Networking Topics :

TCP :

UDP :

HTTP 1 :

HTTP 1.1 :

HTTP/2 :

QUIC and HTTP/3 :

Websocket :

Server Send Events :

Short Poling :

Long Polling :

WebRTC :

XMPP :

Webhook :

Detailed :

TCP (Transmission Control Protocol):

- TCP is a connection-oriented protocol that establishes a reliable and ordered connection between two endpoints.
- It provides reliable delivery by ensuring that all packets are received and retransmitting any lost packets.
- TCP guarantees in-order delivery, meaning the packets are received in the same order they were sent.

- It includes error detection and recovery mechanisms to detect and recover from errors that occur during transmission.
- TCP utilizes flow control mechanisms to manage the rate of data transmission and prevent overwhelming the receiver.
- TCP incorporates congestion control mechanisms to prevent network congestion and ensure fair resource allocation.
- Examples of TCP usage include web browsing, file transfers, email delivery, and other applications that require reliable and ordered delivery.

UDP (User Datagram Protocol):

- UDP is a connectionless protocol that does not establish a reliable and ordered connection between two endpoints.
- It provides unreliable delivery, as it does not guarantee that all packets will be received or in what order they will be received.
- UDP does not include built-in error detection or recovery mechanisms, leaving error handling to the applications themselves.
- It does not have flow control mechanisms to manage the rate of data transmission.
- UDP does not incorporate congestion control mechanisms and does not intervene in preventing network congestion.
- UDP is commonly used in applications where real-time or fast transmission is crucial, such as video streaming, online gaming, DNS lookup, and Voice over IP (VoIP).

It's important to note that TCP and UDP are designed for different purposes, and the choice of protocol depends on the specific requirements of the application or service being used.

TCP server (node.js) :

```
const net = require('net');

// Create a TCP server
const server = net.createServer(socket => {
  // Handle incoming connections

  // Event listener for data received from the client
  socket.on('data', data => {
    console.log('Received data from client:', data.toString());

    // Echo the received data back to the client
    socket.write(data);
  });
  // Event listener for client connection termination
  socket.on('end', () => {
    console.log('Client disconnected');
  });

  // Handle any errors that occur during the connection
  socket.on('error', error => {
    console.error('Socket error:', error);
  });
});

// Start the server and listen on a specific port
const port = 3000;
server.listen(port, () => {
  console.log(`Server started and listening on port ${port}`);
});
```

TCP Server (c++) :

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

const int BUFFER_SIZE = 1024;
const int PORT = 8080;

int main() {
    int serverSocket, clientSocket;
    struct sockaddr_in serverAddress, clientAddress;
    char buffer[BUFFER_SIZE];

    // Create socket
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0) {
        std::cerr << "Error creating socket." << std::endl;
        return 1;
    }

    // Set up server address
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(PORT);
    // Bind socket to address
    if (bind(serverSocket, (struct sockaddr *)&serverAddress,
sizeof(serverAddress)) < 0) {
        std::cerr << "Error binding socket." << std::endl;
        return 1;
    }

    // Listen for connections
    listen(serverSocket, 5);
    std::cout << "Server listening on port " << PORT << std::endl;

    while (true) {
        socklen_t clientLength = sizeof(clientAddress);

        // Accept client connection
        clientSocket = accept(serverSocket, (struct sockaddr
*)&clientAddress, &clientLength);
        if (clientSocket < 0) {
            std::cerr << "Error accepting client connection." << std::endl;
            return 1;
        }

        std::cout << "Client connected." << std::endl;

        // Receive data from client
        memset(buffer, 0, BUFFER_SIZE);
        ssize_t bytesRead = recv(clientSocket, buffer, BUFFER_SIZE - 1, 0);
        if (bytesRead < 0) {
            std::cerr << "Error receiving data from client." << std::endl;
            close(clientSocket);
            continue;
        }

        std::cout << "Received data: " << buffer << std::endl;

        // Send response to client
        const char *response = "Hello from server!";
        ssize_t bytesSent = send(clientSocket, response, strlen(response),
0);
        if (bytesSent < 0) {
            std::cerr << "Error sending response to client." << std::endl;
        }

