Networking Techniques and Strategies to Benefit User Experience in Online Multiplayer Games

Networking Techniques and Strategies to Benefit User Experience in Online Multiplayer Games

Samuel Buzz Appleby
MSc Computer Game Engineering
Newcastle University
Newcastle Upon Tyne, UK
b7034806@newcastle.ac.uk

ABSTRACT

This report discusses the evolution and implementation of networking techniques in the games industry; how certain case studies, such as Quake 3, tackled the networking model to give the user the best experience. Discussion will include the early days of online play: the techniques and strategies used that ultimately resulted in the networking models seen today, leading towards modern day implementation in game engines like Unreal Engine 4. There have been many advances in the algorithms and techniques used in the past 20 years, each to benefit either the system or as a unique implementation. Games exist today, that still use the same methods from when online games were first created. Networking can be implemented in a multitude of ways; this paper will explore these different scenarios and use real case studies for these approaches.

1 Introduction

Any multiplayer game that requires online capabilities[1] must contain some degree of networking infrastructure. The requirements of this network can vary; some games may only require a global leaderboard of player scores, updated whenever a user finishes the game, while others, such as MMOs, need to maintain a persistent online world.

In games where the network has a heavy influence on gameplay, it is important that the best decisions are made for each unique online scenario that can occur. One aspect of the network would oversee user login, authentication, user validation and so forth; while another would oversee message passing across the internet, updating all participants the new live view of the game.

Once players have been authenticated and have entered an online game, the network must then pass information effectively between clients and servers to tackle the problems that arise with a networked game.

Networked games have been in production for over 20 years, and during that time the technologies have evolved greatly. This section will investigate the types of ways messages can be sent across the internet; the different network architectures, and the types of games that benefit from any combination of these.

[1] i.e., Any non-local multiplayer game.

1.1 - Networking Protocols

A network protocol is an established set of rules that determine how data is transmitted between different devices in the same network. [1]

These protocols can be categorized into 3 main types:

1. Communication protocols:
   - These determine how data should be assembled and transmitted to another device.
   Examples: Transmission Control Protocol (TCP) / User Datagram Protocol (UDP) / Hypertext Transfer Protocol (HTTP)

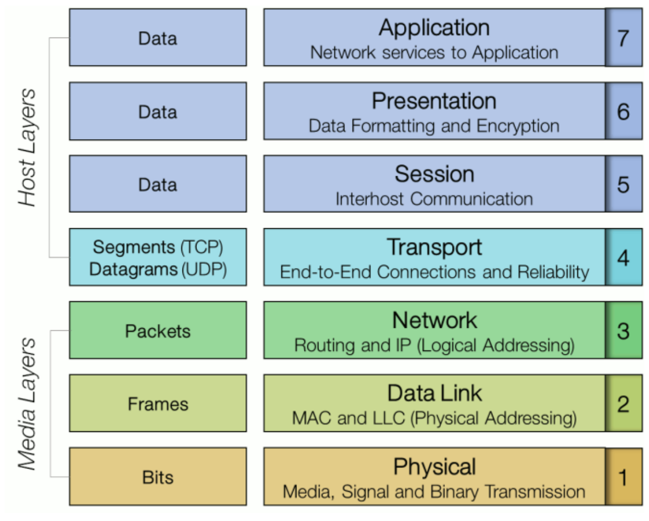
2. Security protocols:
   - These define how the data sent is protected, encrypted, and guarded against any unauthorized users.
   Examples: Hypertext Transfer Protocol Secure (HTTPS) / Secure Socket Layer (SSL)

3. Management protocols:
   - These monitor and maintain the network; affecting lots of devices on the network simultaneously to ensure optimal performance.
   Examples: Simple Network Management Protocol (SNMP) / Internet Control Message Protocol (ICMP). [3]

For any two devices to transmit their data successfully, they must be using the same protocol; for example, a client sending data packets through the Transmission Control Protocol (TCP), would fail if the receiving server was expecting packets through a difference communication protocol.

1.2 – The Open Systems Interconnection (OSI) Model

The OSI model was designed as a conceptual framework that breaks down the functions of a networking system into 7 abstractions.



Host Layers
Media Layers
Data
Application
Network services to Application
7
Data
Presentation
Data Formatting and Encryption
6
Data
Session
Interhost Communication
5
Segments (TCP)
Datagrams (UDP)
Transport
End-to-End Connections and Reliability
4
Packets
Network
Routing and IP (Logical Addressing)
3
Frames
Data Link
MAC and LLC (Physical Addressing)
2
Bits
Physical
Media, Signal and Binary Transmission
1

Figure 1: The OSI Model [3]

As the entire network foundation was growing larger and more complex, the OSI model was designed to remove those complexities and create a definitive model that allows new protocols and network services to have their designated place within a network.

The model follows a hierarchical structure whereby the higher up the layers, the closer the end results. Each layer in the network has its own role and protocols for handling data [4].

1. Physical Layer - This is the lowest-level layer. It handles the transmission of bits. The issues faced here included bit error checking, number of volts to represent a bit change, duration of bits, allowance of simultaneous transmission etc. These issues are normally related to mechanical, electrical, and functional fields.
   Example device: Hubs.

2. Data Link Layer – This provides error-free data transfer between two nodes. Here, the data is broken up into *data frames* that contain the source and destination Media Access Control (MAC) addresses. These frames are sent sequentially and acknowledged by the receiver. Ethernet is an example of a Data Link protocol.
   Example device: Switches.

3. Network Layer – This controls the routing of data. Data exists in the form of packets. Challenges here include deciding how packets should be routed across the network, how to deal with congestion and the optimum paths to take. When a packet reaches this layer, the source and destination IP addresses are added. The IP protocol is an example of a protocol at this layer.
   Example device: Routers.

4. Transport Layer – Here, most of the logistics of data transferal are dealt with, including the size of the data to send, rate of transfer, destination etc. The Transport Layer can correct and resend packets. This involves using a reliable protocol that can detect when an incorrect or damaged packet is received and use acknowledgement to ensure that all the data is transferred. Source and destination port numbers are added at this level. The two most common protocols that are used here include the Transmission Control Protocol (TCP, reliable), and the User Datagram Protocol (UDP, unreliable).

5. Session Layer – Whenever multiple devices need to transmit data to each other; a *session* must be opened across the network. Without a session, no network devices would be allowed to communicate. This layer handles the session set up, coordination [2] and the termination of the connection. Session identification control which parties are allowed access to the session at any given time. Communication between the systems may go in one, or both, directions at once. Tokens are used to ensure that critical operations are not performed by both devices at once, with only the member holding the token being able to perform the action. The Network File System (NFS) is a protocol used here, which allows remote hosts to mount file systems over a network and interact with those file systems as though they are mounted locally [5].

6. Presentation Layer – The data moving through this layer undergoes conversion into something that is readable by the Application layer. As data can be transferred in many formats from various sources, it must be transformed so that an application can understand the message it has received. Data encryption/decryption, character/string conversion and data compression occur here. Messages being sent are converted into a generic form, ready to be passed down the network. Received messages are converted from the generic form into a format that is readable to the receiving application. As devices have different codes for representing data, this conversion allows computers with different data representations to communicate.

7. Application Layer – This is the layer closest to the end user. Information sent by the user is first received here,

[2] Message exchange, response time, for example

and any incoming data must be displayed. Interfaces are provided to applications that allow them to access network services. It is important to note that software applications are not part of the application layer, instead the application layer is responsible for providing the protocols and managing the data so that is it meaningful to the user. For example, a website that requests content from the application layer will receive it back in the required format. Hypertext Transfer Protocol (HTTP) is used at this layer.

2 Communication Protocols

In online games, there are potentially hundreds of users in each world. It is important to choose the best communication protocol to pass data between the users in the game.

2.1 – Packets

2.11 - Packet Switching

Instead of passing entire chunks of data across a network, they are broken down into smaller pieces[3] of information called packets. These packets are then routed across the network and reassembled at the destination. All packets consist of a header and a payload.

Definition of Packet Switching from the Fiber Optics Standard Dictionary:

The routing and transferring of data by means of addressed packets so that a channel is occupied during the transmission of the packet only, and upon completion of the transmission the channel is made available for the transfer of other traffic. [6]

2.12 – Packet Header

The packet header precedes the payload and contains all the information required to be routed across the network and be corrected if needed.

Games have freedom of choice when it comes to choosing the suited communication protocol for the Transport Layer. However, to pass information across the internet the protocol must be embedded inside a protocol higher up on the OSI model, layer 3. The most popular of these is the Internet Protocol (IP).

IPv4 is the protocol used to route the most Internet traffic even today [8].

Here is an example of an IPv4 header:

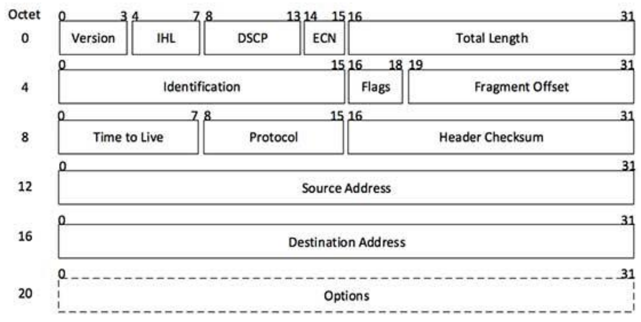


Figure 2: IPv4 Header [9]

This header contains a minimum of 20 bytes[4] of information. The 'Options' field is not normally used but is available if extra information was required that would not normally be available.

Packets are all assigned a value when the data is originally broken down. This allows the original data to be reconstructed using these packets once it has been received at the destination. Data packets can vary greatly, both in size and contents, depending on their protocol. Their configuration will determine the speed across which it they are sent across the network, in addition to the time required to process and reassemble them.

2.13 – Packet Encapsulation / De-encapsulation

It is important to note that the information stored in a packet is *encapsulated*. When data is received at the Application layer, there are no headers, only the data itself. Progressing down the OSI model, other layers will then wrap this data in their own header, for example, when a packet travels to the Transport layer, the protocol used, such as UDP, will add a UDP header. This header will contain all the necessary information to route it through the network. The next layer down would treat *all* the information it receives as data, and then wrap it further in another header.

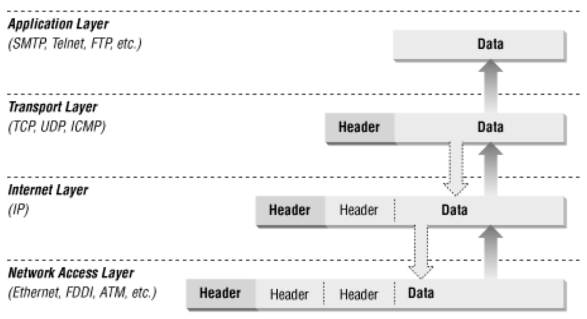


Figure 3: Packet Encapsulation [10]

Each layer will add to its own header the information it requires, so at the Network Layer, an IP protocol will add the Source and

[3] The maximum size of which will be defined by a protocol header.

[4] 5 32 bit words = 20 bytes

Destination Address fields, identifying the IP addresses of the sender and the receiver.

Therefore, the physical layer will contain the largest amount of data. It will contain the headers of each process that added one previously, in addition to the data itself, which persists for the duration of the transfer.

Packet de-encapsulation involves the reverse of the previous process. At the receiving computer, the packet moves back up through the OSI model, starting at the physical layer and moving upwards to the application layer, where it will have the formatted data to display to the user.

To do this, header information must be removed at the appropriate layer, until only the data is left for processing at the application layer. For example, a transport protocol header like a UDP header would be removed when the packet reaches this layer.

A popular analogy that assists in imagining this process is that an onion represents the data, with each layer being peeled off as the model is traversed:

Client sends a packet:

1. Application Layer
   [Data]

2. Presentation Layer
   [Data]

3. Session Layer
   [Data]

4. Transport Layer
   [Data] [UDP Header]

5. Network Layer
   [Data] [UDP Header] [IP Header]

6. Data Link Layer
   [Data] [UDP Header] [IP Header] [Frame Header]

7. Physical Layer
   [Data] [UDP Header] [IP Header] [Frame Header]

Receiving computer retrieves the packet:

8. Physical Layer
   [Data] [UDP Header] [IP Header] [Frame Header]

9. Data Link Layer
   [Data] [UDP Header] [IP Header] [Frame Header]

10. Network Layer
    [Data] [UDP Header] [IP Header]

11. Transport Layer
    [Data] [UDP Header]

12. Session Layer
    [Data]

13. Presentation Layer
    [Data]

14. Application Layer
    [Data]

In games, the main layer of the OSI model that has seen the most dynamic use of varied protocols is the transport layer. The protocols chosen here have great importance and define the elements of the packet (including reliable vs unreliable variables). How these protocols are chosen could mean the difference between waiting 200ms for a response versus instant feedback.

2.2 – Reliable Transport Protocols

Reliable transport protocols ensure that any data sent between 2 systems, will be received in order and in its entirety. The most popular transport protocol is the Transmission Control Protocol (TCP). It is used by most internet applications, including accessing webpages, file transfer, media streaming etc.

When sending packets across a network, there are several problems that can arise. This includes packets loss, packet corruption, congestion, out of order packets etc. TCP aims to tackle this by defining its protocol header in such a way that every packet will have guaranteed ordered delivery.

