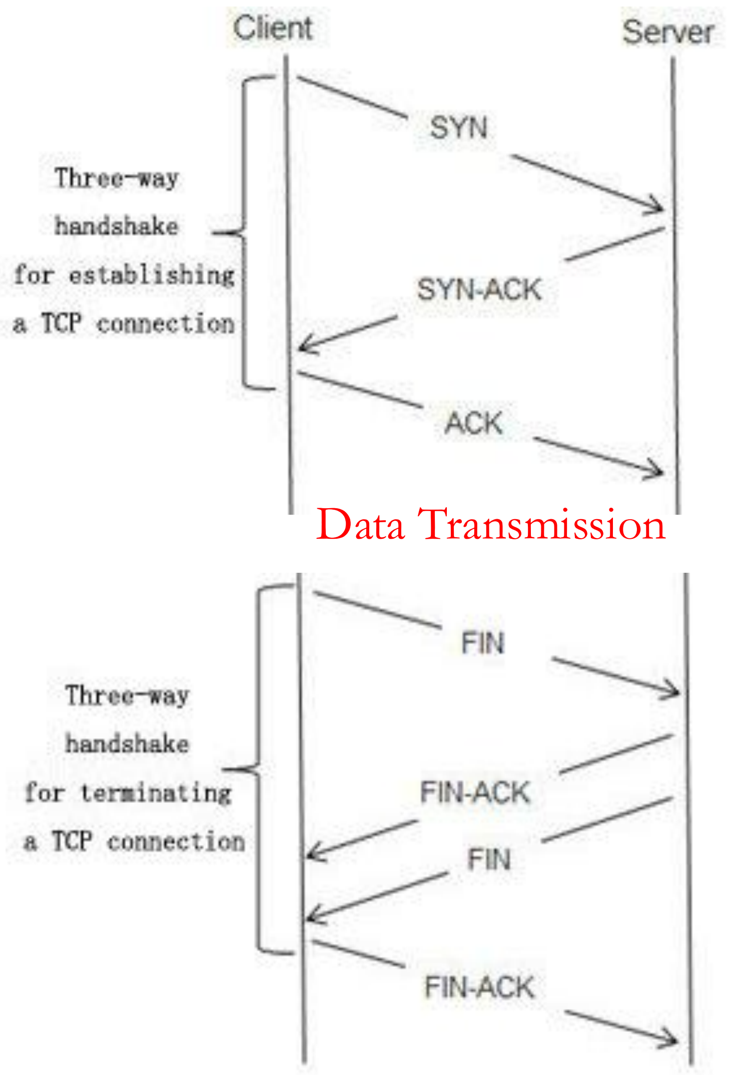
TCP/UDP: Comparison


Client
Server
Three-way handshake for establishing a TCP connection
SYN
SYN-ACK
ACK
Data Transmission
Three-way handshake for terminating a TCP connection
FIN
FIN-ACK
FIN
FIN-ACK
UDP
Client
Server
Message 1
(Chat message, player movement)
Message 2
(Chat message, player movement)
Message 3 (lost)
(Chat message, player movement)
Message 4
(Chat message, player movement)

- TCP (Transmission Control Protocol) is connection-oriented and ensures reliable data delivery. It uses mechanisms like error-checking, acknowledgments (ACKs), sequence numbers, and flow control to guarantee that data is transmitted accurately and in order. However, these features introduce additional overhead, making TCP slower but reliable, which is ideal for applications like file transfers and web browsing.

- UDP (User Datagram Protocol) is connectionless and focuses on fast, lightweight communication. It does not provide reliability, ordering, or error-checking **beyond a basic checksum(no re-transmission due to error)**, making it faster but less reliable than TCP. UDP is well-suited for applications like streaming, gaming, or any situation where speed is more important than data integrity.

TCP/UDP: TCP SYN Flooding (1/3)

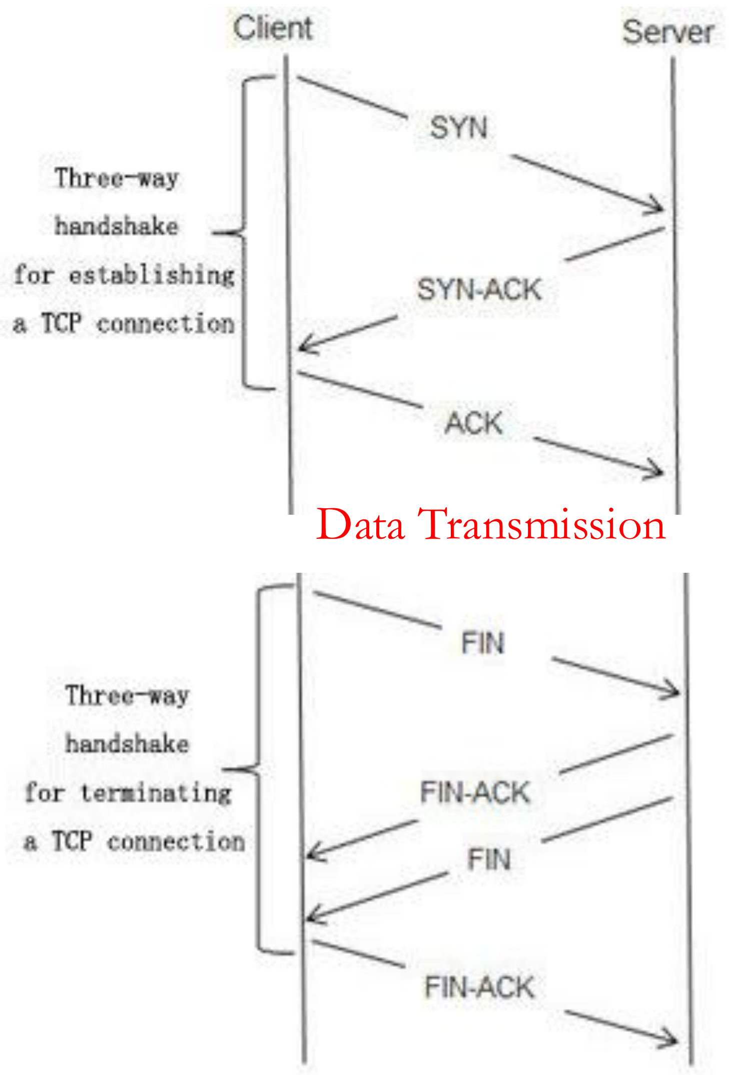

Client
Server
SYN
Three-way handshake for establishing a TCP connection
SYN-ACK
ACK
Data Transmission
FIN
Three-way handshake for terminating a TCP connection
FIN-ACK
FIN
FIN-ACK

In a normal TCP connection, after a client sends a SYN packet to initiate a connection with a server:

- The server responds with a SYN-ACK packet, indicating it's ready to establish the connection.
- The server then allocates resources and waits for the client's ACK response, which never arrives.

Waiting for clients could result in vulnerability

TCP/UDP: TCP SYN Flooding (2/3)

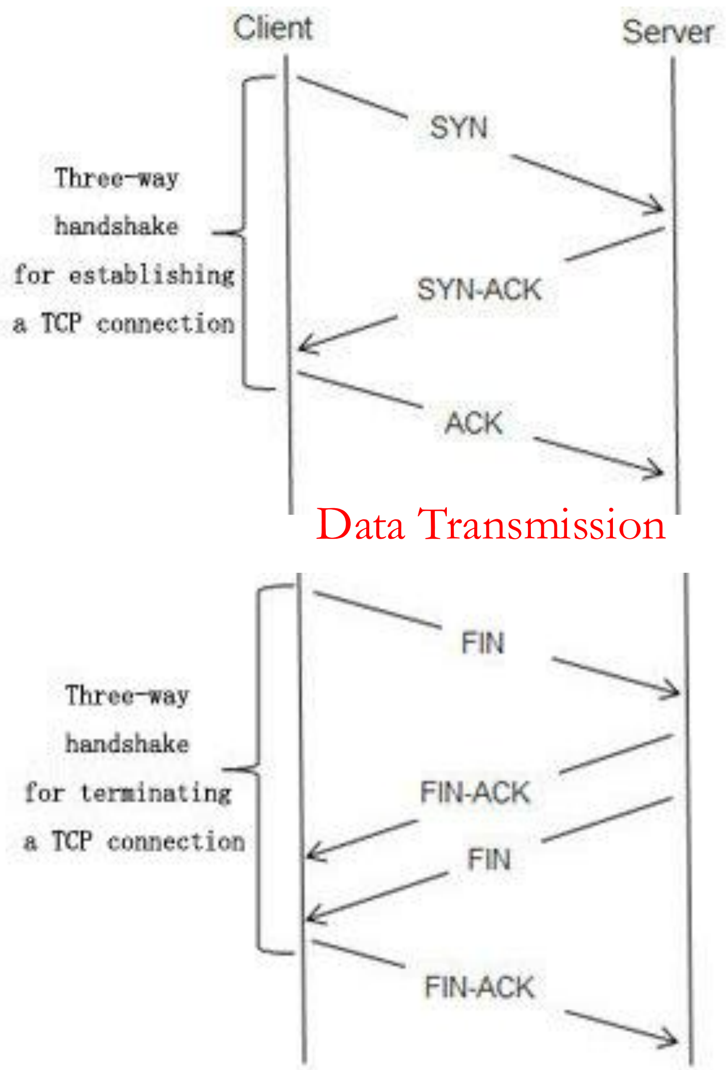

Client
Server
SYN
Three-way handshake for establishing a TCP connection
SYN-ACK
ACK
Data Transmission
FIN
Three-way handshake for terminating a TCP connection
FIN-ACK
FIN
FIN-ACK

A SYN Flood attack is a type of Denial of Service (DoS) attack that targets the TCP handshake process. Here's a brief explanation:

In a <u>normal</u> TCP connection, the client sends a SYN (synchronize) request to the server to initiate a connection. The server responds with a SYN-ACK (synchronize-acknowledge), and the client replies with an ACK (acknowledge), completing the three-way handshake.

