11

Networking in Games Development

In chapter, the following recipes will be covered:

* Understanding the different layers
* Selecting the appropriate protocol
* Serialising the packets
* Using socket programming in games
* Sending data
* Receiving data
* Dealing with lag
* Using synchronised simulation
* Using area-of-interest filtering
* Using local perception filtering

# Introduction

In the modern era of video games, networking plays a huge role for the overall playability of a game. A single player game would offer at an average about 15-20 hours of gameplay. However, with the multi-player (networked) feature, the gameplay time increases manifold as now the users have to play against other human opponents and improve their tactics. May it be a PC game, console or mobile, having multi-layer has become a common feature these days. From a freemium model for games, where the monetization and revenue model is based around in-app purchases and Ads, it is necessary that the game has thousands or millions or active users per day. That is the only way the game will make money. When we speak about multi-player, we should not fool ourselves by thinking that it is restricted to PvP (player vs Player) in real time. It can also be asynchronous multi-player, where the player competes with the “data” from an active player’s deck but not with the player itself. It gives the illusion that the player is competing against a real player. Also with the advent of social media, networking also plays a role in making you compete against your friends. For example, in Candy Crush after you finish a level, you are shown how your friends fared in the same level and who is the next friend to beat. All this adds to the hype and hysteria around the game and makes it compelling to keep playing it.

# Understanding the different layers

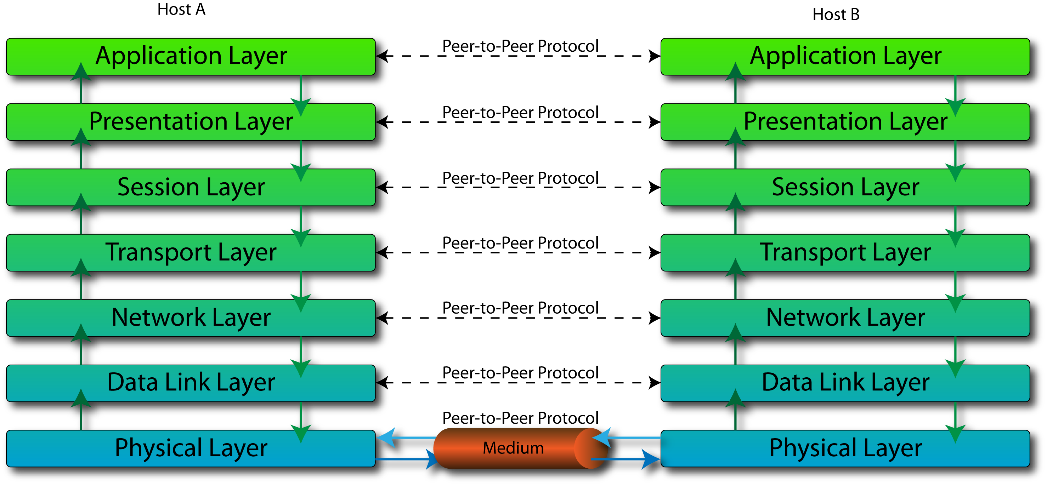
1. From a technical point of view, the entire networking model is divided into multiple layers. This model is also referred to as the OSI (Open Systems interconnection) model. Each layer has a special significance and must be understood properly to be able to interact with other layers of the topology.

## Getting ready

To step through this recipe, you will need a machine running Windows.

## How to do it...

In this recipe we will see how easy it is to understand the different layers of the networking topology.