        // Close client connection
        close(clientSocket);
        std::cout << "Client disconnected." << std::endl;
    }
    // Close server socket
    close(serverSocket);
    return 0;
}
```

TCP Client(node.js) :

```
const net = require('net');

// Create a TCP client
const client = new net.Socket();

// Connect to the server
const port = 3000;
const host = 'localhost';

client.connect(port, host, () => {
  console.log(`Connected to server: ${host}:${port}`);

  // Send data to the server
  const message = 'Hello, server!';
  client.write(message);
});

// Event listener for data received from the server
client.on('data', data => {
  console.log('Received data from server:', data.toString());

  // Close the client connection
  client.end();
});

// Event listener for server connection termination
client.on('end', () => {
  console.log('Disconnected from server');
});

// Handle any errors that occur during the connection
client.on('error', error => {
  console.error('Socket error:', error);
});
```

TCP Client(c++) :

```cpp
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

const int BUFFER_SIZE = 1024;
const int PORT = 8080;

int main() {
    int clientSocket;
    struct sockaddr_in serverAddress;
    char buffer[BUFFER_SIZE];

    // Create socket
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket < 0) {
        std::cerr << "Error creating socket." << std::endl;
        return 1;
    }

    // Set up server address
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &(serverAddress.sin_addr)) <= 0) {
        std::cerr << "Invalid address/Address not supported." << std::endl;
        return 1;
    }

    // Connect to server
    if (connect(clientSocket, (struct sockaddr *)&serverAddress,
sizeof(serverAddress)) < 0) {
        std::cerr << "Error connecting to server." << std::endl;
        return 1;
    }
    std::cout << "Connected to server." << std::endl;

    // Send data to server
    const char *message = "Hello from client!";
    ssize_t bytesSent = send(clientSocket, message, strlen(message), 0);
    if (bytesSent < 0) {
        std::cerr << "Error sending data to server." << std::endl;
        close(clientSocket);
        return 1;
    }

    // Receive response from server
    memset(buffer, 0, BUFFER_SIZE);
    ssize_t bytesRead = recv(clientSocket, buffer, BUFFER_SIZE - 1, 0);
    if (bytesRead < 0) {
        std::cerr << "Error receiving response from server." << std::endl;
        close(clientSocket);
        return 1;
    }

    std::cout << "Received response: " << buffer << std::endl;

    // Close socket
    close(clientSocket);

    return 0;
}
```

UDP Server(node.js) :

```
const dgram = require('dgram');

// Create a UDP server
const server = dgram.createSocket('udp4');

// Event listener for messages received from clients
server.on('message', (msg, rinfo) => {
  console.log(`Received message from client: ${msg.toString()}`);

  // Echo the message back to the client
  server.send(msg, rinfo.port, rinfo.address, (err) => {
    if (err) {
      console.error('Error sending message:', err);
    }
  });
});

// Event listener for server listening event
server.on('listening', () => {
  const address = server.address();
  console.log(`UDP server listening on ${address.address}:${address.port}`);
});

// Bind the server to a specific port and IP address
const port = 3000;
server.bind(port);
```

UDP Server(c++) :

```
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

const int BUFFER_SIZE = 1024;
const int PORT = 8080;

int main() {
    int serverSocket;
    struct sockaddr_in serverAddress, clientAddress;
    char buffer[BUFFER_SIZE];

    // Create socket
    serverSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (serverSocket < 0) {
        std::cerr << "Error creating socket." << std::endl;
        return 1;
    }

    // Set up server address
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(PORT);

    // Bind socket to address
    if (bind(serverSocket, (struct sockaddr *)&serverAddress,
sizeof(serverAddress)) < 0) {
        std::cerr << "Error binding socket." << std::endl;
        return 1;
    }

    std::cout << "Server listening on port " << PORT << std::endl;

    while (true) {
        socklen_t clientLength = sizeof(clientAddress);