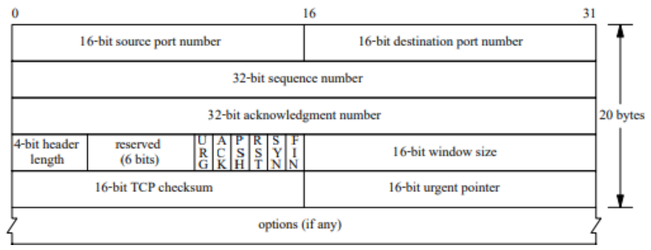


Figure 4: TCP Segment Header [11]

The header is very large compared with unreliable protocols. This is due to the necessary fields to check that the packets have been received and when any should be present if required.

In order to achieve 100% packet delivery, TCP makes the use of acknowledgements. When a packet is sent, or a set of packets, TCP will not keep sending data. Instead, once a comfortable amount of data has been sent, the sender waits for the receiver to reply. If no response is received, a period of which if called a timeout, then the non-acknowledged packets are sent again.

Similarly, the *Checksum* field is a hashed value of the data stored inside the packet. If the receiving device recalculates the hash from the data resulting in a different checksum, then the data is corrupted, and must be resent.

Another issue that can occur with data transfer is unordered packets. The IP protocol will decide the best route that an individual packet should take. As certain routes become more congested, these paths change. As a result, the receiver may be given packets that are not in the order they were sent. This is identified by the *Sequence number* field. If an out of order packet is received, then the receiver 'waits' for the correct one. After a timeout, a request is sent back to the sender using the sequence number to ask for that specific packet again. Only once the correct packet is received does the receiver reply with an acknowledgement packet that confirms the packet is received in the *Acknowledgement Number* field.

While these guards and checks do ultimately guarantee the data will be sent in its entirety, the data transfer is much slower than unreliable protocols.

Techniques are used to tackle this, for example using a sliding window to control the flow of data.

Sliding window definition by IBM:

The TCP sliding window determines the number of unacknowledged bytes, x, that one system can send to another.

Two factors determine the value of x:

- *The size of the send buffer on the sending system.*

- *The size and available space in the receive buffer on the receiving system.* [12]

The number of bytes that can be sent is represented by the *Window Size* field, controlling how much data can be sent at once in a TCP segment. The usage of this field has a big impact of the efficiency of the data transfer.

The sender cannot send more bytes than is available on the receive buffer. Therefore, the sender starts by sending small amounts of data, as the stability of the network is not currently known, nor the size of the receive buffer. At this point, the sender will wait for the acknowledgement packets to be sent. Once the receiver acquires this data, it will calculate a new Window Size field and return an acknowledgement packet, containing this new value.

This Window Size is the amount of space left in the receive buffer. If the receive buffer is full, then the Window Size will be set to 0 bytes, indicating that no more data should be sent. As the receiving application processes and reads the data, the receiver can then send a packet with a new Window Size, equaling the available space in the buffer.

The sender, on receipt of the new packet, now knows the appropriate amount of data to send. The sender can also control the size of this window, for example if no acknowledgement was received due to a lost packet, then the window ize can decrease to limit the processing time wasted on larger packets that may be lost again. The best-case scenario for TCP data transfer is that the receiving application can read data as quickly as it receives it, meaning the window size is almost always the whole size of the receive buffer. The most data that the sender can transmit is as large as its send buffer size, so even if the receiver is currently able to receive a large amount of data, the sender might not be able to send it. However, this sliding window is very effective at limiting the slow data transfer from unoptimized TCP.

2.3 – Unreliable Transport Protocols

Unreliable transport protocols are more simple than reliable transport protocols; the sender will send any data it has, and just continue while it can do so. There is no concept of acknowledgements, rather the receiver will process the data it receives straight away. There is no monitoring of the order of the received packets, or control over the amount of data that can be sent in each time frame, rather the sender sends all the information it can, and the receiver will process as much as it can.

The most common protocol used in games in the User Datagram Protocol (UDP). It is used for applications that are 'lossy'[5], such as live streaming services, Voice Over IP (VOIP) and video conferencing.

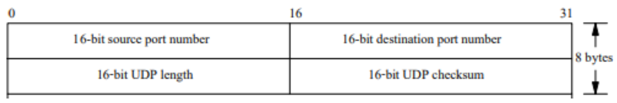


Figure 5: UDP Datagram Header [11]

These headers are much smaller in length. Here, the total size is only 8 bytes, compared with the 20 bytes of a TCP header. There is still a *Checksum* field, to ensure that the data sent can be checked for corruption (without this there may be video glitches in a video stream for example), but otherwise the packet is accepted with no extra validation.

Unreliable protocols are unideal for sensitive file transfer, or any data where order is important. Instead, UDP is very effective in time sensitive scenarios where the smallest amount of latency between client and server is required, at the cost of occasionally losing a packet.

[5] Applications that can handle packet loss.

2.4 – TCP vs UDP in Games

The benefits gained from using reliable protocols like TCP may seem like the best choice for most, if not all, applications across a network. However, the latency gained from using TCP may not result in an ideal experience in many games.

2.41 TCP

In real-time fast paced games, such as FPS, instant feedback to the user is required. For example, the packets containing the information that the user has fired their gun at an enemy should be returned in the shortest time. As the user fires their weapon, visual/audial and gameplay feedback should be instant and any data regarding the positions of other online players should as close to the server at a given time.

With TCP, sending these packets, then waiting for an acknowledgement before processing, requires too much time. If a packet took 100ms to be sent to the server, then a return acknowledgement took another 100ms, that's 200ms between the client sending a packet. The result would be much higher latency, and result in a 'laggy' experience.

The worst case is where a packet is lost. With TCP, the data stream will be stopped, and the receiver waits for the failed packet to be re-sent, before processing any more data. For a time-relevant game, this causes a problem; if the packet updating the position of an item that moved 3 seconds ago was lost, then every other packet is paused. This is wasteful as subsequent packets will contain newer positions anyway, so there is no need to wait for out-of-date information. This entire process could take $2 * Round\ Trip\ Time\ (RTT)$ for the lost packet to be successfully resent, which in a time sensitive game is too long. This is known as head-of-line (HOL) blocking and is one of the biggest issues with TCP in games.

Another problem is the automatic bandwidth control that TCP gives. If packets are dropped, the protocol will view this as an indication of network congestion and reduce the transmission rate[6]. In places where this is not the case, rather it is caused by temporary network errors (for example a router losing connection for a very short time), then the bandwidth will be harmed even if the connection has no issues, limiting the number of packets received and creating more lag. While these games therefore might not have their gameplay suited towards a reliable model, other aspects of the game might.

Any areas that are not as time sensitive, and need to be reliable, could use TCP. For example, in-game chat and player achievements are events where packet loss is more harmful; this is data that cannot afford to be lost due to its importance. Therefore, games are not limited to one protocol, they are able to designate a protocol to each area of the game that is most suited.

[6] Using the sliding window in 1.41

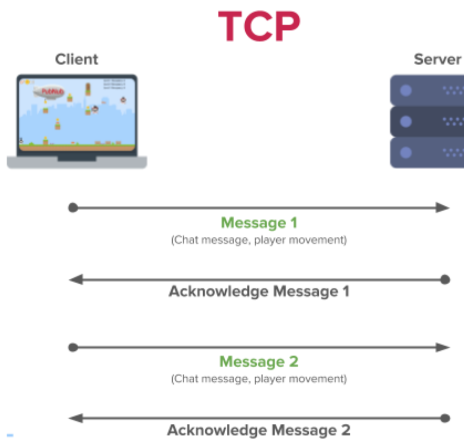


Figure 6: TCP for a client/server model [13]

In non-real-time games, where it is less important if an action will take a little extra time to occur, or games where the tradeoff between a reliable protocol is worth it, TCP can be used. Often, in the case of a turn-based game, this delay can be hidden behind gameplay mechanics, such as animations, audio etc., so if a user was forced to wait 200ms for the character to start moving, it might not even be noticeable. As a result, some games can use TCP for both gameplay and any social mechanics.

2.42 UDP

With UDP, the data is sent immediately, and will improve not only latency, but prevent any unwanted drawbacks from TCP, such as the packet loss problem. As a result, most time-critical games, where the user cannot afford to wait to guarantee packet delivery, use UDP.

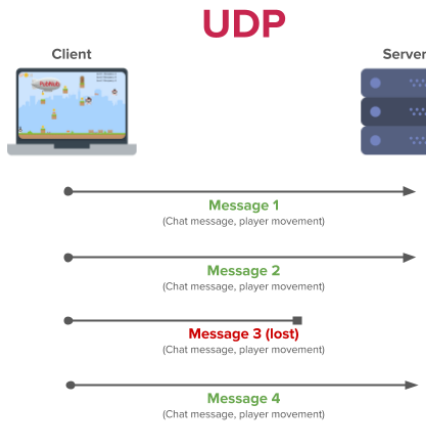

UDP
Client
Server
Message 1
(Chat message, player movement)
Message 2
(Chat message, player movement)
Message 3 (lost)
(Chat message, player movement)
Message 4
(Chat message, player movement)

Figure 7: UDP for a client/server model [13]

With UDP, some packets will eventually get lost in transmission, while others will become corrupted. Due to the checksum contained in the UDP header, any invalid packets can be checked prior to being processed, preventing any potentially harmful data from entering the system. However, due to UDP being unable to retransmit any data, there is nothing that can be done about packets that are lost or corrupted, they must be forgotten. However, modern games use techniques such as State Synchronization, Entity Interpolation, and Input Prediction to help target this.

2.5 – Reliable UDP (RUDP)

RUDP is a reliable protocol built on top of UDP. It contains specific information regarding how the network should handle the data. Its main aim is to provide fast message passing between systems, while also guaranteeing greater reliability than UDP, without the poor latency of TCP.

As a result, RUDP is a protocol that has been investigated for use in games networking and papers [17] have analyzed its use against other popular transport layer protocols (mainly UDP).

RUDP guarantees the delivery of all the packets in order. It extends UDP by adding the following additional features:

1) Acknowledgment of received packets
2) Windowing and flow control
3) Retransmission of lost packets [18]

RUDP is used when both fast packet delivery, and guaranteed packet order delivery, but UDP is too primitive.

There are several methods used to reduce the poor transmit times of a reliable protocol but guarantee the correct ordering of packets, so all clients are receiving the complete and correct data.

2.6 – Techniques to Improve Reliable Data Communication

With a reliable communication channel, there are several techniques used to maximize channel bandwidth using the sliding window.

2.61 The Evolution of Transmission Modes

Since reliable protocols have been used in networks, bandwidth requirements must be considered. For example, if there is a large amount of data to send, but the destination also has data to send, then a system is needed to control the flow of packets between the networks without causing congestion or poor latency.

In modern day home networks, networking capabilities have improved, due to advances in the network infrastructure. This is mainly due to the advances in fiber optic technologies and as such bandwidth has a much greater capacity and allows us to send data using different transmission modes, to improve the efficiency of the network. Modern fiber optic technology and transmission modes can allow for bandwidth speeds of 1Gb/s to customers, allowing great quantities of data to be sent, across multiple channels.

2.611 Simplex Transmission

Simplex transmission is the most primitive form of data transfer. For a network following a simplex mode, a sender can send data only, *never* receive data. It is a *unidirectional* form of communication. In the fiber optic model, the data will be passed across one wire, carrying only one ray of light at a time. It is useful for communications across large distances (e.g., across a country)[7] [14], containing data that is needed in one way only.

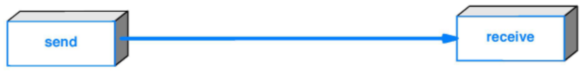


Figure 8: Simplex Transmission [14]

An example of where simplex transmission is used could be:

1. Oil line monitor that sends data about oil flow back to a central location.
2. Interstate trucking scale that sends weight readings back to a monitoring station. [14]

It is used when there are large amounts of data to send, and there is not a system to deal with lost data.

2.612 Half-duplex Transmission

For a reliable data connection to have a method through which it can receive acknowledgements and the state of the packets, the model must be expanded to allow for a duplex transmission.

With half-duplex, each machine can both send *and* receive data. As a result, the data can be sent, and wait for a response from the receiver, confirming packet delivery.

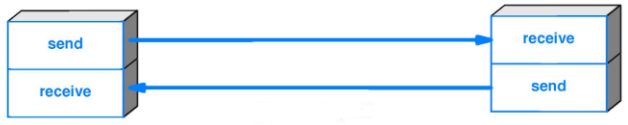


Figure 9: Half-duplex Transmission [14]

Half-duplex does have a limiting factor; it is only capable of sending signals in one direction at a time. Therefore, half-duplex can be used where there is limited network infrastructure, or full bandwidth utilization in unavailable.

An example of where half-duplex transmission is used could be:

1. Voice communication (e.g., walkie talkie) – the signal can be sent, or we can receive a signal, not both [14]

2. Hubs

[7] For example, across a country

Ethernet connections will use half-duplex if only using a single cable.

2.613 Full-duplex Transmission

In a network system, if the required network architecture is in place[8], full-duplex can be used. In a full-duplex transmission, data can travel in *both* directions at once. As one device sends data, it can simultaneously receive it. The requirement of this is that the physical cables involved (ethernet, fiber optic etc.), need at least two wires, to allow for multi-directional transmission of data.

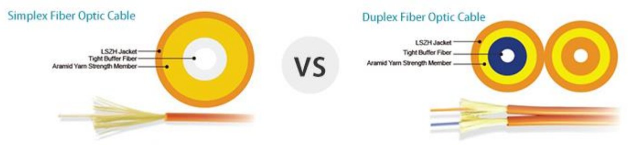


Figure 10: Simplex vs Duplex Fiber Optics [15]

Channels used to be half-duplex, meaning that data could only be send in one direction for a given stream. This resulted in poor latency; in games, if a client needed to send data to the server but also the server had its own data to send, then *either* the client or the server could send data, not both.