* Observe following diagram

## How it works...

To understand the OSI model, we have to observer the model from the bottom of the stack to the top. The layer is the OSI model are:

* Physical Layer: This establishes the actual physical connection of the network. This defined whether we are using copper wires or fibre optics. It defines the network topology that is used, ring or bus and so on. It also defines the transmission mode whether it is simplex, half duplex or full duplex.
* Data link Layer: This provides the actual link between 2 connected nodes. There are 2 sublayers that the data link layer has; the MAC layer (Media Access Control) and the LLC layer (Logical Link control)
* Network Layer: This layers provides the functional means of transfer of variable length data called datagrams. The transfer happens from one connected node to another on the same network. This sort of forms the IP.
* Transport Layer: This layer also provides the functional means of transfer of data. The data is transferred from a source to destination travelling via one or more networks. Some of the protocols used here are TCP and UDP. TCP is the transfer control protocol and is a secured connection. UDP is the User datagram protocol and is the less secured one. In video games, we use both the TCP and the UDP protocol. When there is a situation when the user has to log into the server, we use TCP as it is more secure because the next information from the client is not sent unless there is an acknowledgement from the server about the previous data. So it can be slow, however because in the current situation security is more important than speed, we use TCP. After the user logs in, the game starts after other players have joined. Now we use UDP for majority of the situations as speed is more important than security and a few dropped packets should have a huge impact.
* Session Layer: This layer controls the connections between the network and the remote computer. This layer is responsible for establishing, managing and terminating a connection.
* Presentation Layer: This layer controls the different semantics that needs to be established between the connections. All the encryption logic is written in this layer.
* Application Layer: This layer deals with the communication with the software application itself. This is the closest layer from the end user point of view.

# Selecting the appropriate protocol

In games, most of the time there is an important decision which must be taken. The decision is whether to use TCP or UDP. The decision often ends up in favor of UDP, still it is important to understand the difference between the two.

## Getting ready

You need a Windows machine. No other pre-requisite is needed.

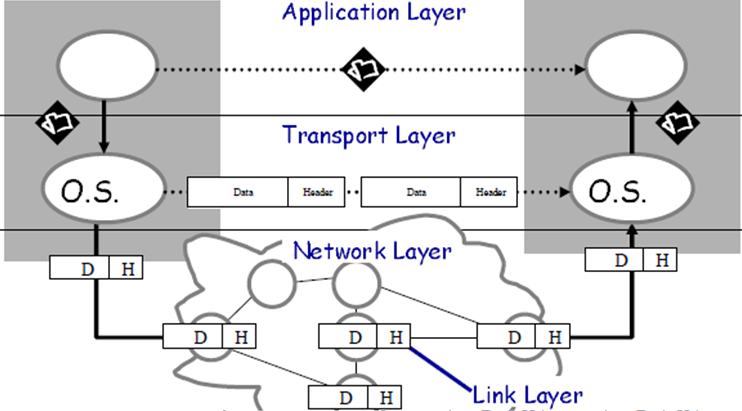
## How to do it...

In this recipe we will find out how easy it is to make a decision whether to use TCP or UDP.

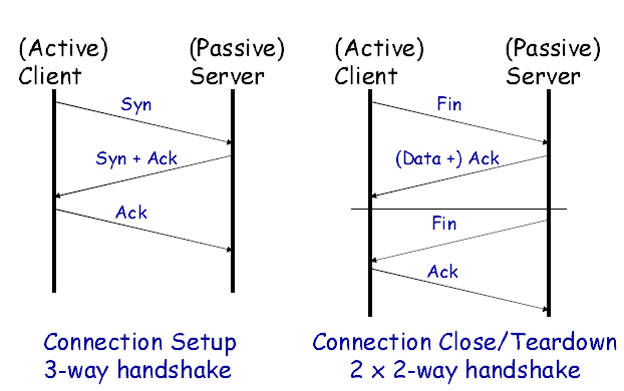
* Ask the following questions
  + Does the system require guaranteed delivery?
  + Is there a requirement for retransmission?
  + Does the system require any hand-shaking mechanism?
  + What kind of congestion control does it need?
  + Is speed a consideration in the system?

## How it works...

The TCP and the UDP is built on top of the IP layer.



A TCP connection is considered reliable because there is a 2-way handshake system enabled. Once the message gets delivered to the endpoint, an acknowledgement message is sent out. It supports various other services as well like congestion control and multiplexing. The fact that TCP is also full duplex, makes it quite a potent connection to use. The way it handles the reliable transfer of data is through byte sequence number. It sets a timeout function and based on time-outs it can decide whether a package has been delivered or not. The below mentioned diagram shows how the handshaking protocol is established.