In a <u>SYN Flood attack</u>, the attacker sends many SYN requests to the server but never completes the handshake by sending the final ACK. The server allocates resources and waits for the ACK that never comes, resulting in a large number of half-open connections. As the server's resources are consumed, it becomes overwhelmed and unable to handle legitimate connections, leading to service disruption.

TCP/UDP: TCP SYN Flooding (3/3)


Client
Server
SYN
Three-way handshake for establishing a TCP connection
SYN-ACK
ACK
Data Transmission
FIN
Three-way handshake for terminating a TCP connection
FIN-ACK
FIN
FIN-ACK

To mitigate a SYN Flood attack, we can use:

- SYN Cookies: This technique involves the server not allocating resources immediately upon receiving a SYN request. Instead, the server encodes information (namely cookie) in the SYN-ACK packet, and resources are only allocated if the client responds with the correct ACK. This avoids resource exhaustion from half-open connections.

- TCP Filtering: Implementing filters that drop suspicious or malformed SYN packets can help prevent excessive SYN requests from being processed.

TCP/UDP: UDP Flooding (Amplification Attack) (1/3)

UDP Flooding (Amplification Attack) is a type of Distributed Denial of Service (DDoS) attack that exploits the stateless nature of the User Datagram Protocol (UDP) and often amplifies the attack using a feature like UDP-based services that respond with larger data than the request sent.

1. Attacker Sends Small UDP Packets: The attacker sends a large volume of small UDP packets to various servers. These packets typically request data from services like DNS, NTP, or other UDP-based services that reply with larger responses.

2. Spoofing the Source IP: The attacker forges (spoofs) the source IP address of the UDP packets to make it appear as though they are coming from the victim's IP address.

3. Amplified Response: The server responds to the spoofed IP with a much larger data payload than the original request. Since UDP is connectionless, the server has no way to verify the legitimacy of the source IP address.

4. Flooding the Victim: The victim (no the server)'s network gets overloaded with massive amounts of response data, effectively clogging the network and degrading or taking down services.

TCP/UDP: UDP Flooding (Amplification Attack) (2/3)

Example:

- Memcached DDoS Attack (2018): This attack set records in amplification, peaking at 1.7 Tbps of traffic. It leveraged Memcached, a database caching system, to perform an amplification attack. By sending small requests (15 bytes) to vulnerable Memcached servers, attackers could trigger responses as large as 50,000 times the size of the original request, swamping the victim with an overwhelming flood of data.

- Overwhelmed Resources: The victim’s network bandwidth, CPU, and memory get overwhelmed by the large amount of incoming traffic.

TCP/UDP: UDP Flooding (Amplification Attack) (3/3)

Third-Party Server:

1. Rate Limiting and Traffic Shaping: Third-party servers can implement rate limits on responses to reduce the amount of traffic they can send out.

2. Response Filtering: Using response filtering techniques can prevent servers from sending large responses unless the request is verified.

3. Disable Unnecessary UDP Services: Third-party servers should disable UDP services that are not essential to avoid them being used for amplification.

Victim:

1. **Limit UDP-based Communication:** The victim's network can reduce its reliance on UDP-based services, as these are most vulnerable to amplification attacks.

2. **Implement Firewalls and Intrusion Prevention Systems:** Victims can configure firewalls to detect abnormal traffic patterns that could indicate an amplification attack. Blocking traffic from known vulnerable third-party servers can reduce the effect of the attack.

TLS

The protocol that encrypts talks before TCP between applications in different devices.

TLS: Background (1/2)

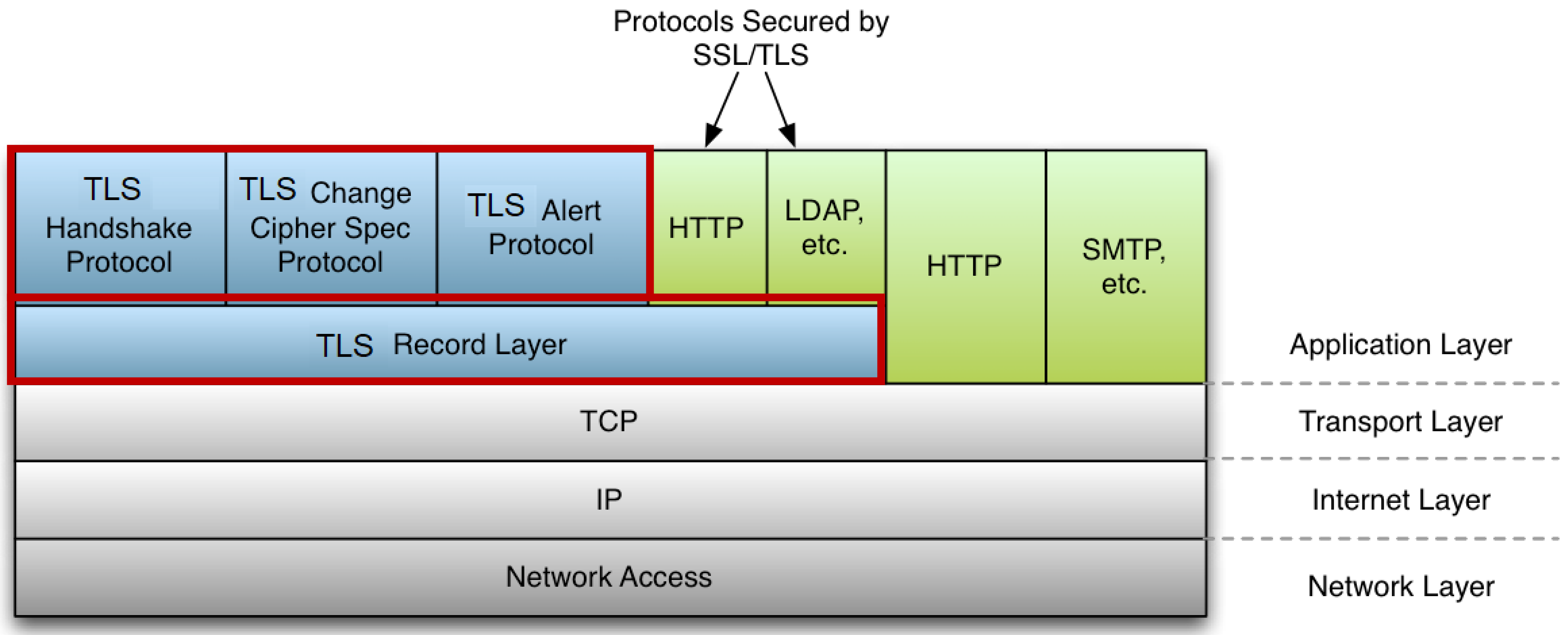
- TCP Payload in Plaintext: The payload in TCP packets consists of the actual data being communicated by applications. Without additional protection, this data is transmitted in plaintext, making it vulnerable to interception or tampering by malicious adversary.

- TLS for Payload Protection: The TLS (Transmission Control Protocol) protocol secures the communication by encrypting the payload of TCP packets, ensuring confidentiality, integrity, and authenticity. This prevents eavesdroppers from reading or altering the transmitted data.

TLS: Background (2/2)

1. Origin of SSL: SSL (Secure Sockets Layer) was developed by Netscape in 1994 to secure internet communications, providing confidentiality, integrity, and authentication.
2. SSL Versions: The first version, SSL 1.0, was never released due to security flaws. SSL 2.0 was released in 1995, followed by SSL 3.0 in 1996, which addressed many vulnerabilities.
3. Transition to TLS: In 1999, the Internet Engineering Task Force (IETF) standardized SSL 3.0 as TLS (Transport Layer Security) version 1.0, incorporating improvements and updates.
4. Subsequent Versions: TLS 1.1 was introduced in 2006, addressing weaknesses in TLS 1.0. In 2008, TLS 1.2 was released introducing support for stronger encryption algorithms and improved security mechanisms.
5. TLS 1.3: The latest version, TLS 1.3, was finalized in 2018, significantly simplifying the handshake process, reducing latency, and improving security by removing outdated cryptographic algorithms.

TLS: Architecture (1/2)



4.2 TLS: Architecture (2/2)

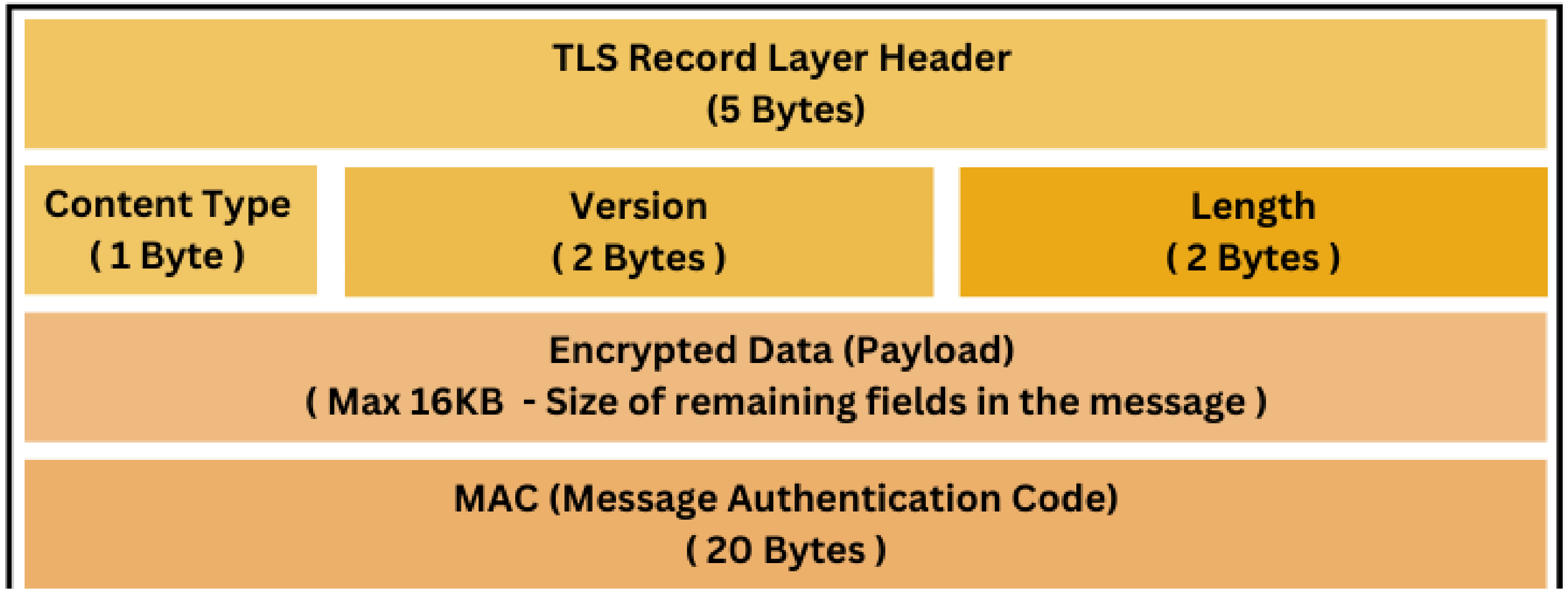
1. Handshake Protocol: It establishes a secure connection between the client and server by negotiating cryptographic algorithms, authenticating the parties, and exchanging session keys.