        // Receive data from client
        memset(buffer, 0, BUFFER_SIZE);
        ssize_t bytesRead = recvfrom(serverSocket, buffer, BUFFER_SIZE - 1,
0, (struct sockaddr *)&clientAddress, &clientLength);
        if (bytesRead < 0) {
            std::cerr << "Error receiving data from client." << std::endl;
            continue;
        }

        std::cout << "Received data: " << buffer << std::endl;

        // Send response to client
        const char *response = "Hello from server!";
        ssize_t bytesSent = sendto(serverSocket, response, strlen(response),
0, (struct sockaddr *)&clientAddress, clientLength);
        if (bytesSent < 0) {
            std::cerr << "Error sending response to client." << std::endl;
        }
    }

    // Close server socket
    close(serverSocket);

    return 0;
}
```

UDP Client(node.js) :

```
const dgram = require('dgram');
```

```
// Create a UDP client
const client = dgram.createSocket('udp4');

// Message to send to the server
const message = 'Hello, server!';

// Send the message to the server
const serverPort = 3000;
const serverAddress = 'localhost';
client.send(message, serverPort, serverAddress, (err) => {
  if (err) {
    console.error('Error sending message:', err);
  }

  console.log(`Message sent to server: ${message}`);
});

// Event listener for messages received from the server
client.on('message', (msg, rinfo) => {
  console.log(`Received message from server: ${msg.toString()}`);

  // Close the client connection
  client.close();
});

// Event listener for client close event
client.on('close', () => {
  console.log('UDP client connection closed');
});

// Event listener for client error event
client.on('error', (err) => {
  console.error('UDP client error:', err);
});
```

UDP Client(c++) :

```
#include <iostream>
#include <cstdlib>
#include <cstring>
```

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

const int BUFFER_SIZE = 1024;
const int PORT = 8080;

int main() {
    int clientSocket;
    struct sockaddr_in serverAddress;
    char buffer[BUFFER_SIZE];

    // Create socket
    clientSocket = socket(AF_INET, SOCK_DGRAM, 0);
    if (clientSocket < 0) {
        std::cerr << "Error creating socket." << std::endl;
        return 1;
    }

    // Set up server address
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_port = htons(PORT);
    if (inet_pton(AF_INET, "127.0.0.1", &(serverAddress.sin_addr)) <= 0) {
        std::cerr << "Invalid address/Address not supported." << std::endl;
        return 1;
    }

    std::cout << "Enter a message: ";
    std::string message;
    std::getline(std::cin, message);

    // Send data to server
    ssize_t bytesSent = sendto(clientSocket, message.c_str(),
message.length(), 0, (struct sockaddr *)&serverAddress,
sizeof(serverAddress));
    if (bytesSent < 0) {
        std::cerr << "Error sending data to server." << std::endl;
        close(clientSocket);
        return 1;
    }

    socklen_t serverLength = sizeof(serverAddress);

    // Receive response from server
    memset(buffer, 0, BUFFER_SIZE);
    ssize_t bytesRead = recvfrom(clientSocket, buffer, BUFFER_SIZE - 1, 0,
(struct sockaddr *)&serverAddress, &serverLength);
    if (bytesRead < 0) {
        std::cerr << "Error receiving response from server." << std::endl;
        close(clientSocket);
        return 1;
    }

    std::cout << "Received response: " << buffer << std::endl;

    // Close socket
    close(clientSocket);

    return 0;
}
```

HTTP/1: Suppose you want to load a webpage that consists of an HTML file, CSS stylesheets, JavaScript files, and images. In HTTP/1, each resource requires a separate TCP connection. So, when you make a request for the HTML file, the connection is established, the file is transferred, and then the connection is closed. The same process is repeated for each resource. This leads to a lot of overhead and slower page loading times. For example, if the HTML file takes longer to load, it will block subsequent requests for other resources, causing delays.

HTTP/1.1: HTTP/1.1 introduced several improvements over HTTP/1. It added support for persistent connections, meaning multiple requests can be sent over a single connection without reopening it for each resource. This reduces the overhead of establishing connections for each request. Additionally, HTTP/1.1 introduced the concept of pipelining, allowing multiple requests to be sent without waiting for the corresponding responses. However, pipelining has limitations and can be affected by head-of-line blocking.