Full-duplex is very similar to half-duplex, the main improvement being data transmission rates. For example, assuming there is 100Kb worth of data on a game client, that is ready to send to the server, while the server also had 100Kb of data to send, over a network with bandwidth speeds averaging 100Kb/s [9] .

Half Duplex

The total transmission time over a half-duplex network would be:

(100 / 100) + (100 / 100) = 2 seconds

The data must be sent client to server, then server to client.

Full Duplex

With a full-duplex network, time can be reduced by approximately half:

((100 / 100) + (100 / 100)) / 2 = 1 second

As the data can be sent bi-directionally.

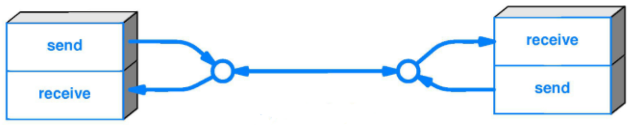


Figure 11: Full-duplex Transmission [14]

An example of where half-duplex transmission is used could be:

1. Telephone communication – most voice communications allow for receiving data (here transmitting sound data through the speak), and transmission (sound inputting into microphone)

2. Ethernet switches

For single and half-duplex cables, only one fiber is required to send data, while full duplex require at least 2 fibers. As a result, full duplex systems are more expensive to implement in a network, the main limiting factor that prevents access to faster networking speeds for lots of networked games users.

As full and half duplex systems are very similar, the main difference being the bandwidth speed, most networked applications (e.g., multiplayer networked games) are programmed in the same way, with certain users gaining additional bandwidth of a full-duplex model where this exists.

In most cases, if a client's bandwidth connection was to fall below a threshold value (regardless of transmission mode), then lots of games deal with this in their own way, such as kicking the player from the game. Having a full-duplex system will certainly limit the chance of this occurring, however.

2.62 Piggybacking

RUDP takes the duplex model even further with piggybacking. With non-optimized reliable protocols, half of the data that will ever be sent is acknowledgement data. For a trustworthy network, where packet loss is minimal, most acknowledgements will never be used; they are simply discarded. As modern bandwidth connections have a much larger capacity[10] than is often required; a server may need to send several 100Kb packets of data to the client but have a maximum bandwidth of 10Mb.

If the network must wait for an acknowledgement before sending the next data frame, most of the bandwidth capacity is left empty, and the RTT[11] is large. The resulting latency, therefore, is sub-optimal, and users will wait longer than needed for a complete packet delivery.

[8] Fiber optic cables support dual way communication, two pairs of cables.
[9] Modern network speeds are much faster than this.

[10] Determined by the sliding window in a reliable protocol.
[11] In a reliable protocol, the total time taken (ms) for a packet to be sent, and the acknowledgement received.

Piggybacking takes advantage of the spare bandwidth by hooking prior acknowledgements of the previously received packet onto the new data packet it wants to send.

Piggybacking protocols follow 3 rules:

1. If station X has both data and acknowledgment to send, it sends a data frame with the ack field containing the sequence number of the frame to be acknowledged.

2. If station X has only an acknowledgment to send, it waits for a finite period of time to see whether a data frame is available to be sent. If a data frame becomes available, then it piggybacks the acknowledgment with it. Otherwise, it sends an ACK frame.

3. If station X has only a data frame to send, it adds the last acknowledgment with it. The station Y discards all duplicate acknowledgments. Alternatively, station X may send the data frame with the ack field containing a bit combination denoting no acknowledgment. [16]

In the client/server model for a game, there is a client X and server Y, and X wants to send data to Y (e.g., an event marked for replication has just occurred, and the data needs to be sent):

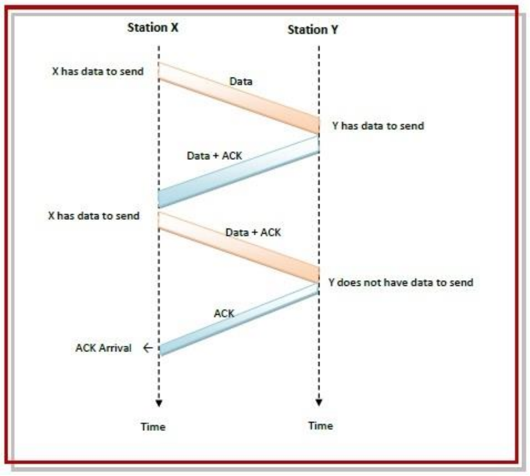


Figure 12: Piggybacking Between a Client and Server [16]

1. X has data to send. It will send this data with the last acknowledgement packet onto it.

2. Y receives the data and the acknowledgement. The acknowledgement is discarded if duplicated. An acknowledgement is generated for the received data. Y waits to see if there is data to be sent to the client. Before the timeout, data is generated. Data and acknowledgement are sent to X.

3. X receives the data, updates all character positions, and generates an acknowledgement. Before the timeout, data is generated. X sends the data and the acknowledgement.

4. Y receives the data and the acknowledgement. The acknowledgement is not discarded as it has not received the acknowledgement of this packet yet. An acknowledgement is generated for the received data. During the timeout, no new data is generated, so the acknowledgement is sent without any data.

5. X receives the acknowledgement, and the process continues.

Piggybacking is a technique that is used across reliable networks, specifically in sliding window protocols. These protocols manipulate piggybacking to utilize the bandwidth as optimally as possible.

2.63 Sliding Window Protocols

In most sliding window protocols, the sender can transmit a given number of frames without blocking (Window Size); the sender can continually fill up the window without waiting for each acknowledgement of every packet. This is known as *pipelining* and aims to improve the throughput of data across a reliable network.

2.631 One-Bit

In one-bit sliding window protocols, there is a Window Size of 1. Only one data frame can be sent at once, an acknowledgement of the previous frame *must* be received before sending the next one.

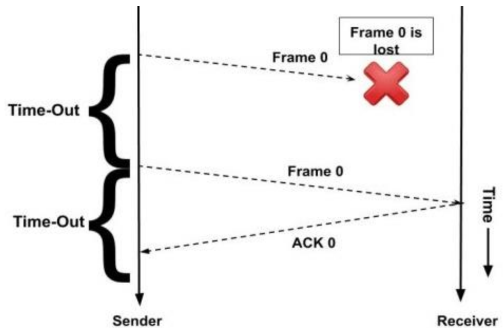


Figure 13: One-Bit Sliding Window Protocol [17]

This protocol is an example of a stop-and-wait protocol: whenever a packet is sent, the sender is unable to perform any other action. It remains idle until either it reaches its timeout duration, at which point the data will be resent, or until it receives the correct acknowledgement, at which point the next data frame can be sent.

2.632 Go Back N

In one-bit protocols, bandwidth utilization is not optimized, only one frame is ever sent at a time. Go Back N improves this; the Window Size is variable, and pipelining is used to increase the throughput.

The problem with pipelining is how to respond in a scenario where a packet is lost. Data is still received, but packet order needs to be guaranteed, so the lost packets are needed before processing any of this data. Therefore, RUDP needs to know what to do with the frames it is still receiving.

If a packet is received that is not expected, no acknowledgment is sent, and subsequent frames are discarded. The sender buffer size is variable, so it continues to send data. The receiver continues to discard frames, waiting for the sender to timeout.

When this timeout is reached, the sender will retransmit all unacknowledged packets. Only when the lost packet is received, does the receiver send the acknowledgement for it, at which point it will accept any further data. Go Back N provides the mechanism to control this:

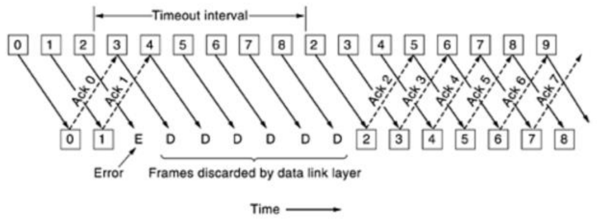


Figure 14: Go Back N Protocol [18]

Here, packet 2 becomes lost. Therefore, when packet 3 is successfully sent to the receiver, it is discarded, as it never received packet 2. The sender, unaware that the packet was ever lost, continues to send data, which is discarded by the receiver. Only when the timeout is reached, does the sender resend the corrupted packet *and* all packets after this.

The disadvantage of this protocol is that because the receiver has a Window Size of 1, every packet after an error, even if successfully transmitted, must be discarded, wasting bandwidth, and increasing the RTT.

2.633 Selective Repeat

In pipelines networks where the Window Size of the receiver is variable, selective repeat can be used. Here, because the receiver can buffer the memory of multiple data frames at once, the system can be more efficient in the way in which it deals with error cases.

Selective repeat makes use of *Negative Acknowledgements (NAK).* These are used to notify either system of an error upon receipt of an unexpected packet. They improve the performance of the system because the error handling no longer relies purely on timeouts; if a NAK is received, then it knows which packet was lost, and so does not need to wait for a timeout of each packet in case it is lost.

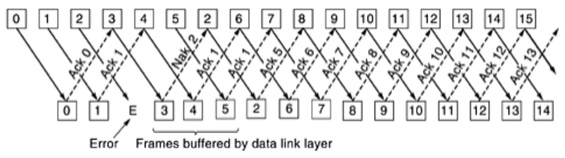


Figure 15: Selective Repeat Protocol [18]

Here, upon receiving packet 3, the receiver sends a NAK back to the sender, as it never received (nor sent the acknowledgement for) packet 2. The sender does not have to wait until it reaches a timeout. Instead, after receiving the NAK for packet 2, it can resend *this packet only*. Meanwhile, the receiver continues to buffer[12] frames 3, 4 and 5. Upon receipt, packet 2 can then be placed prior to all buffered packets, and the data can be processed all at once, in the correct orders.

As a result, this protocol has much higher performance than *Go Back N,* as the correctly received packets can be buffered, rather than discarded.

3 – Game Server Architecture

Data transfer is a major part of the communication between machines over a network, but a decision still needs to be made about which machines to send this information to.

For example, in a multiplayer FPS games containing 12 players, whenever a machine[13] has data packets to send (player inputs for example) it needs to know the final destination(s), and who should receive them. In game networking this can be done from 2 different server models: Peer-to-Peer, and Client-Server.

3.1 – Peer-to-Peer

The first implementations of online networking in games used a peer-to-peer (P2P) model to control the flow of packets across the network.

A P2P model is defined as:

A distributed application architecture that partitions tasks or workloads between peers. Peers are equally privileged, equipotent participants in the application. They are said to form a peer-to-peer network of nodes. [19]

In these models, each client direction communicates with *all* other machines on the network its current state, and any further information. Several games use P2P, such as Street Fighter 5 and StarCraft.

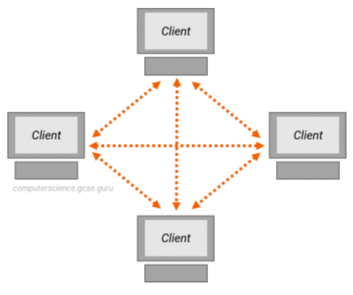


Figure 16: Peer-to-peer Architecture [20]

[12] It can buffer frames up to its maximum buffer size.
[13] From here on *machines* and *clients* refer to one and the same thing

Even without a centralized server, P2P models can still make use of acknowledgements to ensure that the game state is correct, allowing all clients to have a mostly identical world.

3.11 Benefits

This model has several benefits from the client/server model:

- Reduced latency - Because all systems have equal authority, there is no need to validate bad data, i.e., each machine will take the incoming packets and honor it as correct. Because of this, there is reduced time testing the data, and latency is reduced.

- Reduced RTT - As data is transferred directly to other machines without going through a centralized server, our RTT is significantly reduced. Other clients will receive the new states more quickly, resulting in a better synchronized game state between machines.

- Improved flow - With no dependency on the connection to a host machine, clients are free to connect/disconnect at any point, without any interruption to the game.

- Reduced costs - From a game studio's point of view, costs are reduced as no servers are required as all information is purely passed between players.

3.12 Drawbacks

While P2P may seem very advantageous from a client/server model, mainly due to the reduced latency, there are major limitations:

- Cheating – Without any validation from a server, players are free to manipulate their own game state. Due to the equal authority nature of a P2P model, other machines will honor this information. As a result, if a hacker was to inject into the game code that their health was infinite, all other players on the network would obey this, and they would be invincible.

- Scalability – P2P will *mostly* work without many issues will a low player count with average bandwidth states. Each client only needs to send data to a few other machines, so bandwidth should be able to manage. However, as the number of machines is increased, there is an exponential growth in the number of data streams that are sent:

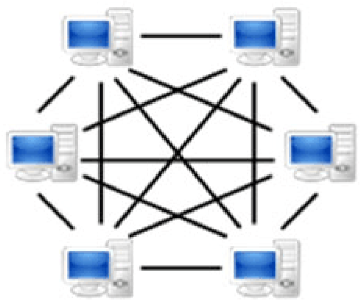


Figure 17: 6 Player Peer-to-peer [21]

In a 2-player game, there are 2 data streams (each client need only send its state to the other).
In a 3-player game, this increases to 6[14] *(+4).*

In a 4-player game, this further increases to 12[15] *(+6).*

In a 5-player game, a larger increase still, to 20[16] *(+8).*

Etc.