Another mechanism for TCP is the sliding window mechanism by which it guarantees reliable delivery of data. It ensures that the data packets are delivered in a sequential manner and a flow control between the sender and receiver is established.

UDP is used when we are not too concerned if the data packets are delivered out of order. The main concern is how fast the packets are delivered. There is no reliability and no guarantee that the packets will be delivered.

Applications that require ordered delivery must restore datagram ordering themselves. Datagrams can be written to a target address without knowing if it exists or is listening. Messages can also be broadcast to all hosts on a particular subnet. DOOM had done this. Sometime if we require a very minimal reliability, UDP is open to adding that functionality. At the point it is also referred to as reliable UDP.

# Serialising the packets

Serialization is a key feature to have in the networking system. The process of serialization involves converting a message or data to a format that can be transmitted over the network and finally decoding it There are a variety of ways to serialize and deserialize data and it comes down to a matter of choice.

## Getting ready

1. You need to have a working Windows machine and Visual Studio. No other requirement is needed.

## How to do it...

In this recipe we will see how easy it is to serialize data.

Create a source file and derive from the serializer class.

using namespace xmls;

class LastUsedDocument: public Serializable

{

public:

LastUsedDocument();

xString Name;

xString Path;

xInt Size;

};

class DatabaseLogin: public Serializable

{

public:

DatabaseLogin();

xString HostName;

xInt Port;

xString User;

xString Password;

};

class SerialisationData: public Serializable

{

public:

SerialisationData();

xString Data1;

xString Data2;

xString Data3;

xInt Data4;

xInt Data5;

xBool Data6;

xBool Data7;

DatabaseLogin Login;

Collection<LastUsedDocument> LastUsedDocuments;

};

LastUsedDocument::LastUsedDocument()

{

setClassName("LastUsedDocument");

Register("Name", &Name);

Register("Path", &Path);

Register("Size", &Size);

};

DatabaseLogin::DatabaseLogin()

{

setClassName("DatabaseLogin");

Register("HostName", &HostName);

Register("Port", &Port);

Register("User", &User);

Register("Password", &Password);

};

SerialisationData::SerialisationData()

{

setClassName("SerialisationData");

Register("Data1", &Data1);

Register("Data2", &Data2);

Register("Data3", &Data3);

Register("Data4", &Data4);

Register("Data5", &Data5);

Register("Data6", &Data6);

Register("Data7", &Data7);

Register("Login", &Login);

Register("LastUsedDocuments", &LastUsedDocuments);

setVersion("2.1");

};

int main()

{

// Creating the Datas object

cout << "Creating object..." << endl;

SerialisationData \*Datas=new SerialisationData;

Datas->Data1="This is the first string";

Datas->Data2="This is the second random data";

Datas->Data3="3rd data";

Datas->Data4=1234;

Datas->Data5=5678;

Datas->Data6=false;

Datas->Data7=true;

Datas->Login.HostName="aws.localserver.something";

Datas->Login.Port=2000;

Datas->Login.User="packt.pub";

Datas->Login.Password="PacktPassword";

for (int docNum=1; docNum<=10; docNum++)

{

LastUsedDocument \*doc = Datas->LastUsedDocuments.newElement();

std::stringstream docName;

docName << "Document #" << docNum;

doc->Name = docName.str();

doc->Path = "{FILEPATH}"; // Set Placeholder for search/replace

doc->setVersion("1.1");

}

cout << "OK" << endl;

// Serialize the Datas object

cout << "Serializing object... " << endl;

string xmlData = Datas->toXML();

cout << "OK" << endl << endl;

cout << "Result:" << endl;

cout << xmlData << endl << endl;

cout << "Login, URL:" << endl;

cout << "Hostname: " << Datas->Login.HostName.value();

cout << ":" << Datas->Login.Port.toString() << endl << endl;

cout << "Show all collection items" << endl;

for (size\_t i=0; i<Datas->LastUsedDocuments.size(); i++)