HTTP/2: With HTTP/2, the major upgrade was the introduction of multiplexing. Instead of relying on separate connections for each request, HTTP/2 allows multiple requests and responses to be sent concurrently over a single connection. This enables efficient use of network resources and reduces latency. For example, when you request a webpage in HTTP/2, all the resources can be sent over the same connection simultaneously, significantly improving the page loading speed.

HTTP/3: HTTP/3 is the next major version after HTTP/2 and is based on the QUIC protocol. The key difference in HTTP/3 is the use of the QUIC transport protocol instead of TCP. QUIC is designed to address the limitations of TCP, such as head-of-line blocking. It utilizes UDP (User Datagram Protocol) and includes built-in encryption. HTTP/3 retains the multiplexing feature of HTTP/2, allowing concurrent requests and responses. The main advantage of HTTP/3 is improved performance and reduced latency due to the use of QUIC.

QUIC: QUIC (Quick UDP Internet Connections) is a separate protocol from HTTP, although it is often used as the underlying transport protocol for HTTP/3. QUIC is built on top of UDP and includes features like multiplexing, encryption, and reduced latency. It eliminates head-of-line blocking by allowing independent streams of data to be transmitted concurrently over a single connection. QUIC also provides built-in security with encryption. It is designed to improve performance for web applications by reducing connection setup times and improving data transfer efficiency.

Head-of-line blocking is a problem that occurs in TCP (Transmission Control Protocol), specifically in the context of HTTP/1 and HTTP/1.1 protocols.

In TCP, data is transmitted in the form of packets, and these packets need to be received in the correct order to reconstruct the original data. Head-of-line blocking refers to a situation where a delayed or lost packet in a TCP connection causes subsequent packets to be held up until the missing packet is received or retransmitted. This can result in a delay in the delivery of subsequent packets and cause performance issues.In the context of HTTP/1 and HTTP/1.1, which use a separate TCP connection for each request/response transaction, head-of-line blocking occurs when a slow or delayed response for a particular resource blocks subsequent requests for other resources. For example, if a web page consists of multiple resources such as images, CSS, and JavaScript files, and one of these resources experiences a delay in the response, it holds up the delivery of subsequent requests, even if they could be processed independently. As a result, the overall loading time of the webpage is increased.

Head-of-line blocking is a significant drawback of HTTP/1 and HTTP/1.1, as it limits the parallelism and efficient utilisation of network resources. This problem was addressed and improved in later versions like HTTP/2 and HTTP/3, which introduced multiplexing and other techniques to overcome head-of-line blocking issues and enhance performance.

Websocket :

WebSocket is a communication protocol that provides full-duplex, bidirectional communication between a client and a server over a single, long-lived connection. It enables real-time data transfer, allowing both the client and server to send messages to each other at any time without the need for frequent requests and responses.

Here are some key aspects and features of WebSocket:

Full-duplex communication: WebSocket enables simultaneous, bidirectional communication between the client and server. Unlike traditional HTTP communication, where the client initiates requests and the server responds, WebSocket allows data to flow in both directions simultaneously.
Persistent Connection: Once established, a WebSocket connection remains open as long as needed, allowing real-time communication with low overhead. The persistent connection eliminates the need for repeated connection establishment and teardown for each interaction.
Low Latency: WebSocket reduces latency compared to traditional request-response mechanisms like HTTP polling. It enables instantaneous and efficient data transmission, making it suitable for real-time applications such as chat applications, collaborative tools, stock tickers, and multiplayer games.
Lightweight Header: The WebSocket protocol uses a minimal overhead in the form of a small header, resulting in efficient data transfer and reduced bandwidth consumption compared to HTTP.
Message-based: WebSocket sends messages between the client and server rather than HTTP request-response pairs. Messages can be of any format, such as plain text, JSON, XML, or binary data, allowing flexibility in data exchange.
Event-driven: WebSocket communication is event-driven. Both the client and server can trigger events, and the corresponding event handlers execute the necessary logic. This enables real-time notifications, updates, and interactions based on specific events or triggers.