2. Record Protocol: It is responsible for providing confidentiality and integrity through encryption and message authentication, and ensuring that data is transmitted securely over the established TLS connection.

3. Alert Protocol: It communicates error and warning messages, indicating issues that arise during the handshake or data transfer phases and facilitating graceful termination of the connection when necessary.

4. Change Cipher Spec Protocol: It is a part of the TLS handshake process that signals the transition to a new cipher suite. When activated, it informs another party that subsequent messages will be protected using the negotiated encryption and authentication algorithms.

TLS: Record Protocol (1/5)

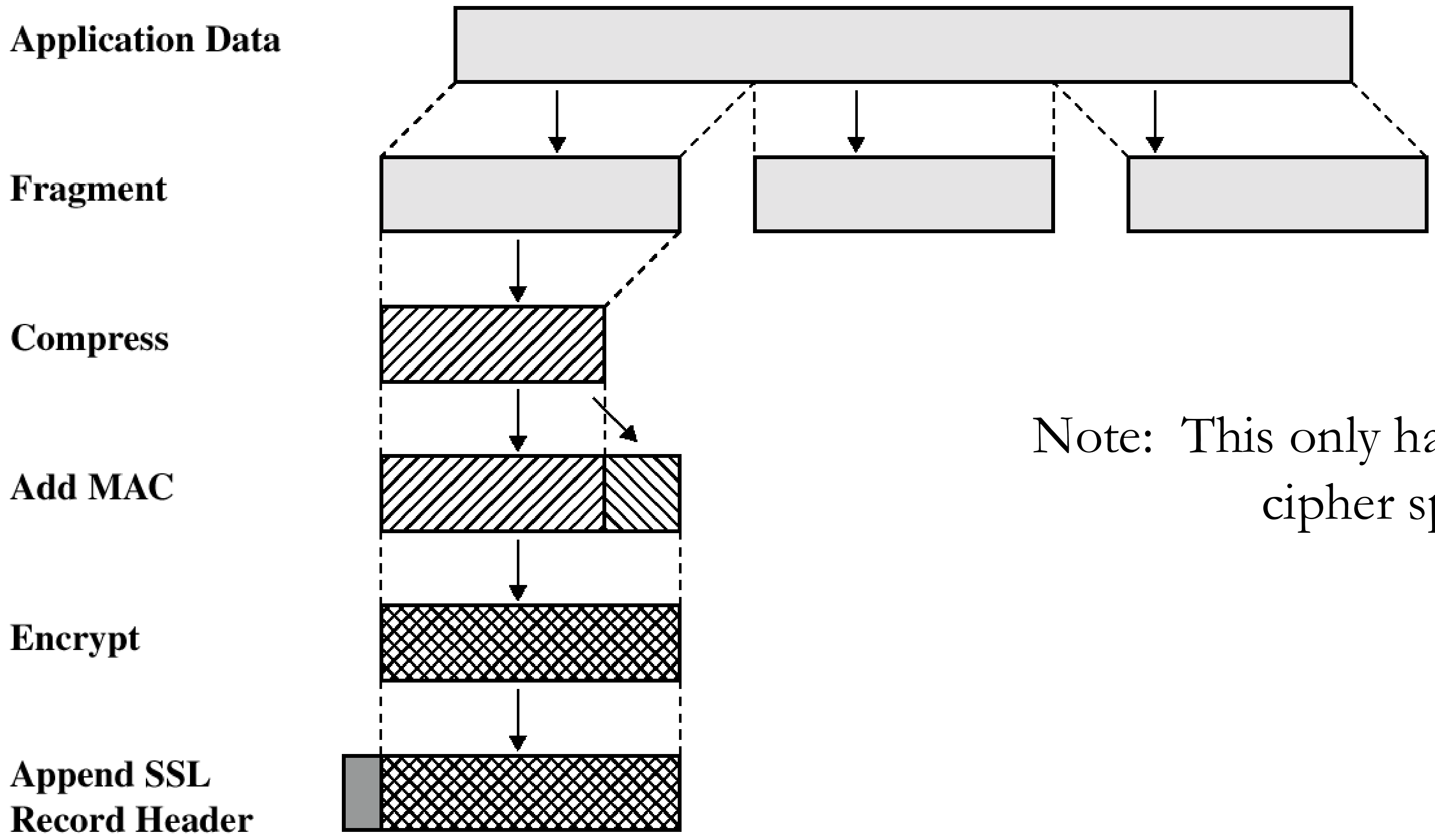


Note: plaintext before handshake protocol and change cipher spec protocol

Record Protocol Header: Each TLS record begins with a header that contains :

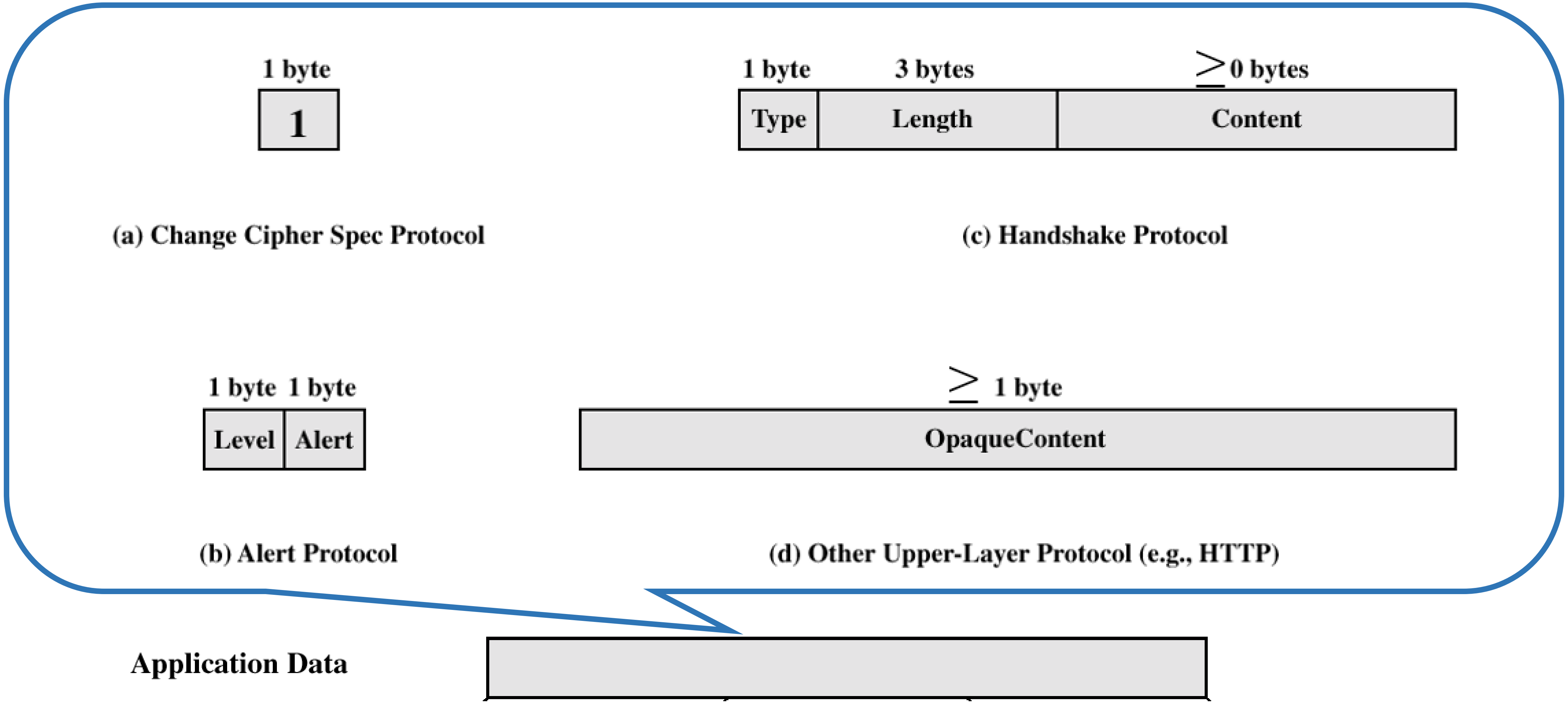
- Content Type: This type of payload protocol (e.g., Handshake, Application Data, Alert).
- Version: This specifies the version of the TLS protocol being used.
- Length: This field indicates the length of the data in the record.

TLS: Record Protocol (2/5)

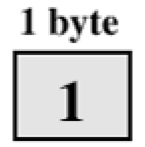


Note: This only happens after the change cipher spec protocol

TLS: Record Protocol (3/5)



TLS: Record Protocol (4/5)

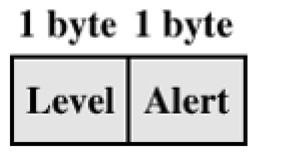


(a) Change Cipher Spec Protocol

- Consists of a single message with the value 1.

- The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the newly negotiated CipherSpec and keys.