{

LastUsedDocument\* doc = Datas->LastUsedDocuments.getItem(i);

cout << "Item " << i << ": " << doc->Name.value() << endl;

}

cout << endl;

cout << "Deserialization:" << endl;

cout << "Class version: " << Serializable::IdentifyClassVersion(xmlData) << endl;

cout << "Performing deserialization..." << endl;

// Deserialize the XML text

SerialisationData\* dser\_Datas=new SerialisationData;

if (Serializable::fromXML(xmlData, dser\_Datas))

{

cout << "OK" << endl << endl;

// compare both objects

cout << "Compareing objects: ";

if (dser\_Datas->Compare(Datas))

cout << "equal" << endl << endl; else

cout << "net equal" << endl << endl;

// now set value

cout << "Set new value for field >password<..." << endl;

dser\_Datas->Login.Password = "newPassword";

cout << "OK" << endl << endl;

cout << "compare objects again: ";

if (dser\_Datas->Compare(Datas))

cout << "equal" << endl << endl; else

cout << "net equal" << endl << endl;

cout << "search and replace placeholders: ";

dser\_Datas->Replace("{FILEPATH}", "c:\\temp\\");

cout << "OK" << endl << endl;

//output xml-data

cout << "Serialize and output xml data: " << endl;

cout << dser\_Datas->toXML() << endl << endl;

cout << "Clone object:" << endl;

SerialisationData \*clone1(new SerialisationData);

Serializable::Clone(dser\_Datas, clone1);

cout << "Serialize and output clone: " << endl;

cout << clone1->toXML() << endl << endl;

delete (clone1);

}

delete(Datas);

delete(dser\_Datas);

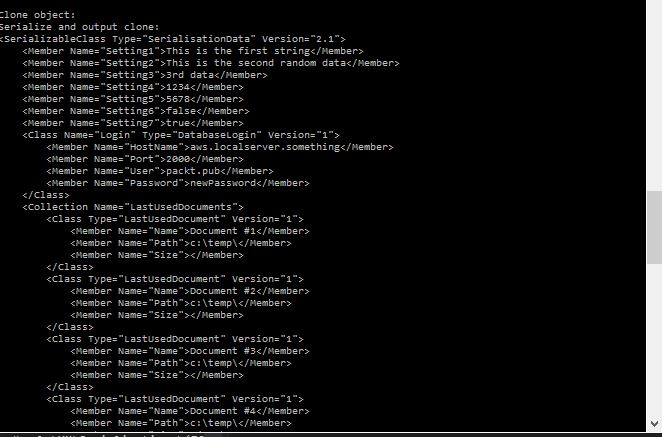
getchar();

return 0;

}

## How it works...

As mentioned before, serialise is to convert the data to a format that can be transferred. We can do this by using the Google API or using the JSON format. In this example we have used a XML serializer originally written by Lothar Perr. The whole idea behind the program is that we convert the data to a XML format. In the class serializable data, we publicly derive it from the serializable class. We create a constructor to register all the data elements and we create the different data elements that we want to be serialised. The data elements are of the type xstring class. In the constructor we register each of the data elements. Finally, from the client side, we assign the correct data to be sent and using the xml serializer class and tinyxml we generate the required xml. Finally, this xml will be sent across the network and on receive, it will be decoded using the same logic. XML can sometimes be considered quite heavy and cumbersome for games. Hence at those situations it is advisable to use JSON. Some of the modern engines like Unity3D and Unreal Engine, already have an inbuilt JSON parser which could be used to serialize the data. An example of a possible output from our code above is shown below.



# Using socket programming in games

Socket programming is one of the earliest mechanism to transfer data between end to end connections. Even now, if one is comfortable writing socket programming, it is a much better option for a relative small game than to use the 3rd party solutions as they add a lot of extra space.