Cross-Domain Communication: WebSocket supports cross-domain communication, allowing clients to establish WebSocket connections with servers on different domains or origins. This is facilitated through the WebSocket handshake and the appropriate headers exchanged during the connection setup.

To use WebSocket in your application, you'll typically need to implement WebSocket server-side code and use a WebSocket client library or the browser's built-in WebSocket API on the client side.

On the server side, you can use libraries such as `ws` for Node.js or frameworks like Socket.IO to handle WebSocket connections, events, and message routing. These server-side components manage the WebSocket handshake, connection management, and the exchange of messages between the client and server.

On the client side, you can use JavaScript WebSocket APIs, such as the `WebSocket` object in modern web browsers or WebSocket libraries for other platforms, to establish and manage WebSocket connections. The client-side code can define event handlers for different WebSocket events, such as connection establishment, message receipt, and connection closure.

With WebSocket, you can build real-time, interactive applications that require continuous communication between clients and servers, enabling rich user experiences and efficient data transfer in various domains.

https://bingx.com/en-gb/ (to check websocket connection)

Chat Server using websocket (node.js) :

```
const WebSocket = require('ws');

// Create a WebSocket server
const wss = new WebSocket.Server({ port: 8080 });

// Store connected clients
const clients = new Set();

// Broadcast a message to all connected clients
function broadcast(message) {
  clients.forEach((client) => {
    if (client.readyState === WebSocket.OPEN) {
      client.send(message);
    }
  });
}

// Event handler for WebSocket connection
wss.on('connection', (ws) => {
  // Add the client to the set of connected clients
  clients.add(ws);

  // Event handler for receiving messages from clients
  ws.on('message', (message) => {
    // Broadcast the message to all connected clients
    broadcast(message);
  });

  // Event handler for closing the WebSocket connection
  ws.on('close', () => {
    // Remove the client from the set of connected clients
    clients.delete(ws);
  });
});

console.log('WebSocket chat server is running on port 8080');
```

Chat Client using Websocket (Node.js) :

```
const WebSocket = require('ws');

// Create a WebSocket connection
const ws = new WebSocket('ws://localhost:8080');

// Event handler for successful WebSocket connection
ws.on('open', () => {
  console.log('Connected to the chat server.');

  // Prompt the user to enter a username
  const username = prompt('Enter your username: ');

  // Send the username to the server
  ws.send(JSON.stringify({ type: 'username', data: username }));

  // Listen for user input and send messages to the server
  process.stdin.on('data', (message) => {
    ws.send(JSON.stringify({ type: 'message', data: message.trim() }));
  });
});
```

```
// Event handler for receiving messages from the server
ws.on('message', (message) => {
  const data = JSON.parse(message);

  if (data.type === 'message') {
    console.log(`${data.username}: ${data.message}`);
  } else if (data.type === 'info') {
    console.log(data.message);
  }
});

// Event handler for closing the WebSocket connection
ws.on('close', () => {
  console.log('Connection to the chat server closed.');
});
```

We can also create websocket server and client connection to demonstrate push notification functionality which is a common part of any real world software .

Websocket server for push Notification :

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

// Store subscribed clients for each event
const subscriptions = {};

wss.on('connection', (ws) => {
  console.log('WebSocket client connected.');

  // Event handler for receiving messages from the client
  ws.on('message', (message) => {
    console.log('Received message:', message);

    // Subscribe client to a specific event
    const { event } = JSON.parse(message);
    subscribeClient(ws, event);
  });

  // Event handler for closing the WebSocket connection
  ws.on('close', () => {
    console.log('WebSocket connection closed.');
    // Unsubscribe client from all events upon connection close
    unsubscribeClient(ws);
  });
});

// Custom function to subscribe a client to an event
function subscribeClient(ws, event) {
  // Create event subscription if it doesn't exist
  if (!subscriptions[event]) {
    subscriptions[event] = new Set();
  }