TLS: Record Protocol (5/5)



(b) Alert Protocol

- Each alert message consists of 2 fields (bytes)
- The first field (byte): “warning” or “fatal”
- The second field (byte): reason

Fatal

- unexpected_message
- bad_record_MAC
- decompression_failure
- handshake_failure
- illegal_parameter
- …

Warning

- bad_certificate
- unsupported_certificate
- certificate_revoked
- certificate_expired
- certificate_unknown
- …

Fatal: An error that cannot be recovered from, leading to the immediate termination.

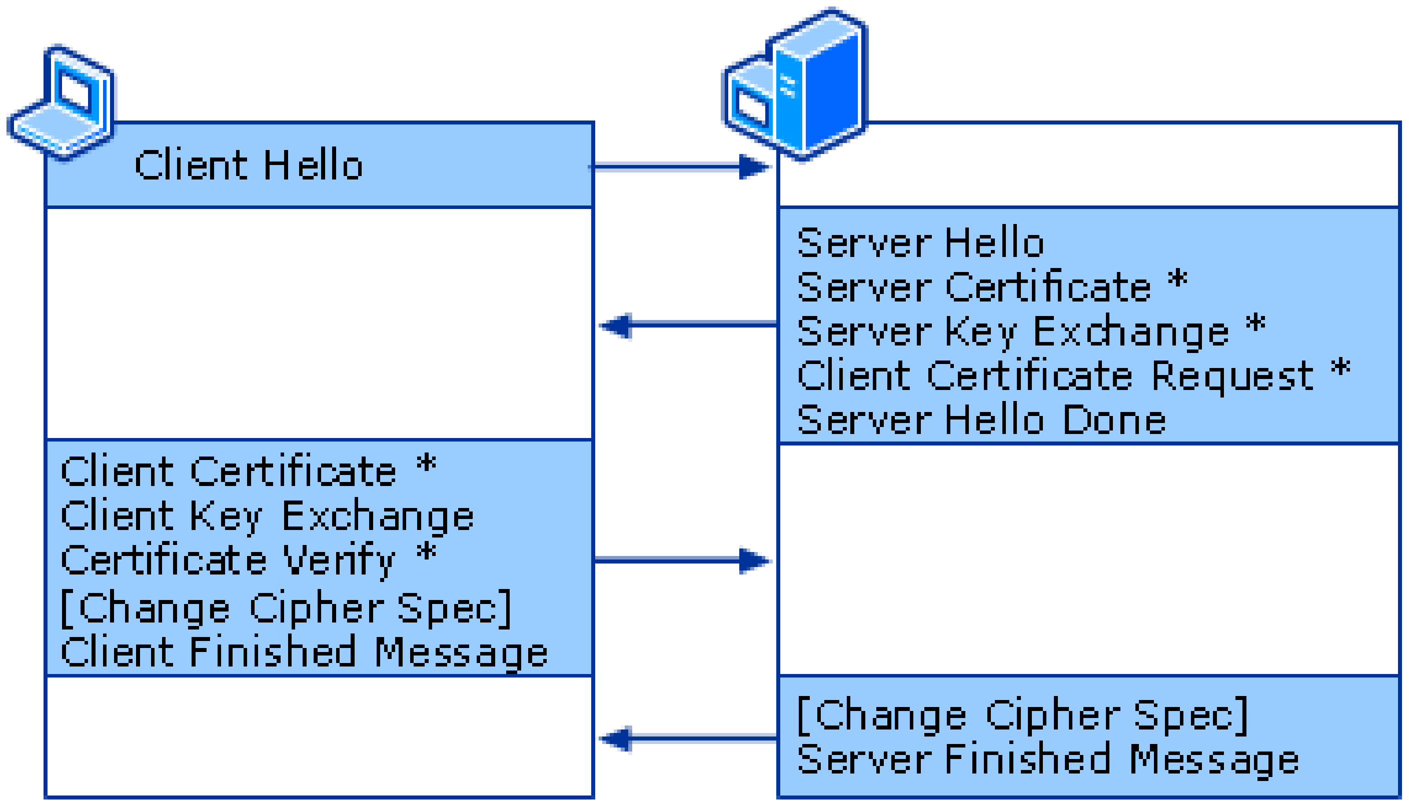
Warning: This indicates a non-critical issue, and the connection may continue.

TLS: Handshake Protocol (1/11)

As the most complex part of TLS, the Handshake protocol allows the server and the client to finish the following tasks:

1. Agree on a version of TLS to be used;
2. Authenticate each other (server authenticates client, optional);
3. Negotiate a set of cryptographic algorithms
4. Negotiate shared keys to be used in the TLS record protocol.

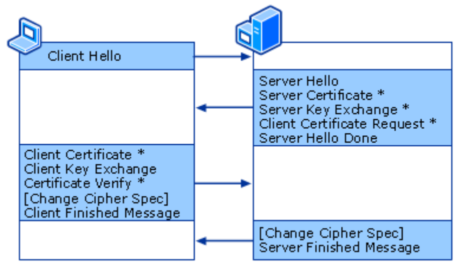
TLS: Handshake Protocol (2/11)



Version 1.2

1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

TLS: Handshake Protocol (3/11)



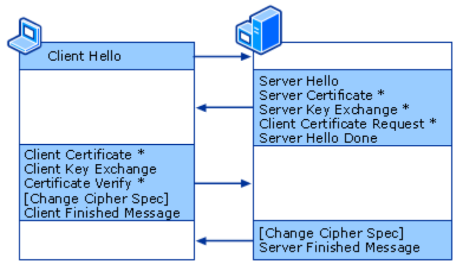
Version 1.2

1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

1. ClientHello: Contain protocol version, client nonce as well as the client's list of preferred ciphersuites

2. ServerHello: Contain chosen protocol version, server nonce as well as the chosen ciphersuite

TLS: Handshake Protocol (4/11)



Version 1.2

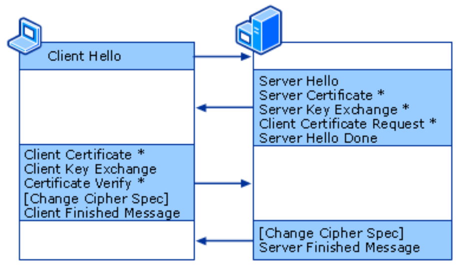
1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

3. Certificate
   - Required when server authentication is needed
   - X.509 certificate for one of the following type (depending on the key exchange method)
     - RSA Encryption Key: pk for encryption
     - Diffie-Hellman Public Key: for secure key exchange
     - ....

<u>The methods of generating shared key are different!</u>

Philosophy: Don't put all eggs in one basket!

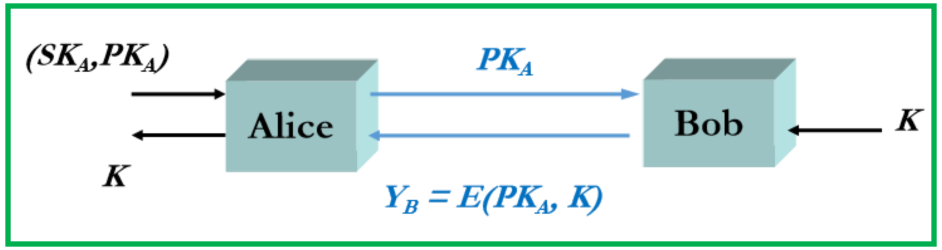
TLS: Handshake Protocol (5/11)

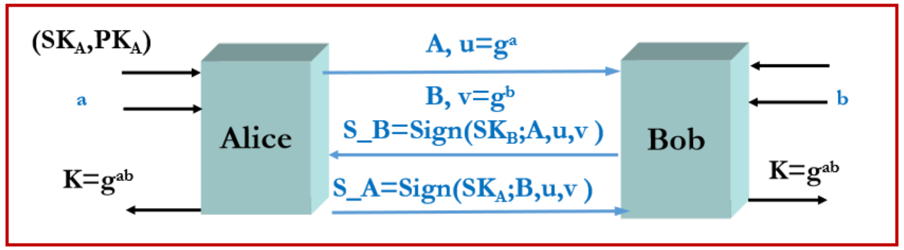


Version 1.2

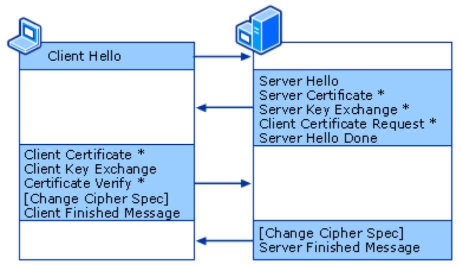
1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

4. Server Key Exchange: send something for key exchange from server when needed
   - Required when Diffie-Hellman key establishment is used
   - Not required when RSA key transport is used as the key exchange method (because client will generate a key and encrypt with RSA key)





TLS: Handshake Protocol (6/11)



Version 1.2

1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

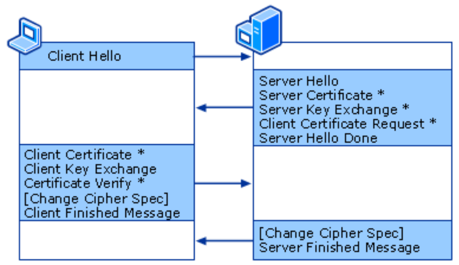
5. CertificateRequest

A sign of message including algorithms that the server requests and support for client-side authentication.

6. ServerHelloDone

A sign indicates that the server has finished sending all of its handshake messages.