From this a given system will require $N * (N-1)$ streams, and an additional machine in the network will require $N * 2$ more streams[17].

As is clear to see, the stream state explodes. If all clients had identical bandwidths, that had very high upload (and download, though to less of an extent) speeds, this would not be much of a challenge. This is not the case, however, and is a disaster for the next issue.

- Asymmetric connections – Most residential routers offer much higher download speeds than upload speeds. For example, being able to download a file from the internet using File Transfer Protocol (FTP), will be achieved in much less time that uploading it. Regarding a P2P architecture, this means that players will have a much greater capacity to receive incoming data then transmit it. As soon as the bandwidth's output maximum is reached, our packets are queued until the packets at the front of the queue have been downloaded on the other systems alleviating space for more.

  As a result, players who have the best upload speed have an outright advantage as they can send their message instantly, which other players are also able to download *instantly* (due to much higher download speeds).

[14] 3 clients send 2 streams
[15] 4 clients send 3 streams
[16] 5 clients send 4 streams
[17] Where N is the current number of clients in the network

This scenario in an FPS could mean that if 2 players, Player 1, and Player 2, both shot each other 'simultaneously' (both users' inputs occurred at the same moment in real-time, *not* according to the game state), then it would become a race in who could upload their packet the quickest.

Player 1 has the better upload speed, so will send their packet first, resulting in Player 2 getting shot (their packet is still on their queue).

According to the other players (and Player 1), they have acknowledged that Player 1 shot Player 2, but in reality, they both shot each other, with Player 2's packet still waiting to be sent.

As each client has complete authority[18] over their game state, Player 2's game state believes that they shot Player 1 (in addition to knowing they were shot themselves).

At this point, Player 2 disagrees with all other clients. This is a case of state desynchronization and occurs when the game state differs on each client's machines. Causes and solutions regarding desynchronization is discussed later in this paper.

Some games can handle these scenarios in different ways. Some games obey the fact that Player 1 should be dead and will try to hide the desynchronization, at the most extreme falsifying the death by use of environmental, to ensure that this occurs on all machines. Others will be stricter and try to avoid any client disagreement by kicking the desynchronized player from the game (StarCraft does this).

3.13 Peer-to-Peer Lockstep

One of the most common network models for a P2P network infrastructure is lockstep. Lockstep works by effectively causing the game to wait until every client acknowledges all other players input commands:

1. The game simulation is split into designated frames.

2. During each frame, client's inputs are transformed into *commands*. At the end of the frame, these commands are sent to all other clients.

3. Only once all clients have received the commands from all other players, they can apply them to their local game simulation during the next time frame.

[18] Also known as a client authentication (client auth) model

The major disadvantage with this model (unlike with a regular P2P model), is that *no* clients game can progress until *every* client has acknowledged everyone else's commands.

As a result, the game will run as fast as the slowest person's latency. If there is a client with a latency of 500ms, then *everyone's* latency will be 500ms and the game will feel slow. Age of Empires is an example of a game that followed a lock step model. [22]

However, with only commands being sent, if a client misses a command from a player, the game immediately breaks down.

In a lockstep model, such as Age of Empires, there are 2 clients, Client 1 and Client 2:

1. During one of the time frames, Client 1 sends a command to charge Client 2's city with a unit. Client 2 commands a unit elsewhere.

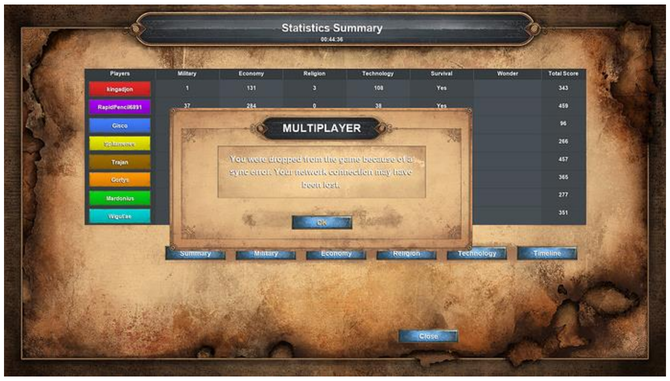
2. At the end of the time frame, the commands are sent to each other. Client 1 acknowledges Client 2's command, but the command for Client 1 is lost on the way to Client 2.

3. All commands are acknowledged, and the game continues on both machines.

4. However, as Client 1's the command was lost on its way to Client 2, Client 2 thinks Client 1 has done nothing, while on Client 1's machines, they are attacking the city.

The command that was lost is not sent to Client 2, and never will be resent. Because commands are the only way the game is manipulated, Client 2's game will always be incorrect, and their game will never be able to reach the correct state again.

In situations like these actions must be made to solve the desynchronization. Players will be flagged whenever their state differs from others, or their messaging is not functioning correctly. These players are, in most cases, kicked from the game:



Players	Military	Economy	Religion	Technology	Survival	Wonder	Total Score
kingadjon	1	131	3	108	Yes		343
RapidPencil6891	37	284	0	38	Yes		459
Gisco							96
[illegible]							266
Trajan							457
Gortys							365
Mardonius							277
Wigulfae							351

You were dropped from the game because of a sync error. Your network connection may have been lost.

Figure 18: Age of Empires Desynchronization [23]

3.131 Determinism
In addition to desynchronization prevention, the game must also be deterministic. For every given command, the same outcome should occur on all machines.

This is because, in place of player positions, state etc. being sent, *only* the command is. Therefore, this command *must* change each game state in the same way.

Games that are deterministic can still have random elements within them. To solve this, every frame of the game generates unique identifier, a seed, that will be used for the random number generator.

Then, each machine, will generate a number using the same seed, and be able to run the game in the same way, and consistency is maintained across machines. This guarantees a pseudo random element to the game.

There are many options to use as a seed. Frames passed since the start of the program could be one. Another example is found in Rapture – World Conquest:

"Right from the start we made sure that our simulation used an independent random number generator from the rest of the game. This is because the game might be running at different framerates on different machines, so any random number generation in our rendering code (a particle emitter for example) will be called a different number of times across machines. Obviously, the random number seed used by the simulator needs to be agreed upon by all machines and we did this by generating a seed from a checksum of the shared launch settings." [24]

3.2 – Client/Server
Considering the negatives regarding the implementation of a P2P networked game, it is much more common for most modern games use the client/server model.

A Client/Server model is defined as:

A distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. [25]

Here, each client does not communicate with all other machines as a P2P model does. Instead, there is a centralized machine, called a server, which receives all the incoming data from clients, updates its own game state, and sends this state back to the clients.

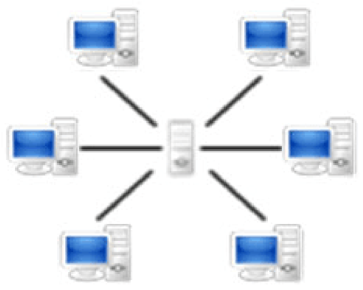


Figure 18: Client/Server Model [20]

Every server update, the server will be receiving data from *all* clients, update the game state, and then retransmit the new game state back. Due to this requirement, the server needs to have: a very well performing network, both in upload and download, so that it can reduce the latency as best as possible between updates; and be powerful enough that it can process and update multiple game states before the next tick.

As a result, this model is much more scalable than a P2P network and is used by almost every game that has multiple players.

As a result, clients upload speeds is no longer a limitation[19]. They only are required ever to send their packets to one other IP address. In addition, each client's bandwidth download *should* throttle less, as it only receives the data from one source, rather than a stream for every instance of a player, though this depends on frequency of the server update.

In some scenarios (mainly due to reducing server costs), a client may act as a client *and* a server. They hold the true game state, and rather than have a centralized server dedicated to gameplay logic only, the instance of the game on this machine is the one that holds authority. *Activision Blizzard* has implemented this in many of their *Call of Duty* games, for example *Call of Duty Modern Warfare 2*.

While methods exist to limit the drawbacks of this approach, for example algorithms that determine and choose the client with the best network connection, it is inferior to a *dedicated* server:

- Even if the client with the 'best' network is chosen, all players are limited to the upload speed of this client. In addition to this, the client's location may be much further away than a centralized one, resulting in very high latency for some users.

- If the host disconnects, there is no longer a server. In these scenarios, it is up to the developers how to handle the game. One option is to end the game altogether and

[19] Except in extreme circumstances

kick players back to the lobby, another technique (used by Activision Blizzard*)* is called host migration [23]. This is where the game is paused and the game will try to decide a new host and continue the game from the same state it left before the previous host disconnected.

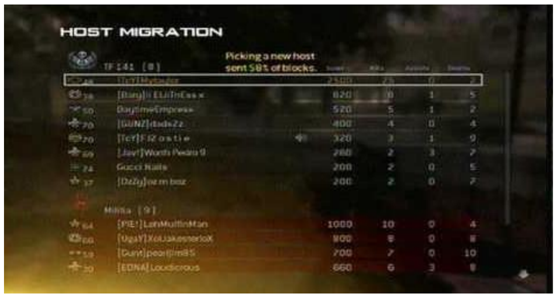


Figure 19: *Call of Duty Modern Warfare 2* - Host Migration [26]

- As soon as a client is allowed to act as a server, they are granted authority over their (and so everyone else's) game state. Therefore, any client with malicious intent has the freedom to inject code into the game, and all other clients will *have* to honor it. Often this is in the form of a form of an 'aimbot', that will lock the player's sight/vehicle etc. to a point on screen, allowing to eliminate everyone in the lobby with ease.

3.21 Benefits

This model has several benefits from the P2P model:

- Network bandwidth – With each client only needing to send its inputs to one location, the concern of a client's upload speed is almost eliminated.

- Server authentication – Because the server knows it will receive data from *all* clients, it can validate absolutely everything that tries to update the game. If a hacker attempted to set their health to infinity, they do not have authority, so it must go to the server. Then, the server will validate the game state from this client. It will analyze newly updated fields and deny any that look suspicious, making cheating much more difficult in the client/server model. There are several anti-cheat software's used in large multiplayer games, for example *BattleEye* [27] (used by *PUBG*, *Fortnite*, *ARMA* etc.), each with their own validation models that aim to deny any 'bad' data. P2P games *can* have anti-cheat systems in place, though they are harder to implement and often not as successful.

- Client hosted server – A client/server model does not require a dedicated server, which can be very expensive for studios. A client may act as the server (they are known as the *host*), eliminating any costs incurred with a dedicated server (though the issues regarding this are previously discussed).

3.22 Drawbacks

Client/server is without a doubt the most reliable network model that guarantees data validation and removes a lot of strain on bandwidth resources[20]. However, this comes at a cost:

- Latency – Unlike P2P, where data can be transmitted directly to all other machines, it must first be sent to the server, validated, and then sent back. This *at least* doubles the RTT for the updated game state to be sent back to the client than with a P2P model. If network conditions are sub-optimal, for example internet traffic, then this must be traversed twice. The longer the latency, the less likely our client prediction will be correct, resulting in rubber banding and lag [28].

- Server monitoring & maintenance – In many cases, the number of clients attempting to connect to our server at once. For example, only enough resources (hardware, bandwidth) are given to a server to accommodate 1000ccu (Concurrent Users). If the actual number of players trying to connect was much higher, e.g., 5000ccu., the server is going to be throttled. As a result, *everyone* in the game is going to experience network delay, as there are not enough resources to accommodate for network requests.

  This often occurs to large multiplayer games on launch, where the servers are not yet configured to handle the capacity of the player base. *World of Warcraft* is an example that, on release (in addition to the launch days of its DLC) has had server issues, and players across the globe, no matter server location, have had unresponsive network conditions. This is because it is difficult to estimate an appropriate number of servers/resources until the user count can be analyzed and resources given to accommodate this. Often, studios *know* their servers will perform poorly at the release of a game, but it is more economically viable to record the number of players trying to get into games after release, and then spend money on additional resources, rather than invest too much earlier, only for it to potentially be a waste.

5. Server costs – In an ideal world, every game could host an unlimited number of servers and allocate as many resources as required, and there would be no server related issues for the client. However, servers are expensive to run, especially ones that can host many users at once. Often studios will purchase *just* enough

[20] For clients

resources to keep their player base from issues and adjust if the CCU gets larger/smaller.

Even with the costs of running servers and the issues related to a client/server model, they are the by far the most popular choice for network architectures. Some games do still implement P2P networks, but these will be with very low player counts per game where the latency induced with a lockstep model can be hidden with animations/sound etc. For games requiring more than a few players, P2P would be almost unplayable unless *all* clients had extremely fast broadband, which is impossible to guarantee.

4 –Client/Server Networking Model in Games

4.1 Absolute Server Authority

Some early examples of client/server networked games felt very laggy and unresponsive; having pressed an input, there would be a short delay before any client-side state was updated. This was very apparent in the original Quake, where inputs would always feel delayed. As Glenn Fielder claims:

"In the original Quake you felt the latency between your computer and the server. Press forward and you'd wait however long it took for packets to travel to the server and back to you before you'd actually start moving. Press fire and you wait for that same delay before shooting." [29]

In these games, the client would do very little regarding game state manipulation and processed little code on its own. Instead, at every given tick, client inputs would be sent to the server, the server would simulate the new game state, and would return this state. Only upon receiving this should the client's world update. As a result, depending on latency and server conditions, there would be a consistent delay between client's key/controller presses, and the result of that on screen.

4.2 Client-Side Prediction

To counter the high latency feel of an absolute server model, modern games use a system called *client-side prediction*. This works as follows:

1. On input presses, the client predicts the new game state. For example, a forward's movement would be processed on the client machine (even with lack of authority).