  // Add client to the event subscription
  subscriptions[event].add(ws);

  console.log(`Client subscribed to event: ${event}`);
}

// Custom function to unsubscribe a client from all events
function unsubscribeClient(ws) {
  for (const event in subscriptions) {
    subscriptions[event].delete(ws);
  }

  console.log('Client unsubscribed from all events');
}

// Custom function to send push notification to subscribed clients
function sendPushNotification(event, message) {
  if (subscriptions[event]) {
    subscriptions[event].forEach((client) => {
      if (client.readyState === WebSocket.OPEN) {
        client.send(message);
      }
    });
  }
}

// Example: Send a push notification after 5 seconds
setTimeout(() => {
  const event = 'newMessage';
  const message = 'You have a new message!';
  sendPushNotification(event, message);
}, 5000);
```

```
console.log('WebSocket server is running on port 8080');
```

Websocket Client for Push Notification :

```
const WebSocket = require('ws');

const ws = new WebSocket('ws://localhost:8080');

// Event handler for successful WebSocket connection
ws.on('open', () => {
  console.log('WebSocket connection established');

  // Send a message to subscribe to a specific event
  const subscription = { event: 'newMessage' };
  ws.send(JSON.stringify(subscription));
});

// Event handler for receiving messages from the server
ws.on('message', (message) => {
  console.log('Received message:', message);

  // Perform any actions based on the received push notification
});

// Event handler for closing the WebSocket connection
ws.on('close', () => {
  console.log('WebSocket connection closed');
});
```

Long Polling :

Long Polling is a technique where the client sends a request to the server and the server holds the request open until new data is available or a timeout occurs. If new data is available, the server responds with the updated data, and the client immediately sends another request to maintain a continuous connection. This approach enables real-time updates without the need for constant polling from the client.

Server Code example for Long polling (node.js) :

```
const express = require('express');
const app = express();

// Simulated data that can be updated
let latestData = 'Initial Data';

// Array to store pending client responses
const clientResponses = [];

// Endpoint for long polling requests
app.get('/data', (req, res) => {
  const timeout = 30000; // Long polling timeout (30 seconds)

  // Create a timeout for the long polling request
  const timer = setTimeout(() => {
    // Remove the current response object from the pending
responses array
    const index = clientResponses.indexOf(res);
    if (index !== -1) {
      clientResponses.splice(index, 1);
    }

    // Send an empty response to the client indicating no updates
    res.json({ data: null });
  }, timeout);

  // Add the response object to the pending responses array
  clientResponses.push(res);

  // Remove the response object from the pending responses array
when the connection is closed
  res.on('close', () => {
    clearTimeout(timer);
    const index = clientResponses.indexOf(res);
    if (index !== -1) {
      clientResponses.splice(index, 1);
    }
  });
});

// Endpoint to update the data
app.post('/update', (req, res) => {
  // Update the data when receiving a POST request
  latestData = 'Updated Data';

  // Notify all pending client responses about the updated data
  clientResponses.forEach((clientRes) => {
    clientRes.json({ data: latestData });
  });

  // Send a response to the client
  res.send('Data updated successfully');
});

app.listen(3000, () => {
  console.log('Long Polling server listening on port 3000');
});
```

Difference between long polling and Websocket Communication :

WebSocket protocol and long polling are both techniques used to achieve real-time, bidirectional communication between a client (usually a web browser) and a server. However, they differ significantly in their approach and behavior:

Protocol:

- WebSocket: WebSocket is a standardized communication protocol that provides full-duplex, bidirectional communication channels over a single TCP connection. It allows both the client and server to send and receive messages at any time during the connection's lifespan.
- Long Polling: Long polling is not a standardized protocol; instead, it is a technique that uses HTTP. The client sends an HTTP request to the server, and the server keeps the request open until new data is available or a timeout occurs. When data is available, the server responds to the client's request with the new data, and the client immediately processes it and sends another request to keep the connection open.

Connection Management:

- WebSocket: WebSockets establish a single persistent connection between the client and the server that remains open as long as both parties want it to stay open. This allows real-time data to be sent in both directions without the need to repeatedly open and close connections.

- Long Polling: With long polling, the connection is continuously opened and closed in a request-response fashion. Each time the client receives a response from the server, it immediately sends another request to keep the connection open. This process is repeated in a loop, allowing the server to push data to the client when it becomes available.

Short Polling :

Short polling is a client-server communication mechanism where the client periodically sends requests to the server at fixed intervals to check for updates. The server responds to each request with the current data, regardless of whether it has been updated since the last request. The client then processes the response and decides whether to make subsequent requests for updates.

Code for client side doing short polling (Node.js) :