TLS: Handshake Protocol (7/11)

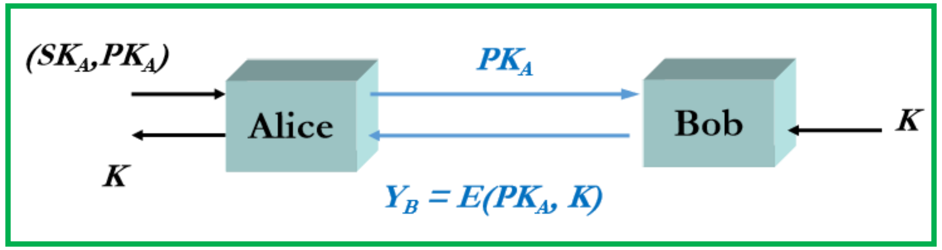


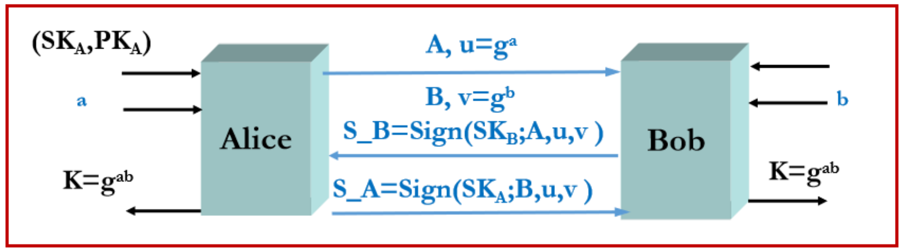
Version 1.2

1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

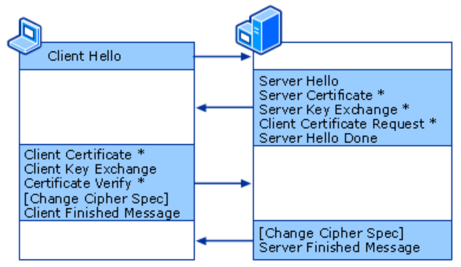
7. ClientCertifiate: send client's certificate

8. ClientKeyExchange
   – sends this message depending on algorithms for generating shared keys.





TLS: Handshake Protocol (8/11)



Version 1.2

1. ClientHello
2. ServerHello
3. Certificate
4. ServerKeyExchange
5. CertificateRequest
6. ServerHelloDone
7. ClientCertificate
8. ClientKeyExchange
9. CertificateVerify
10. Finished

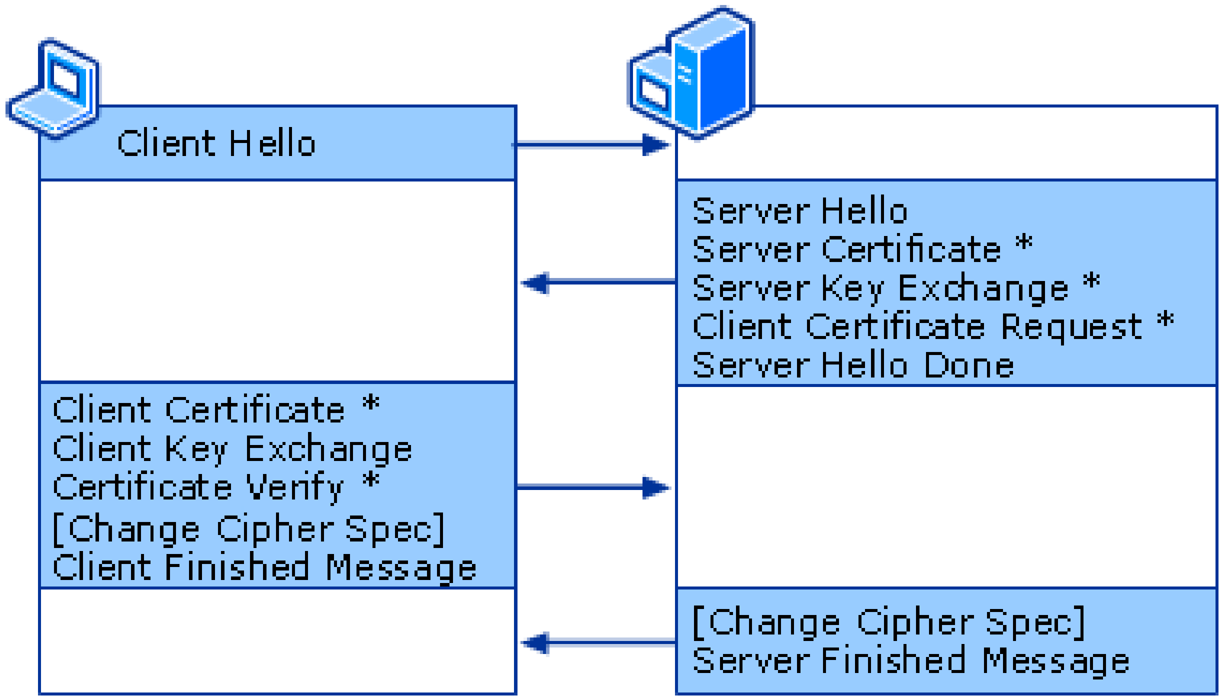
9. CertificateVerify

- Required only when client has a certificate for RSA/DSS signature and wants to be authenticated.
- Client signs all the handshake messages it has sent and received previously

10. Finished

This message is sent by both the client and server to indicate the end of the handshake process, ensuring all previous messages were sent and received correctly.

TLS: Handshake Protocol (9/11)



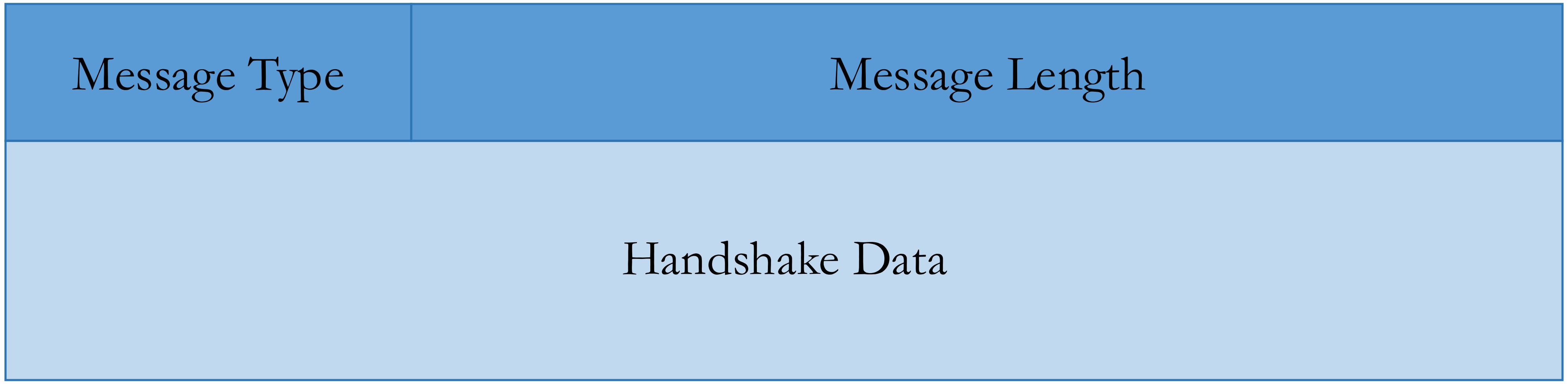
The Finished message in TLS is sent after the encryption has been established, meaning it is encrypted using the negotiated cipher suite. After both the client and server exchange the ChangeCipherSpec message, they both switch to the new encryption settings, and the Finished message is the first message encrypted using those settings.

Change Cipher Spec protocol is called before the handshake protocol (Finished) is completed

TLS: Handshake Protocol (10/11)

Handshake Protocol Header: Each message within the Handshake Protocol also has a header:

- Message Type (1 byte): This identifies the specific handshake message type
  - ☐ (e.g., Client Hello, Server Hello, Certificate).
- Message Length: (3 bytes) This indicates the length of the message following the header.



TLS: Handshake Protocol (11/11)

- The key derived by an TLS key exchange method through the handshake protocol is called pre-master secret (48 bytes), which can be
  - ☐ a secret encrypted with server's RSA public key or
  - ☐ a value derived by the DH key exchange technique.

- pre-master secret → master secret → shared keys
  - ☐ master secret = PRF(pre_master_secret, 'master secret', client_random||server_random)
  - ☐ shared keys = PRF(master_secret, 'key expansion', server_random||client_random)

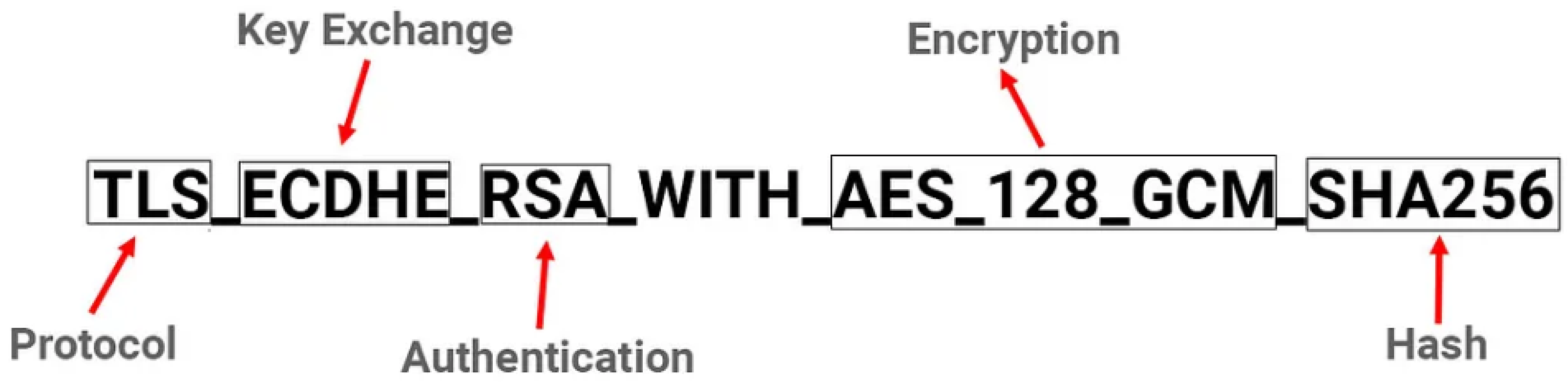
Note: PRF is a pseudo-random function. Using separate encryption keys for communication between two parties not only enhances security but also provides flexibility and compliance with protocols.

TLS: Cipher Suite (1/2)

A cipher suite in TLS is a set of cryptographic algorithms that define how secure communication is established between a client and a server. Each cipher suite specifies the following components:

- ☐ <u>Key Exchange Algorithm</u>: This algorithm is used to securely exchange keys between the client and server. It establishes a shared secret that will be used for encryption.
- ☐ <u>Authentication Algorithm</u>: This determines how the parties authenticate each other.
- ☐ <u>Symmetric Encryption Algorithm</u>: This algorithm is used to encrypt the data once the secure connection has been established.
- ☐ <u>Message Authentication Code (MAC) Algorithm</u>: This is used to ensure the integrity and authenticity of the data transmitted over the connection. It verifies that the message has not been altered during transmission.