2. The server would receive these inputs (and all the inputs from other machines) and add them to a buffer called a *jitter buffer*. Every frame, the server grabs the oldest inputs from its jitter buffer and simulates the new world state. This state would be sent back to the clients.

3. Each client, on receiving this new state; would compare it against its own previously predicted state. For any values that do not pass a given threshold, the client will take the server's state and correct itself.

While more code is running on the client, the delay is eliminated, the inputs are processed immediately (as it would in a non-networked game).[21]

4.21 The Jitter Buffer

The server needs to maintain a *jitter buffer*. Each frame, clients send their inputs to the server, where they are added to the jitter buffer. The server will remove the oldest inputs from its buffer and simulate them. This simulated state is then sent back to the clients.

The length of this buffer is very important and is vital for effective client-side prediction, usually it is several frames long.

4.3 Client-Side Correction

One difficult problem generated from of client-side prediction is that client state may differ from the server. This is due to either:

1. External factors that cannot be predicted. This mainly consists of the actions of other clients (e.g., being shot by another player). These are impossible to know until the client receives this data from the server.

2. Client-side prediction produces inaccuracies which build up over time and eventually do not pass designated *threshold* values. This is normally for client position, where their predicted position and server position are 'too different' and must be corrected.

In case 2, the client should determine if it should correct itself using *threshold* values, and these are often difficult to calibrate.

For example, for a given player model that has movement around the world, what value (in metric terms) should be decided upon?

- A threshold too low, and the client will be too regularly corrected by the server, resulting in lots of snappy position adjustments on the client's machine.

- A threshold too high, and the client is given too much freedom. In fast-paced games like PUBG and Apex Legends, this may mean that a client who thinks they're behind cover, is actually in the open. As all other clients take the server's state, which says the player is in the open, they can shoot the other client who still thinks they're in cover (this is a very common issue in these types of games).

This threshold is not limited only to a measurable value as in the example above. It could be any form of condition that checks what a client *thinks* has happened, and what the server says *actually* happened. For example, a client might predict that it shot a player, but the server says (for whatever reason) that this did not. At this point the game states are more than a simple position value out of sync, and game logic must control what should happen, in this case the player the client thought they shot should respawn on their machine, and the kill should be removed altogether.

The example given above is an extreme case, and for 'more important' scenarios like a player eliminating another, the client can be prevented from predicting this altogether. This can be extended even further, by making use of TCP packets for the game data that is more impactful. With a solution like this, not only will the client always be in sync with the server regarding important events, but due to the guaranteed order of packets, all clients will receive this information. Without this approach, game state correction should be hidden through other means (environmental kills for players that *should* be dead etc.).
For more regularly occurring events, like movement, shooting etc., TCP *should not* be used (due the very noticeable latency) and client-side prediction *should* be used, to make the game feel less sluggish. As such thresholds must be used. Many games are different, and therefore these values should be carefully considered on a game-by-game basis.

For slow paced games, such as platformers, often the maximum difference between position values between a server tick is low. Here, the threshold can be set to a relatively small value, because the difference in our client's prediction is going to be small. In a game where these differences can be much higher (due to higher speeds), such as a racing game, the threshold values should be calibrated to larger values. If the small thresholds from an online platformer were applied to an online racing game, all clients would constantly 'need' correcting, resulting in cars constantly snapping towards the server's position, resulting in poor player experience.

4.31 The Input Buffer
Client-side prediction is a solution to the unresponsive gameplay feel, however, the current solution poses problems.

Imagine the is a client/server with total RTT of 200ms. If the client sends its inputs to the server, then it will be 200ms before it gets the correct position back. However, this will be the correct position from 200ms ago (at the same time the client sent its inputs). If the client was to listen to the server with no further adjustments, then they will be snapped back to their 'correct' location for the simulated state at the time the inputs were sent.

All the inputs between the corrected frame and current frame are lost, resulting in the client-side prediction is completely undone, with the game feeling exactly as it did without it.

The current client-side correction has an inherent problem. The state that is compared against is in the past (at least an RTT out of date), but no inputs are stored between these frames. When flagged for correction, the client will be corrected towards an old state. However, having been corrected to this old state, there is no current mechanism whereby the inputs can be re-simulated for the frames between the corrected and the client's current frame. Therefore, our client must store its inputs every frame, in a buffer known as an *input buffer*.

This buffer contains past character state and input. This buffer can store up to N inputs, and upon reaching its capacity, the oldest in the buffer are discarded.

The new client/server flow is as follows:

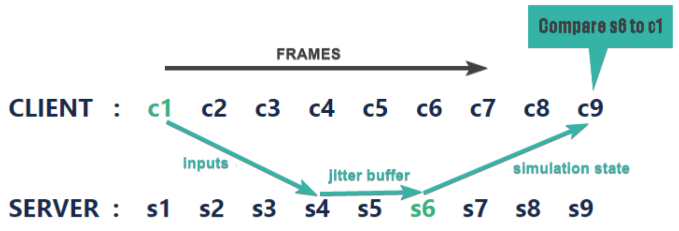

Compare s6 to c1
FRAMES
CLIENT : c1 c2 c3 c4 c5 c6 c7 c8 c9
inputs
jitter buffer
simulation state
SERVER : s1 s2 s3 s4 s5 s6 s7 s8 s9

Figure 20: The Modern Network Model [30]

1. Every frame, the client will receive the player's inputs, and simulate them using client-side prediction. These inputs, in addition to the newly simulated state, are cached. The inputs alone are then sent to the server.

2. These inputs take half the ping time to arrive to the server, where they are pushed onto the jitter buffer. Each frame the server will pull the oldest inputs off the jitter buffer and simulate their outcome. The state of this simulation is then sent back to the client.

3. This new simulation state takes the remaining half ping time to arrive back at the client. The client then compares the received server simulation state for these inputs with the simulation state it cached when it locally processed those inputs.

If a correction is required:

[21] The RTT has been reduced to 0ms

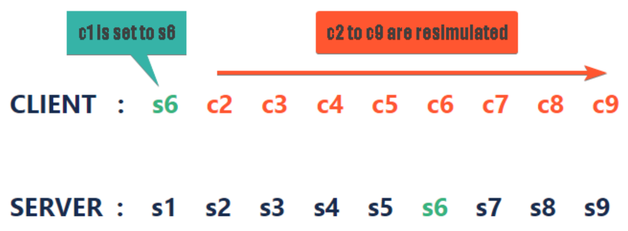

c1 is set to s6
c2 to c9 are resimulated
CLIENT : s6 c2 c3 c4 c5 c6 c7 c8 c9
SERVER : s1 s2 s3 s4 s5 s6 s7 s8 s9

Figure 21: Client-Side Correction (Input Buffer) [30]

1. The client discards any buffered state that is older than the corrected state.[22]

2. The client applies the correct server state, s6.

3. Because the inputs between frames c2 – c9 are buffered in the input buffer, frames c2-c9 are re-simulated.

This results in the client being able to rewind its own local simulation, apply the corrected state from the server, and replay the last N frames of inputs that are stored in its input buffer.

With the input buffer, the appearance of 0 latency can be faked, with our client and server almost always being in sync, with occasional cases where the client must be corrected.

According to Tim Sweeny, using this method, correction is almost always due to external factors that cannot be predicted:

"Nearly all the time, the client movement simulation exactly mirrors the client movement carried out by the server, so the client's position is seldom corrected. Only in the rare case, such as a player getting hit by a rocket, or bumping into an enemy, will the client's location need to be corrected." [31]

4.4 State Synchronization

Even with client-side prediction functioning correctly, there is one major problem. Per client, the game will feel much smoother with the illusion of no latency, but local prediction means that clients will *always* be out of sync with the server. Clients predicting locally will be ahead of where they actually are on the server, while foreign clients will be behind (other client's inputs still take a RTT to be sent to everyone else, but they can be locally simulated *immediately*).

There are 3 factors that influence the amount by which clients are behind/ahead of other players:

1. Framerate – The lower the framerate, the fewer frames pass between the client sending the predicted game state and receiving the corrected game state, resulting in larger inaccuracies.

2. Latency – The higher the ping, the longer it takes for each client to receive their corrected state, and each player will be further ahead that they are on the server.

3. Jitter Buffer Size – This buffer receives inputs for N frames, before simulating the new game state and transmitting it back to clients. The higher N is, the more frames will pass before the client receives the new state.

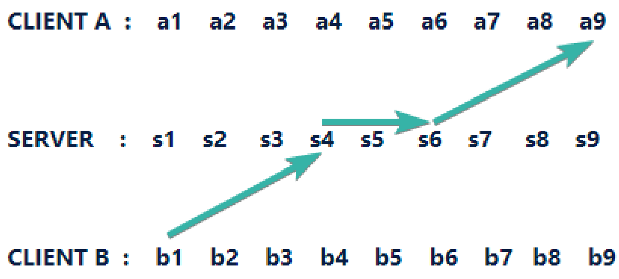

CLIENT A : a1 a2 a3 a4 a5 a6 a7 a8 a9
SERVER : s1 s2 s3 s4 s5 s6 s7 s8 s9
CLIENT B : b1 b2 b3 b4 b5 b6 b7 b8 b9

Figure 22: Desynchronization [30]

For example, assuming:

- Framerate = 60fps
- Latency = 100ms
- Jitter Buffer Size = 2frames

$$Desync\ Distance = Distance\ per\ frame * Frames\ per\ sync$$
$$Distance\ per\ frame = \frac{player\ speed(m/s)}{framerate(frames/s)}$$
$$Frames\ per\ sync = (framerate(frames/s) * latency(s)) + jitterbuffer\ size$$

As the inputs and updated game state occur at s6 on the server, every client is:

$$Frames\ per\ sync = \left(60 * \frac{\frac{100}{2}}{1000}\right) + 2 = 5 frames$$

Ahead of themselves compared with the server. This updated game state still needs to be send to other players (to synchronize their games) in addition to the local client (for local correction if required). The number of frames then increases to:

$$Frames\ per\ sync = \left(60 * \frac{100}{1000}\right) + 2 = 8 frames$$

In FPS games like PUBG, common run speeds are ~6m/s. Using the previous example (the *best-case* scenario for desync is 8 frames), clients will be:

$$Distance\ per\ frame = \frac{6}{60} = 0.1m$$

$= 0.1 * 8 = 0.8m$ minimum out of position.

[22] I.e., Anything older than s6

This results in situations like below:

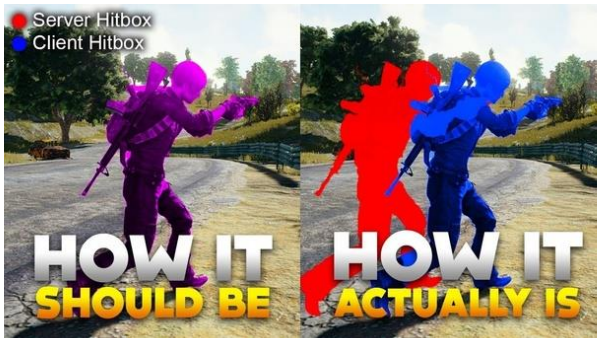


Figure 23: Desynchronization in PUBG [32]

The result is that players who believe they're behind cover (client hitbox), are not at all (server hitbox), and are shot by another client who only considers their server position.

This problem of game desynchronization is further exaggerated in games where position between frames can be much larger than a regular character-based game.

Take a racing game, for instance, where vehicles can travel up to 40m/s. Using the same network conditions, the inaccuracy now increases to:

$$Distance\ per\ frame = \frac{40}{60} = 0.66m$$

$$= 0.66 * 8 = 5.3m \text{ minimum out of position.}$$

Distances like these are much more obvious to the player. Firing a rocket at a moving vehicle now won't just hit it slightly differently than was expected, it will 'miss' it altogether.

4.41 Lag Compensation

Lag compensation is another tool available to games to tackle desynchronization, especially where client-side prediction is used. In a client/server model, clients will always be seeing other players at their positions in the past, caused by input prediction and latency.

For example, there are 2 clients: Client 1 is running in a field, and Client 2 is about to shoot Client 1.
In this frame, Client 2 has a direct shot on Client 1, and shoots. These inputs are sent to the server. However, without lag compensation, this might not register as a hit at all. The reason is latency.

Client 2 will shoot Client 1, but the outcome of this input cannot be confirmed until the server simulates the packet and updates its world. So, the input packet is sent to the server before anything on Client 2 can happen. Games will often still produce the visual and audial effects at this point (muzzle flash, explosion etc.), but the register of the inputs cannot happen until the authority (here, the server) says so.

The input packet is sent over the network, but it takes time ($\frac{RTT}{2}$). Meanwhile, the server continues to simulate the game world. When the packet reaches the server, it checks to see if the bullet hit Client 1, but at the current game state, Client 1 has moved, along with their hitbox, and the bullet, in the current simulated state, should not be a hit.