```
function fetchData() {
  fetch('http://localhost:3000/data')
    .then((response) => response.json())
    .then((data) => {
      console.log('Received data:', data);

      // Continue polling for updates after a short delay
      setTimeout(fetchData, 5000);
    })
    .catch((error) => {
      console.error('Error:', error);

      // Retry after a short delay in case of errors
      setTimeout(fetchData, 5000);
    });
}

// Start fetching data
fetchData();
```

Server-Sent Events also known as SSE :

Server-Sent Events (SSE) is a unidirectional communication technique where the server can send real-time updates or events to the clients over a persistent HTTP connection. SSE is built on top of standard HTTP, allowing servers to push data to clients without the need for continuous client requests.

Here's an overview of how Server-Sent Events work:

> The client establishes an HTTP connection with the server using the SSE protocol.
>
> The server responds with an SSE stream by setting the response content type to `text/event-stream` and sending a series of event messages to the client. The SSE stream consists of individual events, each separated by a newline character (`\n`).
>
> Each event message contains one or more event fields, such as `event`, `data`, `id`, and `retry`, each with its respective value.
>
> The server can send events at any time by writing event messages to the SSE stream. The client receives these events as they are pushed by the server.
>
> The client's SSE API listens for incoming events and triggers event handlers when new events are received.
>
> The client can perform actions based on the received events, such as updating the user interface or executing specific logic.

SSE Server Code (Node.js) :

```
const express = require('express');
const app = express();

// Route to handle SSE connections
app.get('/sse', (req, res) => {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');
  res.setHeader('Access-Control-Allow-Origin', '*');
  // Send a welcome message to the client
  res.write('event: welcome\n');
  res.write('data: Welcome to the SSE stream!\n\n');

  // Generate and send an event every second
  const eventInterval = setInterval(() => {
    const eventTime = new Date().toLocaleTimeString();
    const eventData = `Event generated at ${eventTime}`;

    res.write('event: server-time\n');
    res.write(`data: ${eventData}\n\n`);
  }, 1000);

  // Close the SSE connection after 10 seconds
  setTimeout(() => {
    clearInterval(eventInterval);
    res.end();
  }, 10000);
});

// Start the server
app.listen(3000, () => {
  console.log('SSE server listening on port 3000');
});
```

SSE Client code (Node.js) :

```
const eventSource = new EventSource('http://localhost:3000/sse');

eventSource.addEventListener('welcome', (event) => {
  console.log('Received welcome event:', event.data);
});

eventSource.addEventListener('server-time', (event) => {
  console.log('Received server-time event:', event.data);
});

eventSource.onerror = (error) => {
  console.error('SSE error:', error);
};
```

REST API :

REST (Representational State Transfer) is an architectural style for designing networked applications. RESTful APIs (Application Programming Interfaces) are a way to implement the principles of REST in web services. REST APIs allow different systems to communicate with each other over the internet using standard HTTP methods like GET, POST, PUT, DELETE, etc.

REST server for todo list (Node.js) :

```
const express = require('express');
const app = express();

app.use(express.json());

let todos = [
  { id: 1, task: 'Learn Node.js', completed: false },
  { id: 2, task: 'Read a book', completed: true }
];

// Get all todos
app.get('/todos', (req, res) => {
  res.json(todos);
});