TLS: Cipher Suite (2/2)



Two Examples:

- Key Exchange: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- Key Transport: TLS_RSA_WITH_AES_128_CBC_SHA

Note: The key transport merges key generation and authentication together!

TLS: HTTPS

HTTPS = HTTP + TLS

- use https:// rather than http://
- use port 443 rather than 80
- encrypts URL, document contents, form data, cookies, HTTP headers
- HTTP basic user authentication can be used!

Note: The adversary can till see the domain name from TCP protocol even one IP is hosting multiple domains.

TLS: VPN

Bob is a programmer who has developed an app that enables communication between apps on different devices. The first version of the app transmits data in plaintext. For version 2, Bob wants to implement secure communication and is considering either SSH or TLS. What are the key differences he would face when designing version 2 using SSH or TLS?

Answers: Both solutions will require partial rewriting of the app. SSH typically involves more manual client-side configuration, such as setting up port forwarding. TLS, on the other hand, integrates more seamlessly with applications, offering a more automated solution for secure communication with minimal client-side configuration (often through built-in libraries).

TLS: Summary

Advantage:

- Strong cryptography protections (C.I.A)
- Widely supported in many applications

Disadvantages:

- Performance Overhead: The encryption and decryption process can introduce latency and computational overhead, particularly in resource-constrained environments.
- Certificate Management and Requirement: The need for trusted certificates
- Complex Configuration: If programmers are designing TLS based secure apps, they need to know how TLS works inside apps and implement them.

DTLS

protocols similar to TLS but working on UDP

DTLS: Background (1/2)

UDP protocols are more preferable than TCP protocols in the applications of where low latency and efficiency are prioritized over reliability and ordered delivery.

- <u>Real-Time Communications:</u> Video conferencing require minimal delay for real-time interaction. UDP's lack of handshakes and retransmissions minimizes latency, making it suitable for these applications.
- <u>Live Streaming:</u> Streaming media like live broadcasts or sports events benefit from UDP, as minor packet loss is less noticeable to viewers than delays caused by retransmissions in TCP.
- <u>Online Gaming:</u> Many multiplayer online games use UDP to ensure fast updates of player positions and actions, where real-time responsiveness is more important than guaranteed delivery.

DTLS: Background (2/2)

TLS security protocol over TCP protocols is successful having the below key reasons:

- Reliable Transmission: Ensures no packet loss during communication.
- Ordered Transmission: Guarantees that all packets are received in the correct sequence.

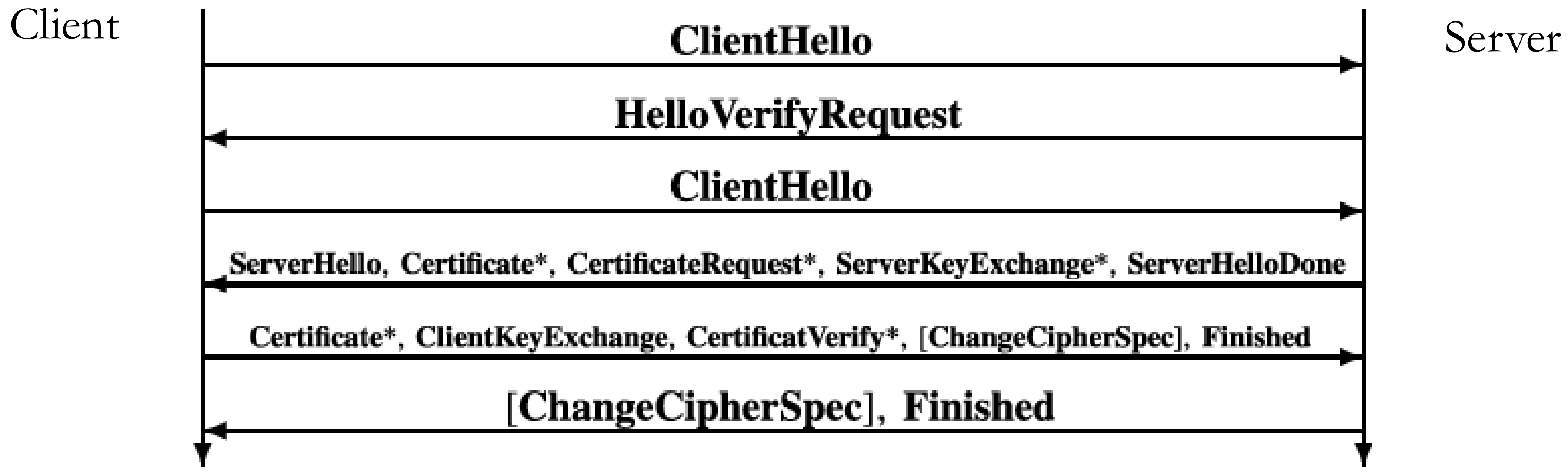
These properties are critical for TLS functionality. If either is compromised, TLS will encounter errors. Therefore, it is not feasible to directly adapt TLS for UDP without significant modifications.

DTLS: Protocol (1/3)

Designing a variant of TLS over UDP must account for:

- Unreliable Nature of UDP: DTLS must handle packet loss, reordering, and duplication inherent in UDP. Mechanisms are implemented within DTLS to ensure a successful handshake under these conditions.
- Replay and Out-of-Order Protection: DTLS includes safeguards against packet replay attacks and the possibility of out-of-order packet arrivals.
- Insecurity of UDP: DTLS addresses potential security vulnerabilities associated with the open nature of UDP.
- Real-Time Constraints: DTLS must optimize for efficiency, especially when critical messages, such as those in the handshake protocol, require retransmission.

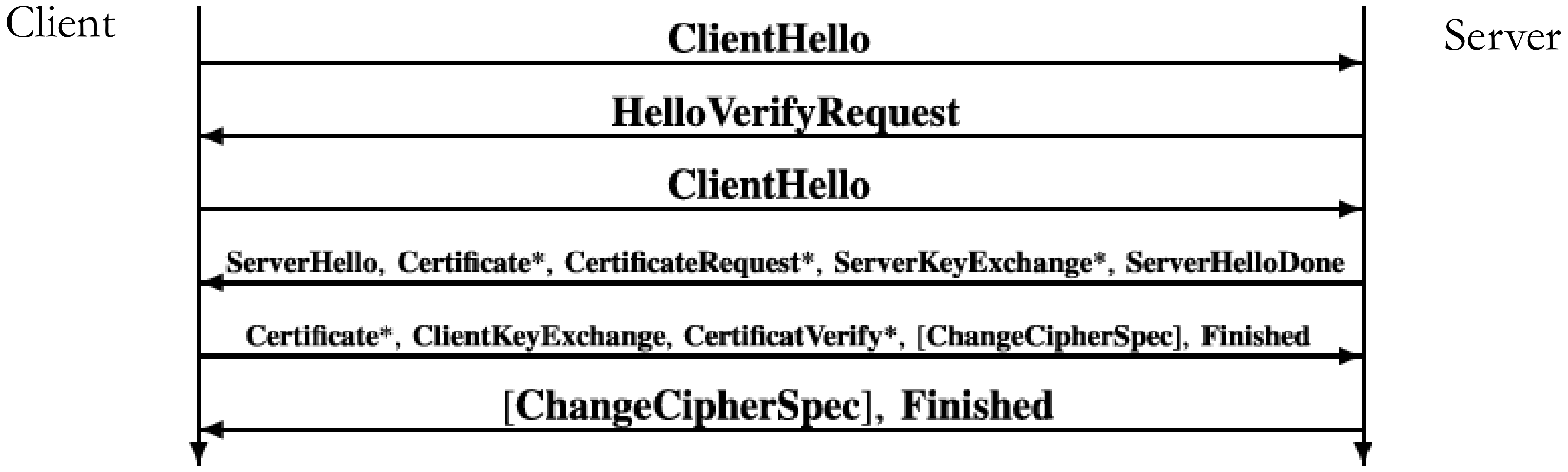
DTLS: Protocol (2/3)


Client
Server
ClientHello
HelloVerifyRequest
ClientHello
ServerHello, Certificate*, CertificateRequest*, ServerKeyExchange*, ServerHelloDone
Certificate*, ClientKeyExchange, CertificatVerify*, [ChangeCipherSpec], Finished
[ChangeCipherSpec], Finished

HelloVerifyRequest (Cookie):

To prevent DoS attacks, the server responds with a "HelloVerifyRequest", including a cookie. The server remains stateless at this point, asking the client to send the cookie back in a subsequent message. TLS does not need this because it runs over TCP, which already establishes a connection before the handshake.

DTLS: Protocol (3/3)


Client
Server
ClientHello
HelloVerifyRequest
ClientHello
ServerHello, Certificate*, CertificateRequest*, ServerKeyExchange*, ServerHelloDone
Certificate*, ClientKeyExchange, CertificatVerify*, [ChangeCipherSpec], Finished
[ChangeCipherSpec], Finished

Without considering retransmissions, reordering, and packet loss, the cookie is the key differentiator in the handshake protocol itself. The rest of the handshake steps are very similar between TLS and DTLS.

QUIC

advanced DTLS protocols

QUIC: Background (1/2)

- TCP is reliable but slow.
- Combining TLS with TCP is secure but further increases overhead.
- UDP is fast but lacks reliability and security.
- DTLS aims to create secure transmission over UDP.

	Reliable	Fast	Secure
TCP Protocol	YES	NO	NO
UDP Protocol	NO	YES	NO
TLS Protocol	YES	NO	YES
DTLS Protocol	NO	YES	YES

QUIC: Background (2/2)

- QUIC (Quick UDP Internet Connections) was originally developed by Google in 2012 as an experimental transport protocol. The goal was to improve web performance by reducing latency and enhancing the user experience for applications that rely on secure and fast connections.