There is a difference between game states, known as *Client* Delay, and is calculated as:

$$\text{Client Delay(ms)} = \frac{\text{Latency(ms)}}{2} + \text{InterpolationDelay(ms)}$$

This is where lag compensation is used. According to Valve's Source net code:

The lag compensation system keeps a history of all recent player positions for one second. If a user command is executed, the server estimates at what time the command was created as follows: [30]

$$Command\ Execution\ Time = Current\ Server\ Time - Packet\ Latency - Client\ View\ Interpolation$$

With this knowledge, all players then have their positions adjusted (along with animations, state etc.) to the point at which the client initially sent their input. The detection of whether Client 1 was shot can then happen correctly:

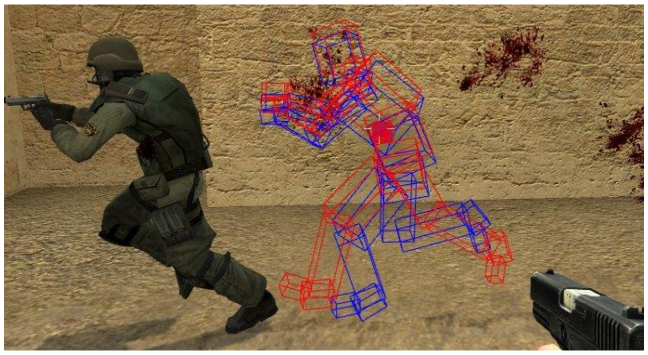


Figure 24: Lag Compensation in CSGO [33]

Whether or not it was a hit, all players are rewound back to their server time positions, and the game is updated as normal, with the result of the shot being set back to all clients.

Lag compensation can be used in conjunction with client-side prediction. Ultimately the prediction is only there to make the gameplay feel more responsive, but the server knows all past positions and current inputs of each player, so it *always* has the ability to rewind and re-simulate.

The compensation needs to be fine-tuned:

- Over tuned, and it will result in players being rewound too far, allowing clients to shoot behind targets.

- Under tuned, and players are not rewound far enough, requiring players to lead their targets.

Example of lag compensation, using Client 1 and Client 2:

1. Client 2 has Client 1 in their crosshairs, they shoot.

2. This action is added to their inputs and sent in a packet to the server.

3. Meanwhile, the server simulates the game.

4. The server receives Client 2's packet, which took $(\frac{Client\ 2\ Latency}{2})$ ms to arrive

5. The server calculates the time that Client 2 shot and searches its memory buffer to grab this state.

6. The server rewinds all players back to their positions at this state, and then carries out the collision detection for the shot. It is a hit.

7. The server unwinds, and continues to simulate, returning the outcome of the inputs to all clients.

8. This packet takes the remaining $(\frac{Client\ x\ Latency}{2})$ to return to each client respectively. Client 1 is shot, and Client 2 earns a kill.

With this system, no matter the latency of the client, the server will always rewind time to the appropriate positions at which those inputs were sent.

Without lag compensation, clients would have to lead their targets to match with the state of the client when the packet was reached. In reality this would be very difficult due to the varying nature of latency and internet traffic and make for poor user experience.

This still does not perfectly solve desynchronization. Client 1 may have gone into cover at the point the packet that they are shot finally reaches them. The time it took the packet to arrive is equal to:

$$\text{Receiving Client Delay(ms)} = \frac{\text{Client x Latency(ms)}}{2} + \frac{\text{Receiving Client Latency(ms)}}{2} + \text{TimeSinceLastTick(ms)}$$

However, this is unavoidable. As previously mentioned, there are always *external factors that cannot be predicted*. Without returning to a lock-step model, there is no way of preventing clients from being out of sync.

4.42 Improving Latency

When the internet was designed, it created and followed the idea of *network neutrality*. This is:
The principle that Internet service providers (ISPs) must treat all Internet communications equally, and not discriminate or charge differently based on user, content, website, platform, application, type of equipment, source address, destination address, or method of communication. [34]

This, in theory, is a good idea, and processes packets on a '*best-effort delivery*' basis. However, this methodology existed before differences between the types of internet packets were really established. For example, the internet still treats a packet for a real time application the same way it does when a webpage is requested. In reality, some packets (such as those for users watching a stream or playing an online game) are more performance reliant than those for a file transfer.

As a result, some larger game studios, like Riot Games, build their own private networks that can control the importance of network delivery. Such examples include Riot Direct [35], that handle all internet traffic between Riot's servers.

Other developers do not have the means to afford a personal private network. Network Next is an example of a company who is trying to alleviate this issue by utilizing the spare capacity of existing private networks to other developers.

According to Glenn Fiedler, founder of Network Next:

The end result (of best-effort delivery) is the inconsistent network performance your players get when they play your game online. [36]

Esports is an area where performant data delivery is especially important. Electronic Sports League (ESL) is one of the largest Esports organizers in the world, and they have partnered with Network Next to try and combat the issue of poor latency across the internet [37].

As each client *must* communicate with the server, its physical location is very important. If our central server is very far away (such as clients from the US connecting to a European server), our latency will be very high, and those players will suffer lag. To combat this, games studios may host several servers in different locations. When a client connects to the main game server, they are redirected to the best suited server at the time. In most cases, this would be the geographically nearest, however there may be other external factors that determine this, e.g., internet traffic, server status etc. These methods aim to give the player the best multiplayer experience by reducing the latency between client/server updates.

4.5 Update Rates

4.51 Server Update Rate (Tick Rate)

This is the fixed server update interval using the inputs from its jitter buffer. A server updating at Xhz is capable of sending X packets per second. The higher the value, the more accurate the server simulation will be.

This value can vary greatly and should be determined based on the type of game. For an FPS, it is recommended that this is at least (if not higher) higher than the client's update rate. For example, Valve's Source engine runs its tick rate at 66hz.

4.52 Server Broadcast Rate

This is the frequency at which the server sends the new game state based on its inputs since the last broadcast. This needs to be chosen carefully:

- If too low, then the server is sending the updated game state too infrequently, as a result there will be higher latency and rubber banding (due to increased client-side correction).

- If too high, the server is sending too many unnecessary packets. These are more likely to be lost/dropped and use too much bandwidth that is better used elsewhere.

It is often much lower than the tick rate, as the tick rate is simply how many times our server is updating the physics and would, in most cases, be way too high.

4.53 Client Update Rate

The frequency at which clients accept the new game state from the server. This *should* be the same as the server broadcast rate:

- If lower, then the server is sending packets that are discarded, e.g., "*if the client update rate is 20, and the server (broadcast) is 64, the client might as well be playing on a 20 tick server.*" [38].

- If higher, then the client will be updating its state using multiple instances of the same state. As a result, the server is sending unnecessary packets.

4.54 Client Broadcast Rate

The frequency at which clients send their own inputs to the server. This *should* be the same as the server tick rate:

- If higher, then the client is sending more than 1 packet per tick update of the server. Therefore, these packets are dropped on the server as it does not have the capacity to process them.

- If lower, then the server will be updating, but with no more inputs received. This is a waste of processing time and would has the same effect if no inputs during this frame were sent at all.

4.55 The Relationship Between Client and Server Updates

The best way to modify the transmission rates to create a well performing network is:

$$SERVER\ UPDATE\ RATE == CLIENT\ BROADCAST\ RATE$$

$$CLIENT\ UPDATE\ RATE == SERVER\ BROADCAST\ RATE$$

$$SERVER\ UPDATE\ RATE(hz) \gg CLIENT\ UPDATE\ RATE(hz)$$[23]

If the statements above are kept true, then the least packets should be dropped in transmission, and the game state synchronization is improved.

4.56 Modifying the Tick Rate

Of the rates mentioned, the one which is most utilized to improve synchronization is modifying the server tick rate. The reason for this is regardless of how many times the server sends the state to the client, its own simulation needs to be accurate. As the server has authority, all client's experiences are going to result on the tick rate. This value needs fine tuning to have the best impact on the game.

- A rate too low, and the server simulation is inaccurate. Therefore, the new state that is calculated is less physically correct than it could be. This could result in bullets being less accurate/movement positions slightly off etc. In very low cases it may cause *rubber banding*[24].

- A rate too high may become very computationally expensive and require large expensive servers to deal with the high requirement. Most games studios do not have the means (nor need) to purchase such systems.

For games that do not require such fidelity within their physics engine, such as a turn-based strategy game, high tick rates are overkill. For fast paced FPS games (especially for competitive games), it is more advantageous for this to be a large value:

[23] Hereon, assume that 'server update rate' will always equal 'client broadcast rate', and 'client update rate' will always equal 'server broadcast rate'
[24] The client is desynchronized, and is re-corrected towards their 'correct' position

Tick rate for games like first-person shooters is often between 120 ticks per second (such is Valorant's case), 60 ticks per second (in games like Counter-Strike: Global Offensive and Overwatch), 30 ticks per second (like in Fortnite and Battlefield V's console edition) and 20 ticks per second (such are the polemic cases of Call of Duty: Modern Warfare, Call of Duty: Warzone and Apex Legends). [39]

CSGO allows users to set their tick rate in offline games, which can result in running simulations like this:

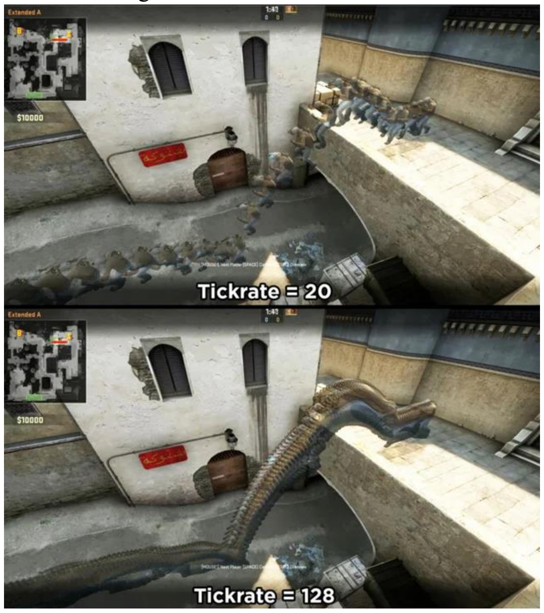


Figure 25: CSGO Tick Rate [40]

In FPS games that have a lower tick rate, calculations will be less accurate, and will result in situations with shot around corners, inaccurate bullet travel etc.

For example, *Valorant*, running at 120hz, will be able to run 1 physics step every 8.3ms, while *Call of Duty: Warzone*, running at 20hz, will only be able to run its physics every 50ms. Therefore, when the server broadcasts this new state back, *Valorant* will have a much more accurate game state than *Call of Duty: Warzone*. The trade off here is that *Valorant* will require a more powerful server infrastructure (which is expensive).

Activision Blizzard's choice to host servers with a tick rate to only 20hz for *Call of Duty: Warzone*, has caused lots of discontent among players:

"There are other examples of tick having an effect on play, but in general it manifests itself as an inaccurate portrayal of what you experienced being displayed in the killcam of the person who killed you, due to lag compensation." [41]

In a purely deterministic physics engine, the tick rate would be must less impactful, the game state at any given time could simply be calculated. However, most physics engines are non-deterministic, so there will always be variance in positions/velocities/etc.

4.57 Modifying the Client Update Rate

Another approach to improving desynchronization is to modify the client update rate (as well as the server broadcast rate). This approach is more focused on targeting the range of bandwidth speeds that clients have.

- A rate too low, and there will be noticeable latency. A rate of 5hz, for example, means that the client will receive the updated game state at least 200ms after inputs (worst case), in addition to ping time. The updates are so infrequent that clients are much more likely to have game states that disagree with the server, and correction will be required much more.

- A rate too high may require too much bandwidth, even with the use of UDP packets. The clients are told too frequently to send their game state (and inputs) to the server. A scenario is then reached, very similarly to a P2P network, where our clients who have asynchronous connections experience poor gameplay. The requirement to send their game state, for example, at a frequency of 40hz, overloads the upload bandwidth capacity, causing a queue to form. This causes the queue to get filled with older and older packets. As a result, when clients receive the updated game state from the server, they are much more likely to be corrected, due to these out-of-date states, and clients will also experience rubber banding.

Therefore, anywhere between 10hz and 30hz are commonly found in multiplayer games:

In whitepapers, Microsoft used to recommend that game developers choose an update interval between 10Hz and 30Hz.[42]

Values ranging these frequencies are high enough that clients are updated just enough to avoid client-side correction *most* of the time, while being low enough to cover the network capabilities of most users.

4.571 Lowering the Client Update rate for poor network conditions

Some users, on poor bandwidth conditions, are going to face the problems caused by their bandwidth download speeds. To combat this, the client update rate could be lowered, e.g., < 10hz, so that it increases the chance that all clients are receiving the game state simultaneously and solves the issue of clients receiving out-of-date packets (at least, as far as the network is concerned).

However, if the client update rate was lowered to such an amount to allow for these conditions, the game would be updated so

infrequently[25] that the chance of client-side correction for *all* players is greatly increased, due to the increased number of frames passed between updates (client-side correction can only *predict*).
Low update rates give clients too much power, and their client-side prediction is more likely to be inaccurate of the actual game state. Then, upon receiving the actual game state, there is a much greater than they should correct themselves, resulting in rubber banding (due to external factors affecting positions), or other more important game effects that the client must be forced to handle (even an inaccuracy in movement could cause this).

Such a situation, for an FPS, would be:

- Server Tick Rate/Client Broadcast Rate: 60hz
- Client Update Rate/ Server Broadcast Rate: 5hz

- Client 1 Bandwidth Conditions:
- Upload – Strong
- Download – Strong

- Client 2 Bandwidth Conditions:
- Upload – Weak
- Download – Average

1) Client 1 is running into a building to hide from Client 2 who is trying to shoot them.

2) Client 2, who can see Client 1 running into a building, shoots Client 1, and these inputs are sent to the server.

3) The server receives these inputs and buffers them in its jitter buffer but does not update until the large time step has passed.

4) Meanwhile, Client 1 believes they made it into the building due to client-side prediction.