## Getting ready

For this recipe, you will need a Windows machine and an installed version of Visual Studio.

## How to do it...

1. In this recipe we will find out how easy it is to write sockets

## struct sockaddr\_in

## {

## short sin\_family;

## u\_short sin\_port;

## struct in\_addr sin\_addr;

## char sin\_zero[8];

## };

## 

## int PASCAL connect(SOCKET,const struct sockaddr\*,int);

target.sin\_family = AF\_INET; // address family Internet

target.sin\_port = htons (PortNo); //Port to connect on

target.sin\_addr.s\_addr = inet\_addr (IPAddress); //Target IP

s = socket (AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); //Create socket

if (s == INVALID\_SOCKET)

{

return false; //Couldn't create the socket

}

## How it works...

* 1. When two applications are communicating with each other on different machines, one end of that communication channel is often mentioned as the socket. It is a combination of an IP address and a port. As we use signals or pipes to communicate in an inter process communication environment, similarly on different machines there is a need for sockets.
  2. Berkley Sockets (BSD) is the first internet socket API developed. Developed at the University of Berkley, California and given freely to all Berkley System Distribution of UNIX (BSD) and is present on all modern operating systems, UNIX variants, including OSX and Linux. Windows Sockets is based on BSD sockets providing additional functionality to comply with the regular Windows programming model. Winsock2 is the newest API
  3. Common domains are:
* AF UNIX: address format is UNIX pathname
* AF INET: address format is host and port number
  1. The various protocols can be used in the following way:
* TCP/IP (virtual circuits): SOCK\_STREAM
* UDP (datagram): SOCK\_DGRAM
  1. Steps of a simple socket connection
* Create a socket
* Bind the socket to an address
* Wait for input/output to be ready on the socket.
* Read and write to/from the socket.
* Repeat from 3 until you are done.
* Close the socket.
  1. int socket(domain, type, protocol);
* *Domain* should be set to PF\_INET (protocol family)
* *Type* is the connection type:
* SOCK\_STREAM for a byte stream socket
* SOCK\_DGRAM for a datagram (packet) socket.
* Protocol is the internet protocol in use:
* SOCK\_STREAM would normally give IPPROTO\_TCP.
* SOCK\_DGRAM would normally give IPPROTO\_UDP.
  1. int sockfd;
  2. sockfd = socket (PF\_INET, SOCK\_STREAM, 0);
* socket() returns a socket descriptor for use in later system calls or -1.
* When the protocol is set to 0, socket chooses the correct protocol based on type specified.
  1. int bind(int Socket, struct sockaddr \*myAddress, int AddressLen )
* Bind ties the socket to a local address.
* Socket is the socket descriptor.
* The myAddress is the local IP address and port.
* The AddressSize parameter gives the size (in bytes) of the address.
* bind() returns -1 on error.
  1. struct sockaddr\_in {
  2. short int sin\_family; // set to AF\_INET
  3. unsigned short int sin\_port; // Port number
  4. struct in\_addr sin\_addr; // Internet address
  5. unsigned char sin\_zero[8]; //set to all zeros
  6. }
  7. struct sockaddr\_in is a parallel structure that makes it easy to reference elements of the socket address. sin\_port and sin\_addr must be in Network Byte Order

# Sending the data

After we have correctly set up the sockets, the next step is to create the correct server and client architecture. Sending of data is pretty simple and just involves a few lines of code.

## Getting ready

To step through this recipe, you will need a machine running Windows with an installed Visual Studio.

## How to do it...

In this recipe, we will see how easy it is to send data.