- In 2016, Google began working with the Internet Engineering Task Force (IETF) to standardize QUIC. In May 2020, the IETF officially published RFC 9000, which standardized QUIC as an IETF protocol. This marked a significant milestone, as QUIC became an official transport layer protocol that could be widely adopted across the internet.

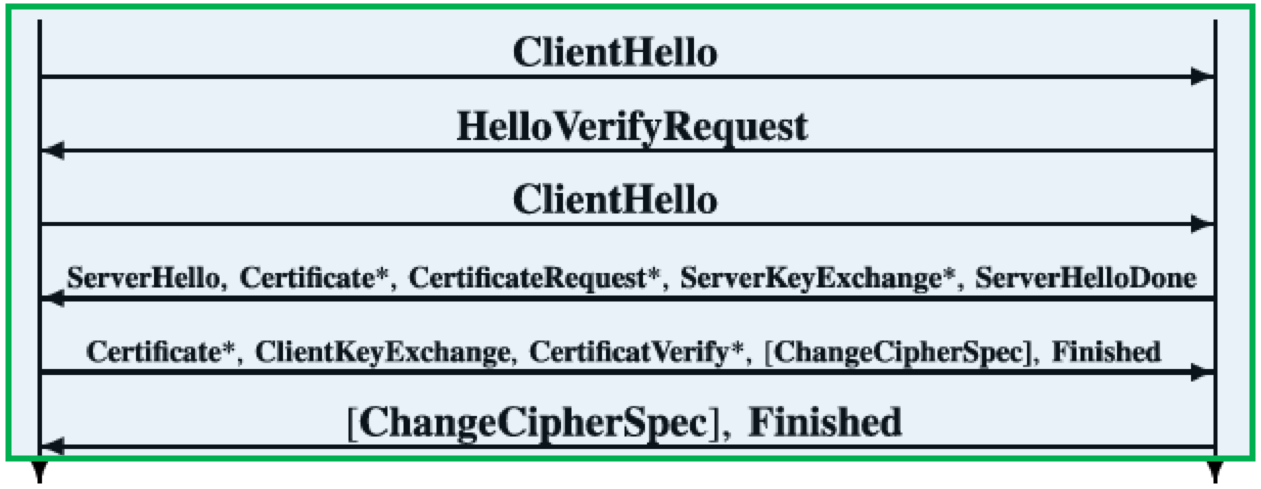
QUIC: Protocol (1/6)

QUIC: Similar to TLS:

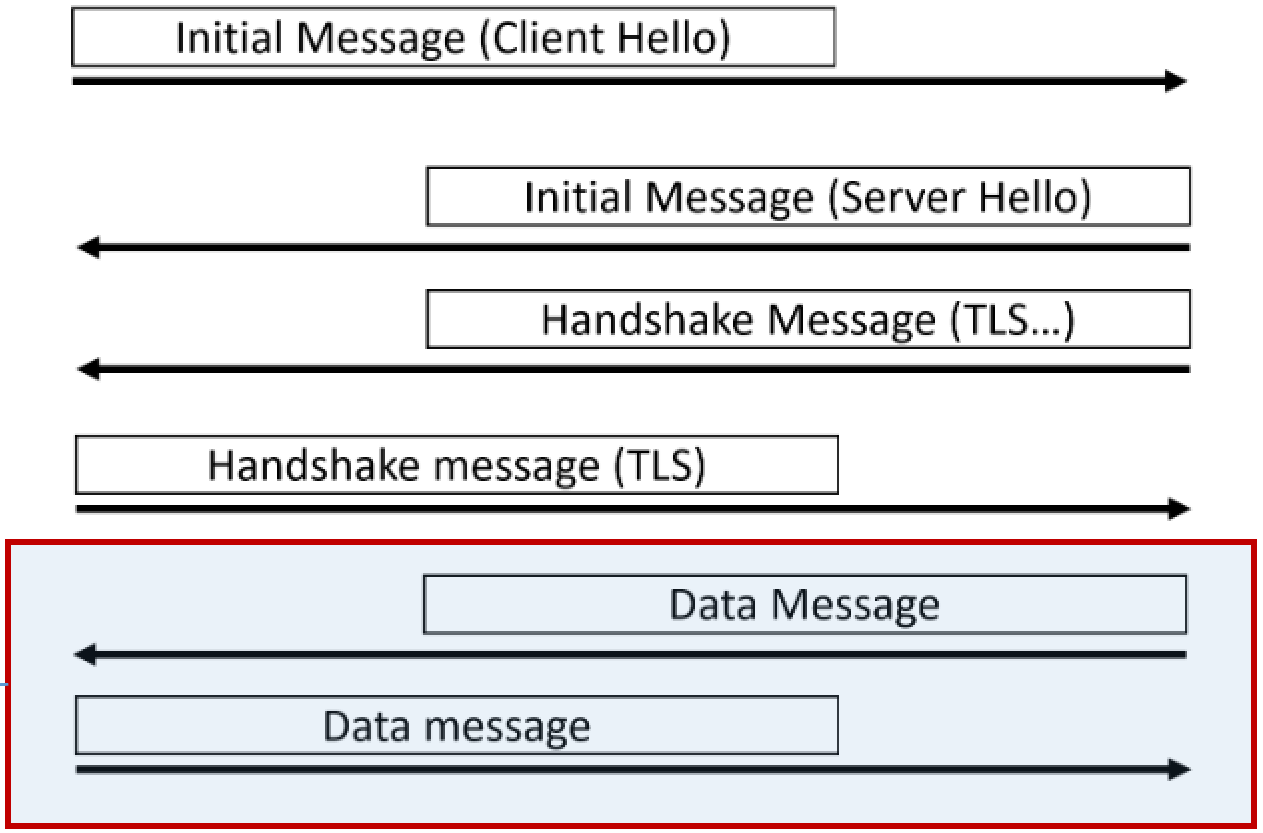
- Handshake Protocol: Establishes secure communication parameters and negotiates encryption algorithms.
- Change CipherSpec Protocol: Signals that the subsequent data will be encrypted using the negotiated keys.
- Alert Protocol: Handles error reporting and connection closure.
- Application Data (Record) Protocol: Transmits encrypted application data once the secure connection is established.

QUIC: Protocol (2/6)

DTLS


ClientHello
HelloVerifyRequest
ClientHello
ServerHello, Certificate*, CertificateRequest*, ServerKeyExchange*, ServerHelloDone
Certificate*, ClientKeyExchange, CertificatVerify*, [ChangeCipherSpec], Finished
[ChangeCipherSpec], Finished

QUIC

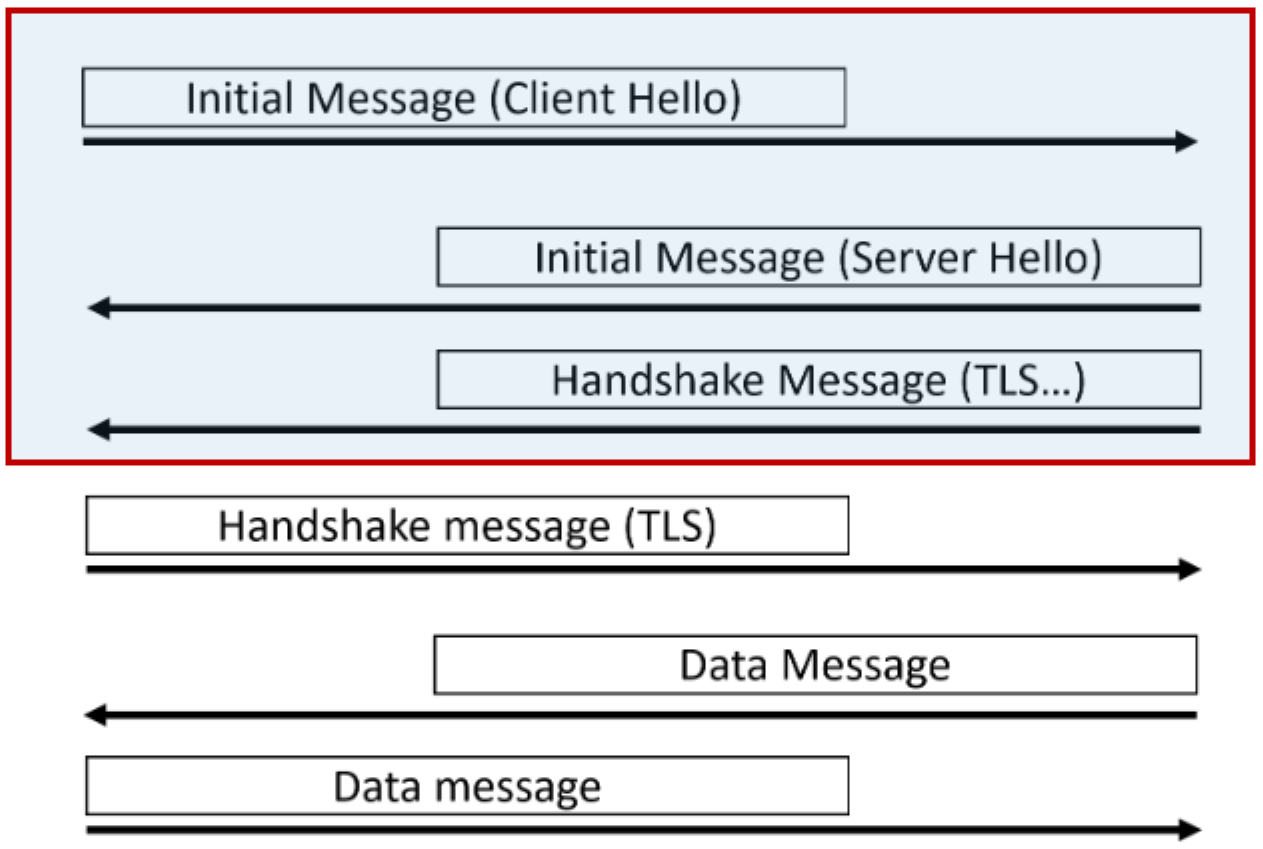

Initial Message (Client Hello)
Initial Message (Server Hello)
Handshake Message (TLS...)
Handshake message (TLS)
Data Message
Data message

QUIC allows servers to start transmissions before the finish of handshake protocol.