5) The server, upon reaching its timestep, calculated the new game state across all the inputs it received, where Client 1 has been shot, and sends the new state back to all clients.

6) These states are received by both clients (roughly) simultaneously (the tick rate is so low that bandwidth conditions are all covered for[26])

7) Client 1, has had (in the worst case):

$$Frames\ Passed\ (Worst\ Case)$$

$$= \left(\frac{Game\ Update\ Frequency}{Server\ Update\ Frequency}\right)$$

$$= \frac{60}{5} = 12\ Frames$$

Passed since Client 2's inputs were sent. Client 1 updates, realizes that they have been shot, and client-side correction will force the client to listen to the server.

[25] For *everyone*
[26] Though, of course, latency may vary still

This highlights the issue surrounding low update frequencies. As far as Client 1 is concerned, they were in cover so there was no reason why they should have been killed. Even with strong bandwidth, because the network system has been calibrated to accommodate for low bandwidth players, it makes no difference. Most of the time this would be somewhat unnoticeable (for example running across a field), but when small distances matter, the issues become extremely apparent.

4.572 Increasing the Client Update rate for strong network conditions

Contrary to above, another approach that can be taken is to update the client at a much higher frequency, e.g., > 30hz. With values like these, the client is likely to be receiving game state from the server at more than half of their game frames.

Assuming that the server can deal with the increase in broadcasts it must send, the limitation here is how well our client's connections can deal with the increased demand in both upload (client sending updates) and download (client receiving the increase in number of game states).

If successful, the result is that (in combination with low threshold values discussed in **4.3**), the client game state and server game state will be well synchronized. In fact, if it could be guaranteed that clients and servers would maintain a consistent state, the threshold could be increased, resulting in almost no correction needed for the duration of the client's session.

There are 2 issues that arise with this:

1) Survival of the fittest – If updates are demanded at such high frequencies, clients whose bandwidth does not meet the requirements are going to be heavily throttled. Late packets will arrive to the server, and correction is going to be required a large amount of the time. The game will be unplayable for these clients.

2) Network overhead – While a UDP header packet may be considerably smaller in size than TCP, sending so many packets per time frame is costly. If most of these packets have very minimal changes in game state (a player moved 0.01m for example) then what is the point? With the exception of eSports, there is rarely the

need of such an accurate game state, at the cost of many player's experiences. This is wasted bandwidth and can be much better utilized.

An example of issue 1 (same conditions as **4.421**) is as follows:

1) Client 1 and Client 2 both shoot each other at (roughly) the same time.
2) Client 1 sends the data immediately. Client 2, who has their inputs queued on their network, adds their packets onto the back of the queue.
3) The server receives Client 1's packet, and very shortly after processes the new game state, returning it to all clients.
4) Client 1 and Client 2 can receive the new state simultaneously (asynchronous connections), where Client 2 is shot. On both player's machines Client 2 dies.
5) Client 2's packet eventually reaches the server, but it is old. It is either discarded immediately (if it does not pass a threshold) or validated to see if it can still be applied to the current game state and dealt with accordingly. The server knows that Client 2 has been shot since the packet was sent, so it does not pass validation and it discarded.

It is important to understand that games can handle scenarios similar to this differently. Some, for example CSGO, will ignore any bullets from a client after they have been killed on the server:

"In some games. e.g. CSGO, if the first shot arriving at the server kills the target, any subsequent shots by that player that arrive to the server later will be ignored. In this case, there cannot be any "mutual kills", where both players shoot within 1 tick and both die." [43]

Those who have the conditions, will have a very accurate multiplayer game, while those with poor conditions will suffer. As previously mentioned, anywhere between 10hz and 30hz is common, and accommodates a range of bandwidths.

Some games have implemented features to that modify their update rates to deal with a range of network connections. Overwatch received a 'high bandwidth patch' that:

Upgrades Overwatch's netcode to a high-bandwidth mode that updates data (something called a tick rate) 60 times per second instead of only 21 times per second on PC… This feature only works with players that have enough bandwidth on their connection to support it. If you don't, Blizzard has built in a dynamic tool that will automatically return you to the old update rate. [44]

Therefore, player's get the best of both worlds: those with the better bandwidth will receive better synchronization, while others will still be able to play without being at a disadvantage.

5– Authority, Roles & Cheating

5.1 Authority

Authority in a network determines which machines are allowed to perform actions in the game. This is needed for several reasons:

1. Cheating – Without authority, there is no mechanism by which the game can differentiate servers from clients, or clients from each other (regarding their privileges). This means that any player has the capacity to modify their game state through code injection, and, in some cases, this will update everyone else's game states also. Methods can be taken to tackle this, in both P2P and client/server, though without authority it becomes more difficult to validate data across the network.

2. Client-Side Prediction – With client-side prediction, inputs can be processed instantaneously on the local machine as soon as they occur. Upon reaching the server, this data needs to be tried for correction. Without knowing what role belongs to that input, the server has no way of knowing if this is a *predicted* input against a regular simulation.

3. Improved Bandwidth Overhead – The ability to break every game object in the world into designated authorities allows much greater control over how these objects should behave. For instance, any objects in the world that do not require client-side prediction, can be given a different authority that those which are; while other objects may be given total authority (only the server really should have this). Allowing clients to perform as many calculations as possible, while still being validated, is a good way or reducing the bandwidth usage of the server

5.11 Client/Server

In a client/server mode, the server runs a copy of the game, the *true* game state. This game need not contains graphics, audio etc., it only requires the information that is impactful to gameplay[27]. In this model the server *always* has authority. The client, therefore, upon receiving the updated game state from the server, will take what the server says, even if its own state is completely different.

Tim Sweeny claims in *Unreal Networking Architecture* that: *"The Server Is The Man"* [31], specifically relating to the behavior between clients and servers in this model.

[27] Movement, inputs, win conditions etc.

5.12 P2P & Lockstep

In a P2P model, clients have the same authority. This means the objects in their game world (including the player character) are open to manipulation, as there is no server control to validate the data. As a result, if one client was to, for example, set their player's health to infinity, there a few mechanisms to deny this from happening.

One solution to the poor security of a P2P network is the lockstep model mentioned in **3.13**. Here, all clients wait for all other player's commands before their own inputs can execute. This can be extended further, by also adding a *commitment* value, before their command is sent. The commitment is just a value that relates to the corresponding input, it could be a hash of the action, for instance.

The commitment is a representation of an action that:

- *Cannot be used to infer the action; and*

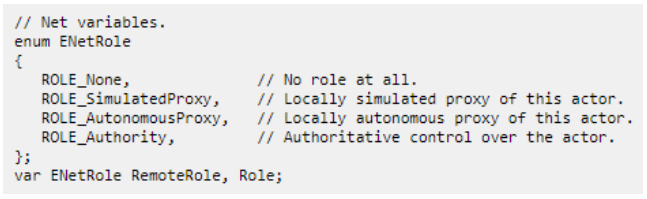
- *Easily compares whether an action corresponds with a commitment.* [45]

All players will receive all other's players commitments and commands. Then, before any command can be executed, it is compared with the corresponding commitment value to validate that they match.

5.13 Roles in Unreal

In Unreal, specifically UE4, authority manifests itself in the application of roles. Every actor in the game world is given a role (even in single player games). Then, whenever these objects are manipulated in some way, change of position etc., these roles are used to help validate the data, and also assist in server-side simulation (extra info from client-controlled objects, for instance).

Roles can be one of 4 types:



```
// Net variables.
enum ENetRole
{
   ROLE_None,              // No role at all.
   ROLE_SimulatedProxy,    // Locally simulated proxy of this actor.
   ROLE_AutonomousProxy,   // Locally autonomous proxy of this actor.
   ROLE_Authority,         // Authoritative control over the actor.
};
var ENetRole RemoteRole, Role;
```

Figure 26: UE4 ENetRole [31]

Role == ROLE_SimulatedProxy - means the actor is a temporary, approximate proxy which should simulate physics and animation. On the client, simulated proxies carry out their basic physics (linear or gravitationally-influenced movement and collision), but they don't make any high-level movement decisions. They just go. They can only execute script functions with the simulated keyword; and they can only enter states marked as simulated. This situation is only seen in network clients, never for network servers or single-player games.

Role == ROLE_AutonomousProxy - means the actor is the local player. Autonomous proxies have special logic built in for client-side prediction (rather than simulation) of movement. They can execute any script functions on the client; and they can enter any state. This situation is only seen in network clients, never for network servers or single-player games.

Role == ROLE_Authority - means this machine has absolute, authoritative control over the Actor.[31]

ROLE_SimulatedProxy and ROLE_AutonomousProxy are not used for non-networked games at all. Therefore, in non-multiplayer, roles would not be needed at all (or at least, all actors would have ROLE_Authority).

In a client/server model, the server will have ROLE_Authority on *all* actors. Whether they are players, objects, effects, the server has complete control and can execute any script. This is also the case for single-player games (here the machine running the game would have authority).

Actors in Single-Player Games:

1. ROLE_Authority – All actors.

2. ROLE_SimulatedProxy – No actors.

3. ROLE_ AutonomousProxy – No actors.

Actors in Networked Games:

Server:

1. ROLE_Authority – All actors.

2. ROLE_SimulatedProxy – No actors.

3. ROLE_ AutonomousProxy – No actors.

Client:

1. ROLE_Authority – Actors that were locally spawned by the client, such as gratuitous special effects which are done client-side in order to reduce bandwidth usage.

2. ROLE_SimulatedProxy – All non-player-controlled actors that require physical simulation.

3. ROLE_ AutonomousProxy – Player-controlled actor.

In UE4, actors do not just have a role alone, they also have a *remote role*. This states the authority of the object to the machine that is receiving it.

Therefore, a client's player character would have the following:

- Role: ROLE = ROLE_AutonomousProxy
- Remote Role = ROLE_Authority

The server would view this actor as:

- Role: ROLE = ROLE_Authority
- Remote Role = AutonomousProxy

i.e., the exact reverse (this is the expected behavior). The inclusion of the remote roll now allows all machines to detect the authority of which these inputs came from. This also acts as an additional guard against cheating.

In most cases, when a player is cheating, they are using a script to manipulate their own client's behavior, for instance an aimbot. When these inputs are sent to the server, their belonging authority can now be detected. The server will identify these commands came from a ROLE_AutonomousProxy, and therefore extra checks can be made to validate the data.

With a script like an aimbot, the validation could include how the player's targeting moves over time. In identifying anything suspicious, in this case a sudden snap of a target across a very small timeframe, flags could be raised. Then actions can be taken against this player, mostly in the form of kicking/banning.

5.2 Machine Learning Cheating

Until very recently, cheating mainly consisted of code injection or additional scripts that gave player's a complete advantage in the game. Therefore, most anti-cheat software (including BattleEye), were aimed at targeting these types of attacks.

As machines have become more powerful, their capability to run powerful algorithms has increased. Machine learning was only capable on a handful of powerful expensive machines. However, modern GPUs have the ability to run machine learning algorithms in a run time environment.

What this means is that, while the game is being player, machines can be used to affect the player's inputs towards 'beneficial' locations, based on their learning rewards. If an algorithm was to be rewarded whenever a player was targeted and killed, then the result is that (with additional hardware), players can be assisted during the game by a machine, which will handle most of the inputs for them.

In July 2021 a script was release that did exactly this, and very quickly the issue was very apparent. The cheat works by:

1. Play the game on a machine (any system capable of screen recording).

2. Connect a video capture card that records and streams the video.

3. Send the stream to a powerful system (i.e., one with a high-performing GPU)

4. Run the script on the machine, where the script will watch the stream and run its learning algorithm.

5. Connect an input-modifying adapter to the original machine playing the game.

6. The input will then receive the corrections from the machine and alter the player's controls for them.

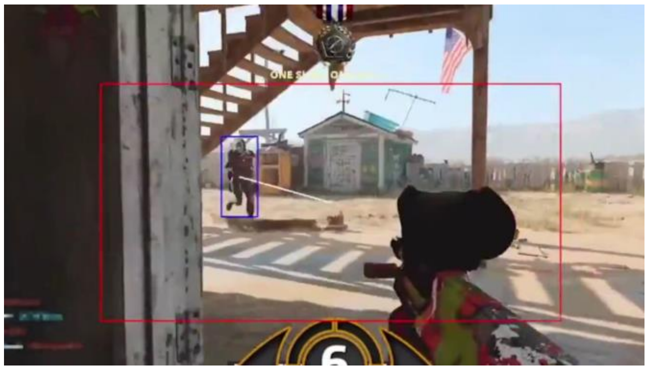


Figure 27: Machine Learning Cheat [46]

Due to the nature of the cheat, with regards to capturing the screen and sending inputs, it is (currently) impossible to detect. This is because, rather than intruding on any game files or injections to get player's positions, it scans the local screen, and sends input back.

The input that is generated can also be *tuned*, which causes even more issues for detectability. The algorithm can be told to what extent should the player's controls be corrected, i.e., should it almost snap, similar to an aimbot, or should it be more subtle, with only small corrections. With smaller corrections, the server cannot determine whether this was generated by a machine or not. The major issue is that, for the algorithm to run, it only requires a video stream, and a GPU. Therefore, machine learning algorithms can be run on *any* platform, even the Nintendo Switch, where previously cheating in games was much more difficult (and more easily detectable).

Due to these issues, action was taken against the provider of this machine learning algorithm. Activision Blizzard, only a few days after the script was released, shut down the providers [47]. Efforts are now in place to try to tackle machine learning cheats through anti-cheat software.