## s = socket (AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); //Create socket

## if (s == INVALID\_SOCKET)

## {

## return false; //Couldn't create the socket

## }

## //Try connecting...

## if (connect(s, (SOCKADDR \*)&target, sizeof(target)) == SOCKET\_ERROR)

## {

## return false; //Couldn't connect

## }

## else

## return true; //Success

## }

## How it works...

The function that is used to communicate over the network is a function called send. int send (int sockfd, const void \*msg, int len, int flags);

sockfd is the socket descriptor you want to send data to (returned by socket() or got from accept()) whereas msg is a pointer to the data you want to send. len is the length of that data in bytes. For simplicity purpose, we can set that flag to 0 for now. sent() returns the number of bytes actually sent (may be less than the number you told it to send) or -1 on error. By just using this function, one is able to send messages or data from one connection point to the other. This function is used for stream of data and hence used for TCP. If we are to use datagrams and connectionless protocols, then we need to use the sendto function.

# Receiving the data

1. After we have correctly set up the sockets, and we have sent the data, the next step is to receive the data. Receiving of data is pretty simple and just involves a few lines of code.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

In this recipe, we will see how easy it is to receive data over the network.

## int recv(int sockfd, void \*buf,int len, int flags);

## How it works...

Just like the send function, just one function is used to receive the data over the network.

int recv(int sockfd, void \*buf, int len, int flags);

sockfd is the socket descriptor to read from. The next paramter buf is the buffer to read the information into whereas len is the maximum length of the buffer. The next parameter recv() returns the number of bytes actually read into the buffer or -1 on errorIf. recv() returns 0, the remote side has closed connection on you.

Using this line of code, we can receive data over the network. If the data is serialised while sending, we have to then take the data and deserialize the data at this point. This process will vary based on the method used to serialize the data.

# Dealing with lag

One of the major problems that occurs in a networked game is latency or lag. When two players are playing against each other and they are on a high speed network and a very low speed network respectively, how do we update the data? We need to update it in such a way that it looks normal for both the players. Also no player should get an undue advantage because of the situation.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

In this recipe, you will see few techniques discussed how to counter lag.

Generally, a networked game will have the following update loop. We need to figure out from the loop structure, what is the best way to counter lag.

## read\_network\_messages()

## read\_local\_input()

## update\_world()

## send\_network\_updates()

render\_world()read\_network\_messages()

read\_local\_input()

update\_world()

send\_network\_updates()

render\_world()

## How it works...

Generally, in most computer games, when networking is implemented a certain client server architecture is chosen. Often, an authorative server is chosen. This means the server dictates the time, result and other factors. The client is basically “dumb” and all it does is simulation based on data from the server. Now let us consider that two players are playing a multiplayer FPS game. One of them is on a high speed internet and the other is very slow. So if the client is dependent on the server for its updates, it will very difficult to accurately render the positions of the players on the client side. Let’s say UserA is on a high speed internet whereas UserB is on a low speed internet. UserA fires a bullet to UserB. Note UserA and UserB are also moving in the world space. How do we calculate the position of the bullet, position of each individual player? If we render exactly the information that is coming from the server, it will not be accurate as UserA would have already moved to a new position by the time UserB gets an update. To counter this there are 2 majorly used solution. One is called Client Side prediction. The other method is further divided into 2 more techniques: interpolation and extrapolation. Note that the round trip time will be quite acceptable if the computers are connected over LAN. All these problems that are being discussed are focusing on networking over the internet.

In client side prediction, the “dumb” factor is taken out of the client and the client starts predicting based on previous movement inputs, what will be the next position and animation states. Finally, when it gets an update from the server, the server will correct the mistakes and the positions would be transformed to the currently received one. There are loads of problems with this system. If the prediction was awry, there would be a big jitter as the position is being changed to the current one. Also let us consider sound effects and vfx effects. If the client (at userA) predicted that the userB was walking and the footsteps sound was played, and later the server informed that userB was actually in water, how do we suddenly rectify that mistake? The same goes for vfx effects and states. This system was used in a lot of the quake worlds.

The second system has 2 parts; extrapolation and interpolation. In extrapolation we render ahead of time. This is in some way similar to above. It takes the last known update from the server and then simulates forward in time. Thus, if you are 500 milliseconds lagged, and the last update you received was that the other player was running 300 units per second perpendicular to your view, then the client could assume that in "real time" the player has moved 150 units straight ahead from that last known position. The client could then just draw the player at that extrapolated position and the local player could still more or less aim right at the other player. However, the problem with this system is that it will rarely happen like that. The movement of the player may change, the state may change and hence this system should be avoided in most cases.