QUIC: Protocol (3/6)

- In QUIC, the ClientHello message includes key exchange parameters, which allows the server to respond with a ServerHello and any additional handshake information in the same exchange.
- This setup eliminates the need for separate messages to establish encryption parameters, which is a common step in traditional handshakes.

QUIC


Initial Message (Client Hello)
Initial Message (Server Hello)
Handshake Message (TLS...)
Handshake message (TLS)
Data Message
Data message

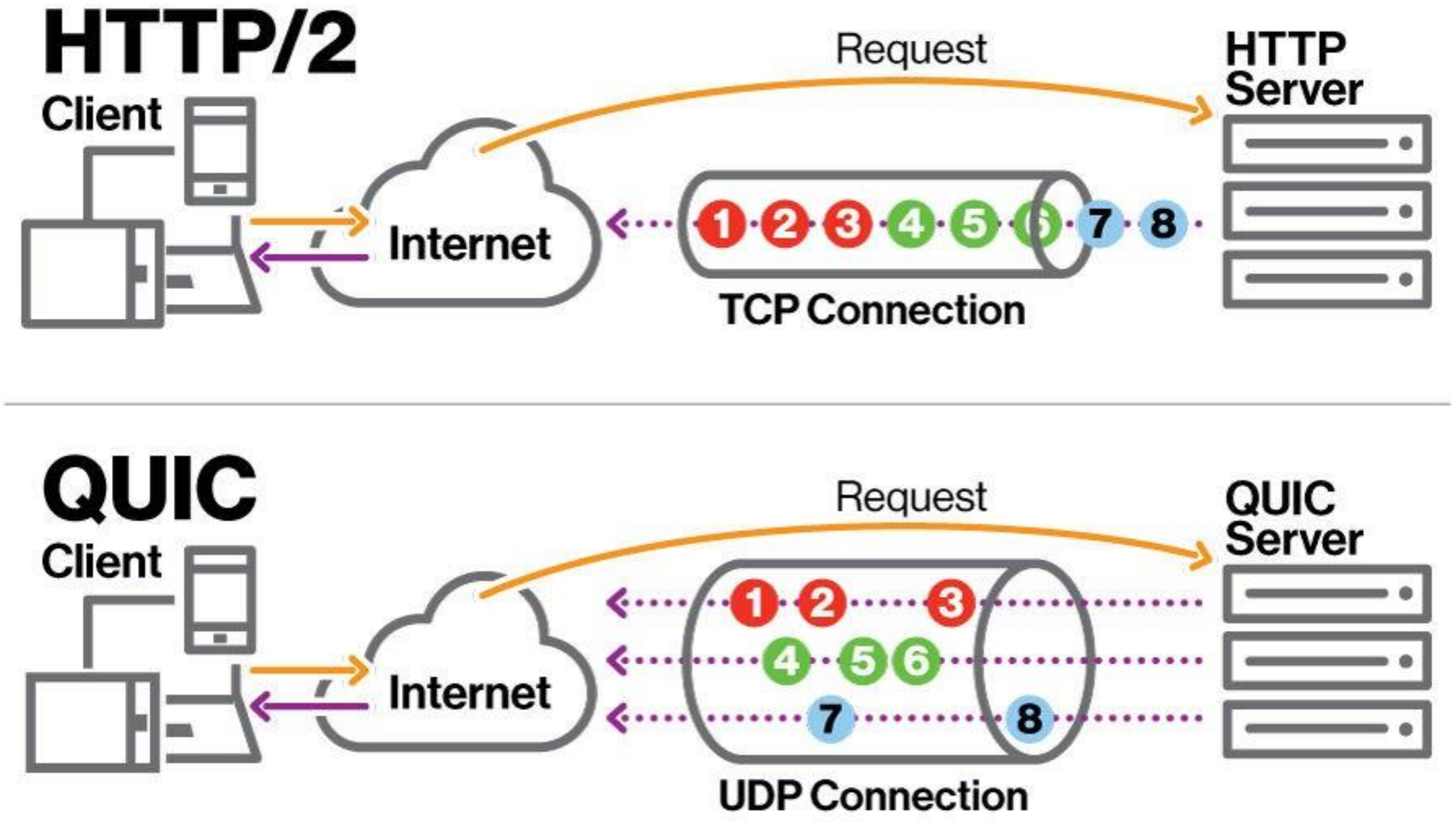
QUIC: Protocol (4/6)

Reliable& Fast (Together):

- Like TCP, QUIC uses ACKs to ensure that data is reliably delivered. Each packet sent by QUIC is acknowledged by the receiver, and if a packet is lost, QUIC retransmits it.

- QUIC tracks acknowledgments for each packet independently, allowing it to retransmit lost packets without affecting other streams of data.
  - ☐ TCP Head-of-Line Blocking: In TCP, when a packet is lost, the entire data stream must wait for that packet to be retransmitted before continuing. This is known as head-of-line blocking, and it can slow down communication.
  - ☐ QUIC's Solution: QUIC supports multiplexing multiple independent streams within a single connection. If a packet in one stream is lost, it does not block the other streams, allowing the connection to continue without unnecessary delays.

QUIC: Protocol (5/6)



multiplexing multiple independent streams within a single connection. If a packet in one stream is lost, it does not block the other streams.

QUIC: Protocol (6/6)

Congestion Control Flexibility in QUIC:

- TCP-like Congestion Control: QUIC supports TCP-style congestion control mechanisms. This provides a balance between efficient data transfer and avoiding network congestion.

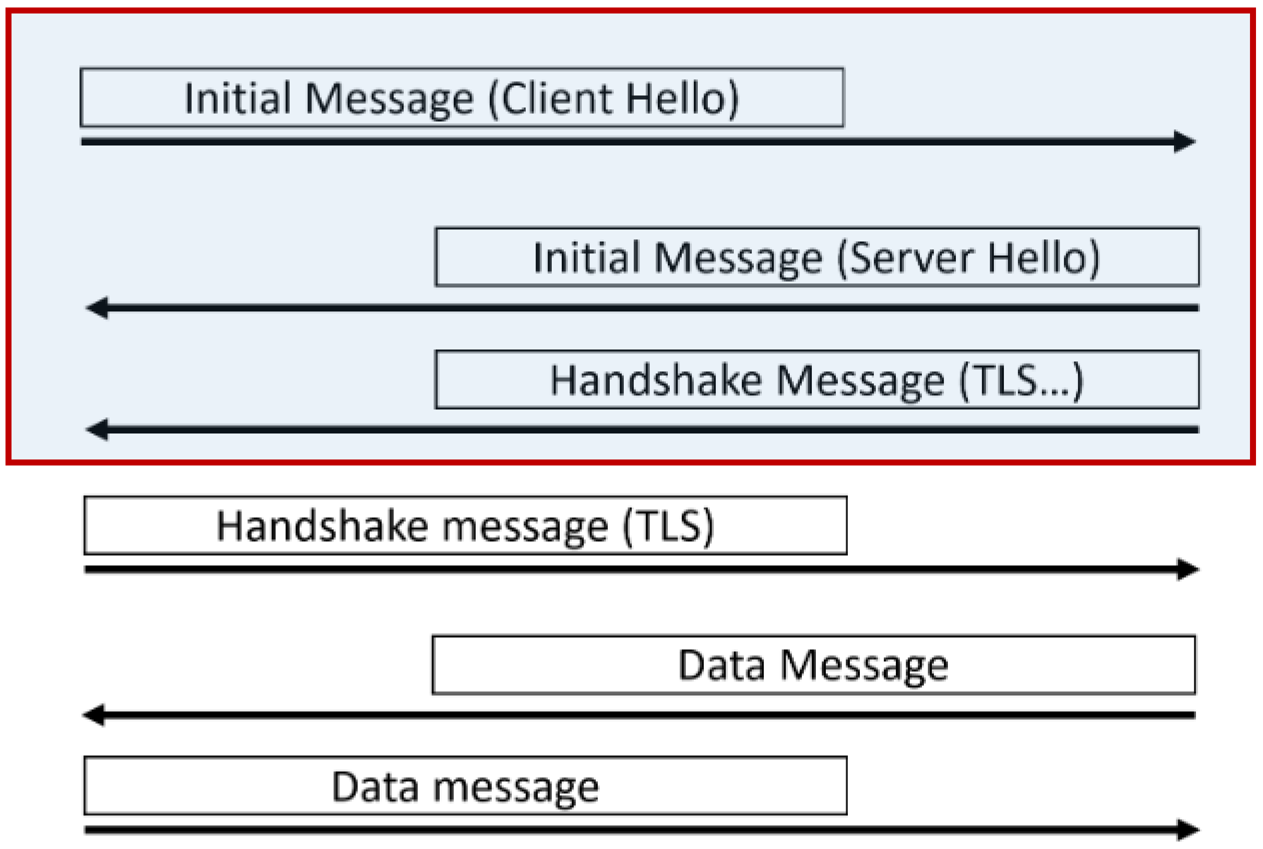
- UDP-Like Speed: Unlike TCP, UDP does not have built-in congestion control, which allows it to send data continuously without worrying about network congestion, leading to lower latency in real-time applications like video or gaming. While QUIC is built on top of UDP, it can bypass TCP's strict congestion control rules to achieve faster data transfer, especially in networks with high bandwidth and low congestion.

QUIC inherits the reliability and congestion control of TCP, while offering improvements such as lower latency, and built-in security.

QUIC: Summary (1/2)

	Reliable	Fast	Secure
TCP Protocol	YES	NO	NO
UDP Protocol	NO	YES	NO
TLS Protocol	YES	NO	YES
DTLS Protocol	NO	YES	YES
QUIC Protocol	YES	YES	YES

QUIC: Summary (2/2)


Initial Message (Client Hello)
Initial Message (Server Hello)
Handshake Message (TLS...)
Handshake message (TLS)
Data Message
Data message

In QUIC, the ClientHello message is the first step in establishing a connection and does not inherently authenticate the client's identity. This means that the server receives a ClientHello without knowing whether the request is coming from a legitimate client or a malicious actor.

Disadvantage: Due to more resouse in server hello. This lack of authentication can lead to scenarios where an attacker sends multiple ClientHello messages to the server (DoS)