6 – Evaluation

Most modern games include some form of network for data transfer. Many games that are 'single-player', still contain data that is sent across the internet (achievements, statistics etc.).

For games that have a larger requirement for a well performing network, e.g., an MMO, more consideration is needed regarding the strategies and techniques used to improve a player's experience.
Regardless of the network requirements, attention must be given to the architecture, message transmission, anti-cheat etc., to ensure that the best solutions are chosen.

Networking in games is an extremely modifiable. For example, the transmission protocol can be customized to suit the needs of the game, e.g., UDP can be used with additions like redundancy, that in effect can almost simulate a TCP packet.

Ultimately, networked games aim to give the user am experience which is as close as possible to a local multiplayer game; that is seamless gameplay, with no indication of latency, with instant user feedback and a well synchronized game world. With all methods used to simulate this, there are always tradeoffs, for example client-side prediction exaggerates desynchronization, so further methods must be used to tackle these.

One issue that arises with handling data packets across a network is the strategies used to deal with any unforeseen errors, e.g., sudden network congestion, data loss etc. While many problems are out of a games' control, it is important that the correct response is made to benefit all other users involved.

This includes a situation such as a game hosted on a client/server model, with one client acting as the server. If this player's bandwidth was constricted, through sudden internet traffic for example (or lost entirely), all other players will get a poor experience, as their data uploads are dependent on the download speed of the host. In scenarios like these, techniques such as host swapping can be used to prevent all other players from experiencing a poor network experience.

Another scenario that is un-predictable, but must be prepared for, is cheating. While the online experience may be flawless in terms of latency and desynchronization, if poor validation techniques are in place, then hackers will ruin the experience for all other players, rendering the efforts to create a well performing network useless.

Once the network model is broken down into the 7 layers of the OSI model, there is lots of flexibility regarding how data can be sent from one source to another. While one decision will have its strengths and weakness, games can create unique solutions to mitigate the downside of any choice.

For example, StarCraft was made using the TCP lockstep model on a peer-to-peer server architecture. Having a game hosted this way, while guaranteeing the delivery of packets in order, would result in slow user feedback and make the game feel 'laggy', due to the higher latency of this solution. However, with the addition of other techniques such as animation/sound/VFX responses, it can be difficult to see the extra latency through a P2P lockstep model using TCP at all.

Understanding of how the underlying protocols handle data transfer at the fundamental level, such as techniques like piggybacking, will encourage the correct networking choices to improve user experience. If the network framework is understood well, then new strategies can be discovered, all with the aim of removing the drawbacks from having to send data potentially hundreds of kilometers away, to many different clients.

The networking system is a careful balance. Across the history of networking in games, methods have been tried and tested. Looking at the outcomes of these can help choose the type of online experience the users should experience, on a game-by-game basis.

Some of the major networking strategies and breakthroughs used in the history of games have been discussed in this paper. Though many more exist, they all follow the same principle: every user should have well synchronized worlds, where the positions, visual representations etc. of each client is as close to the real thing as can be achieved.

While latency can never *truly* be eliminated, and as such the perfect networked scenario can never occur, the techniques used in this paper attempt to reduce the impacts of the problems faced with networked multiplayer games.

ACKNOWLEDGEMENTS

I would like to thank Lucid Games Ltd, my employer, for hiring me as a junior network programmer during my third semester at university.

I would also like to thank Newcastle University, specifically my tutors Rich Davison, Gary Ushaw and Graham Morgan for delivering a fantastic course in difficult circumstances.

REFERENCES

[1] CompTIA. 2018. Network Protocol Definition: Computer Protocol: Computer Networks: CompTIA. (2018). Retrieved June 24, 2021 from https://www.comptia.org/content/guides/what-is-a-network-protocol
[2] CDW. 2021. Types of Network Protocols: The Ultimate Guide. (2021). Retrieved June 24, 2021 from https://www.cdw.com/content/cdw/en/articles/networking/2019/04/09/types-of-network-protocols.html
[3] Coen Goedegebure. 2019. The OSI Model. (April 2019). Retrieved June 24, 2021 from https://www.coengoedegebure.com/osi-model/
[4] Rachelle Miller. 2021. SANS-osi-model-overview.pdf - SANS Institute Information Security Reading Room The OSI Model An Overview Rachelle Miller Copyright SANS Institute 2020: Course Hero. (2021). Retrieved June 24, 2021 from https://www.coursehero.com/file/54487768/SANS-osi-model-overviewpdf/
[5] Red Hat Enterprise. 2006.(2006). Retrieved June 24, 2021 from https://web.mit.edu/rhel-doc/5/RHEL-5-manual/Deployment_Guide-en-US/ch-nfs.html
[6] Martin Weik. Fiber Optics Standard Dictionary. Retrieved June 24, 2021 from https://books.google.co.uk/books?id=ZCYBCAAAQBAJ&amp;pg=PA718&amp;dq=packet-switching&amp;hl=en&amp;sa=X&amp;redir_esc=y#v=onepage&amp;q=packet-switching&amp;f=false
[7] Aengus Matthews.Retrieved June 24, 2021 from https://users.cs.cf.ac.uk/MatthewsAJ1/Page1.html#:~:text=Splitting%20the%20data%20into%20packets,and%20the%20data%20is%20retrieved.
[8] Wikipedia. 2021. IPv4. (June 2021). Retrieved June 24, 2021 from https://en.wikipedia.org/wiki/IPv4
[9] TutorialsPoint. IPv4 - Packet Structure. Retrieved June 24, 2021 from https://www.tutorialspoint.com/ipv4/ipv4_packet_structure.htm#:~:text=Advertisements,referred%20to%20as%20IP%20Payload.
[10] D.Brent Chapman and Elizabeth D. Zwicky. 1999. Building Internet Firewalls. (February 1999). Retrieved June 24, 2021 from http://web.deu.edu.tr/doc/oreily/networking/firewall/ch06_03.htm
[11] Gary R. Wright and W.Richard Stevens. 1995. TCP/IP Illustrated, Boston, Massachusetts : Addison-Wesley Publishing Company.
[12] IBM Corporation. 2020.(2020). Retrieved June 24, 2021 from https://www.ibm.com/docs/en/spectrum-protect/8.1.10?topic=tuning-tcp-flow-control
[13] Joe Hanson. 2018. Run Your Game Servers Separate From Your Chat. (April 2018). Retrieved June 24, 2021 from https://www.pubnub.com/blog/why-you-should-run-your-game-servers-separate-from-your-chat/
[14] Black Box UK. 2005. 3748 - Simplex vs. duplex fiber patch cable. (2005). Retrieved June 24, 2021 from https://www.blackbox.co.uk/gb-gb/page/25069/Resources/Technical-Resources/Black-Box-Explains/Fibre-Optic-Cable/Simplex-vs-duplex-fiber-patch-cable
[15] Sheldon. 2014. Simplex vs Duplex Fiber Optic Cables. (July 2014). Retrieved June 24, 2021 from https://community.fs.com/blog/simplex-vs-duplex-fiber-optic-cables.html
[16] Karthikeya Boyini. 2019.(2019). Retrieved June 24, 2021 from https://www.tutorialspoint.com/what-is-piggybacking-in-networking
[17] Jun-Ho Huh. 2018. Reliable User Datagram Protocol as a Solution to Latencies in Network Games. (November 2018). Retrieved June 24, 2021 from https://www.mdpi.com/2079-9292/7/11/295
[18] Aengus Matthews.Retrieved June 24, 2021 from https://users.cs.cf.ac.uk/MatthewsAJ1/Page1.html#:~:text=Splitting%20the%20data%20into%20packets,and%20the%20data%20is%20retrieved.
[19] Wikpedia. 2021. Peer-to-peer. (August 2021). Retrieved August 19, 2021 from https://en.wikipedia.org/wiki/Peer-to-peer
[20] ComputerScienceGuru. 2019. Peer-to-peer networks. (December 2019). Retrieved August 19, 2021 from https://www.computerscience.gcse.guru/theory/peer-to-peer-networks
[21] TeachComputerScience. 2021. Peer to Peer NETWORK: PURPOSES, methods, types &amp; facts. (February 2021). Retrieved August 19, 2021 from https://teachcomputerscience.com/peer-to-peer-network/
[22] Mark Terrano and Paul Bettner. 2001. 1500 archers on a 28.8: Network programming in age of empires and beyond. (March 2001). Retrieved August 19, 2021 from https://www.gamasutra.com/view/feature/3094/1500_archers_on_a_288_network_.php
[23] kingadjon. 2019. Out of sync error in multiplayer games. (July 2019). Retrieved August 19, 2021 from https://forums.ageofempires.com/t/out-of-sync-error-in-multiplayer-games/44762
[24] Daniel Collier. 2015. Minimizing the pain of lockstep multiplayer. (November 2015). Retrieved August 19, 2021 from https://www.gamasutra.com/blogs/DanielCollier/20151124/251987/Minimizing_the_Pain_of_Lockstep_Multiplayer.php
[25] Wikipedia. 2021. Client–server model. (August 2021). Retrieved August 19, 2021 from https://en.wikipedia.org/wiki/Client%E2%80%93server_model
[26] CallOfDutyWiki. Host migration. Retrieved August 19, 2021 from https://callofduty.fandom.com/wiki/Host_Migration
[27] BattleEye. Interested in USING BATTLEYE? Retrieved August 19, 2021 from https://www.battleye.com/
[28] ICAT Design and Media College. 2018. What is Game Networking? (2018). Retrieved July 4, 2021 from http://blog.icat.ac.in/basics-of-game-networking
[29] Glenn Fiedler. 2010. What Every Programmer Needs To Know About Game Networking. (February 2010). Retrieved June 16, 2021 from https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/
[30] Stonely, D. 2021. Vehicle Networking Tech. .
[31] Epic Games. 2012. Unreal Networking Architecture. (2012). Retrieved June 16, 2021 from https://docs.unrealengine.com/udk/Three/NetworkingOverview.html
[32] Pukit Midha. 2019. What is desync in video games? (June 2019). Retrieved August 19, 2021 from https://www.quora.com/What-is-desync-in-video-games
[33] https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
[34] Wikipedia. 2021. Net neutrality. (August 2021). Retrieved August 19, 2021 from https://en.wikipedia.org/wiki/Net_neutrality
[35] Posted by Cody Haas and Ivan Vidal. 2021. Leveling up networking for a multi-game future. (June 2021). Retrieved August 19, 2021 from https://technology.riotgames.com/news/leveling-networking-multi-game-future
[36] Glenn Fiedler. 2020. The internet doesn't care about your game. (December 2020). Retrieved August 19, 2021 from https://www.networknext.com/post/the-internet-doesnt-care-about-your-game
[37] ESL. 2021. Advancing Network Performance in Esports with Network Next. (April 2021). Retrieved June 16, 2021 from https://www.eslgaming.com/article/advancing-network-performance-esports-network-next-4400
[38] Reddit. 2015. R/Overwatch - everything you need to know about tick rate, interpolation, lag compensation, etc. (2015). Retrieved August 19, 2021 from https://www.reddit.com/r/Overwatch/comments/3u5kfg/everything_you_need_to_know_about_tick_rate/
[39] Wikipedia. 2021. Netcode. (July 2021). Retrieved August 19, 2021 from https://en.wikipedia.org/wiki/Netcode
[40] MahaloMerky. 2019. r/CoDCompetitive - good example of HOW tick rate works in games. 12hz tick rate is completely unacceptable. (2019). Retrieved August 19, 2021 from https://www.reddit.com/r/CoDCompetitive/comments/e6nf9r/good_example_of_how_tick_rate_works_in_games_12hz/
[41] Lawlington. 2020. R/Codwarzone - tick rate and how it effects the game. (2020). Retrieved August 19, 2021 from https://www.reddit.com/r/CODWarzone/comments/foz7ml/tick_rate_and_how_it_effects_the_game/
[42] indigodarkwolf. 2016. R/Gamedev - generally, how often do most realtime multiplayer game servers send data to all clients? (2016). Retrieved August 19, 2021 from https://www.reddit.com/r/gamedev/comments/4eigzo/generally_how_often_do_most_realtime_multiplayer/
[43] Reddit. 2015. R/Overwatch - everything you need to know about tick rate, interpolation, lag compensation, etc. (2015). Retrieved August 19, 2021 from https://www.reddit.com/r/Overwatch/comments/3u5kfg/everything_you_need_to_know_about_tick_rate/
[44] Jeff Grubb. 2016. Overwatch gets 'high bandwidth' patch for pc, not ps4 and xbox one. (September 2016). Retrieved August 19, 2021 from https://venturebeat.com/2016/09/09/overwatch-netcode-tick-rate-60hz/
[45] Wikipedia. 2021. Lockstep protocol. (May 2021). Retrieved August 20, 2021 from https://en.wikipedia.org/wiki/Lockstep_protocol
[46] Humphfries Matthew. 2021. Machine learning is now being used to cheat in multiplayer games. (July 2021). Retrieved August 20, 2021 from https://uk.pcmag.com/games/134383/machine-learning-is-now-being-used-to-cheat-in-multiplayer-games
[47] Wesley Yin-Poole. 2021. Maker of machine learning cheat says "my intent was never to do anything ILLEGAL" after activision clampdown. (July 2021). Retrieved

August 20, 2021 from https://www.eurogamer.net/articles/2021-07-14-maker-of-machine-learning-cheat-says-my-intent-was-never-to-do-anything-illegal-after-activision-clampdown