In interpolation we always render objects in the past. For instance, if the server is sending 25 updates per second (exactly) of the world state, then we might impose 40 milliseconds of interpolation delay in our rendering. Then, as we render frames, we interpolate the position of the object between the last updated position and the position one update before over that 40 milliseconds. The interpolation can be done by using the inbuilt lerp function in C++.As the object just gets to the last updated position, we receive a new update from the server (since 25 updates per second means that the updates come in every 40 milliseconds) we can start moving toward this new position over the next 40 milliseconds. The below diagram shows the difference in positions of the hitbox from the server and client side.



If the packet does not arrive after 40 milliseconds, that is there is a packet drop, then we have two options. The first option is to extrapolate using the method described above. The other option is to make the player go to an idle state till the next packet is received from the server.

# Using synchronised simulation

In a multiplayer there may be hundreds or thousands of computers connected at the same time. All of the computers would have different configurations. The speed would vary on all these computers. So the questions to be discussed is how to we synchronise the clock over all these systems so that they are also in sync.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

In this recipe we will see theoretically what are the two ways to synchronise the clock.

Pseudo Code.

**Method 1**

1. Send a message to UserA. Note the time till he receives the messages.
2. Send a message to UserB. Note the time again.
3. Calculate the median based on the values to decide an update time to update the clock over all these computers.

**Method 2**

1. Let the server do all the calculations.
2. Let the client do some local calculations.
3. When the client receives the update from the server, then either correct its mistakes or interpolate based on the results.

## How it works...

When we are trying to synchronise the clock, there are 2 methods. One method is that the server tries to find a median time when to synchronise all the clocks. To do this, we can include the mechanics in the game design itself. The server needs to find out the response time of each client machine. So it has to send out messages. These messages can be. Press R when Ready or a map is loaded on the client machine and the server takes a note of the time. Finally, when it has got a time from all the machines, it calculates a median and then updates the clock for all the machines at that time. The more messages the server sends out to the machines to calculate this median, the more accurate it will be. However, this in no way guarantees synchronisation.

Hence a better method is that the server does all the calculations and client does some local calculations as well using techniques described in previous recipe. Finally, when the server sends an update to the client, the client can correct itself or interpolate to get the desired result. This is a much better result and a much better system to have.

# Using area of interest filtering

When we are writing a networking algorithm, we need to decide what are the various objects or states that needs updating to the server or from the server. The higher the number of objects, the more time it will take to serialise and send the data across. Hence there is a need to prioritise what needs to be updated every frame and what objects can wait for few more cycles to be updated.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

In this recipe, we will see how easy it is to create an area of interest filtering.

* Create a list of all objects in the scene
* Add a parameter to the object denoting their priority
* Based on that priority number, pass it on to the update logic of the game.

## How it works...

In the game we need to define the objects in a certain priority order. The priority order determines if they are fit to be updated now or at a later time. The objects which require to be prioritised depend on a lot on the game design and a bit of research. For example, in a FPS game the objects in high priority would be the person that the user currently shot, the ammunition lying nearby, and of course the enemies and their positions which are in close proximity. This maybe different in case of a RPG or a RTS. So it definitely varies from one game to another.

After we have tagged each object by a priority number, then we can tell the update loop to just use the objects which are in priority level 1 and 2 for “per-frame” update and use objects in priority level 3 and 4 for late updates. This structure can also be modified by creating some sort of priority queue. From the queue, objects are popped out based on different update logic.

# Using local perception filter

This is yet another method to combat lag in networked games. This entire concept is mathematically based on the concept of perception. The summary of it is that if objects update and render correctly locally to a player, then we can create an illusion of realism. Hence the name local perception filter.

## Getting ready

To step through this recipe, you will need a machine running Windows and Visual Studio.

## How to do it...

In this recipe we will understand the theoretical concept of how easy it is to implement bullet time.

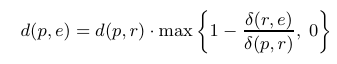
Pseudo Code

* Calculate the velocity local to the player.
* Accelerate the bullet when it starts and slow it down as it reaches the remote player
* From the remote player’s view, the bullet should appear shot at higher speed than normal speed and then slow down to normal

## How it works...

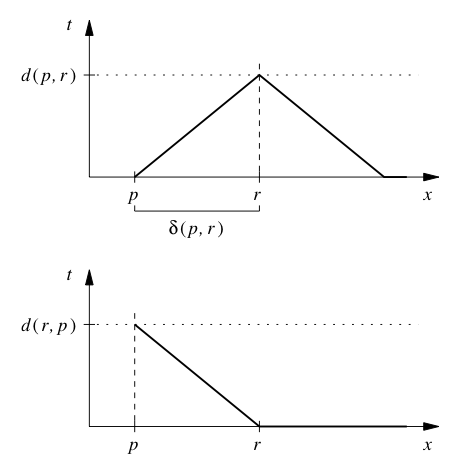
Local perception filters are also called bullet time and it was used for the first time in the movie “The Matrix”. Since then it has been used over a wide range of games. It is quite trivial to do in a single player mode, however in multi-player it gets a bit complex as it involves slowing down the rendering. Essentially the process is to speed up and slow down speed of passive entities when they are near local and remote players. It a method used to hide communication delays in networked virtual environments and was introduced by Sharkey. For our simplicity we will call local players a p, remote players as r and passive entities as e. Let us say that d(i,j) is delay, delta(i,j) is distance, then we get the below mentioned equations.

delay_eq1

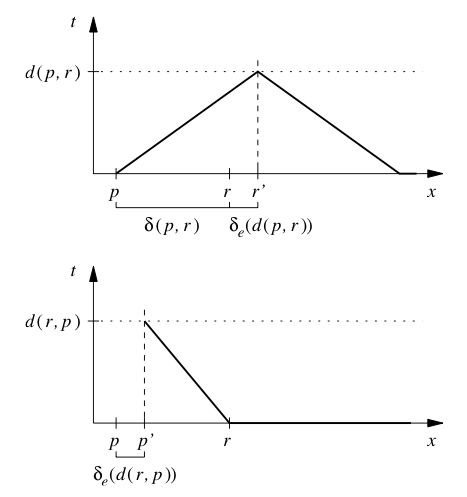
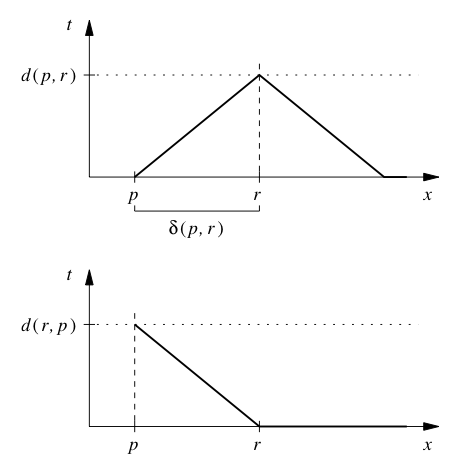


In a graphical format, it can be explained as below. So from p’s perspective it goes at slow uphill and then fast downhill. From r’s perspective it is faster at top.

One major limitation of the method is that this cannot be used for insta-hit weapons.

 The problem is when e reaches r, p’s view of e is not there yet, but e will speed up anyway in p’s view. To tackle this, we introduce a shadow r’ that buffers p’s view of the speedup.

delay_eq2_bdelay_eq2_aAfter adding the buffer, we will get the following revised graphs,



So at top, won’t speed up until r’ and at the bottom it starts show e at p’. This can also be viewed as a demo at the following URL : <http://mikolalysenko.github.io/local-perception-filter-demo/>