// Get a specific todo by ID
app.get('/todos/:todoId', (req, res) => {
  const todoId = parseInt(req.params.todoId);
  const todo = todos.find(todo => todo.id === todoId);
  if (todo) {
    res.json(todo);
  } else {
    res.status(404).json({ error: 'Todo not found' });
  }
});

// Create a new todo
app.post('/todos', (req, res) => {
  const { task, completed } = req.body;
  const newTodo = { id: todos.length + 1, task, completed };
  todos.push(newTodo);
  res.status(201).json(newTodo);
});

// Update a todo
app.put('/todos/:todoId', (req, res) => {
  const todoId = parseInt(req.params.todoId);
  const { task, completed } = req.body;
  const todo = todos.find(todo => todo.id === todoId);
  if (todo) {
    todo.task = task;
    todo.completed = completed;
    res.json(todo);
  } else {
    res.status(404).json({ error: 'Todo not found' });
  }
});

// Delete a todo
app.delete('/todos/:todoId', (req, res) => {
  const todoId = parseInt(req.params.todoId);
  const index = todos.findIndex(todo => todo.id === todoId);
  if (index !== -1) {
    todos.splice(index, 1);
    res.sendStatus(204);
  } else {
    res.status(404).json({ error: 'Todo not found' });
  }
});

// Start the server
app.listen(3000, () => {
  console.log('Server is running on port 3000');
});
```

gRPC :

gRPC (Google Remote Procedure Call) is a high-performance, open-source framework developed by Google that allows you to define remote services and generate efficient client-server code for communication between them. It uses Protocol Buffers (protobuf) as the interface description language and supports multiple programming languages, including Node.js.

gRPC is used where performance, scalability, and efficient data serialization are crucial. It is often preferred in microservices architectures, internal service-to-service communication, and scenarios involving high-volume data transfer.
gRPC primarily relies on HTTP/2 as the underlying transport protocol.

GraphQL :

GraphQL is a query language for your API, allowing clients to request the exact data they need and nothing more. It was developed by Facebook and provides a more efficient and flexible alternative to traditional RESTful APIs. With GraphQL, clients can specify the structure of the response, enabling them to retrieve multiple resources in a single request and avoid over-fetching or under-fetching data.

SOAP :

Webhook :

A webhook is a mechanism for communication between two applications where one application sends data to another application automatically when a specific event or trigger occurs. It allows real-time data transfer from a sender (usually a server or service) to a receiver (also a server or service) without the need for the receiver to poll for updates continuously. Webhooks are commonly used in various scenarios, such as integrating third-party services, handling real-time events, and enabling push notifications.

Webhook Implementation : https://www.youtube.com/watch?v=Jw9Axr_n7J0

DASH :

Popular Video Streaming Protocols :
RTMP, HLS and WebRTC, DASH

When to use gRPC , REST, GraphQL :
https://www.youtube.com/watch?v=veAb1fSp1Lk

P2P Network Protocols :
A peer-to-peer (P2P) network protocol is a communication protocol used in peer-to-peer networks, where individual nodes (peers) can directly communicate and share resources with each other without relying on a central server or intermediary. In a P2P network, each node can act as both a client and a server, making the network more decentralized and distributed.

BitTorrent: A P2P protocol used for file sharing and distribution of large files among users.
WebRTC (Web Real-Time Communication): A P2P protocol for real-time audio, video, and data communication directly between web browsers.

WebRTC (Web Real-Time Communication) is an open-source protocol and technology developed by Google that enables real-time audio, video, and data communication directly between web browsers. It allows web applications to establish peer-to-peer connections without the need for plugins or additional software.

Blockchain protocol is a type of peer-to-peer (P2P) network protocol. Blockchain technology was designed to function as a decentralized and distributed system, where each participant (node) in the network can interact directly with others without the need for a central authority.

Monolith :
Cons :
DEVELOPMENT IS SLOW
PATH FROM COMMIT TO DEPLOYMENT IS LONG AND ARDUOUS
SCALING IS DIFFICULT
DELIVERING A RELIABLE MONOLITH IS CHALLENGING
LOCKED INTO INCREASINGLY OBSOLETE TECHNOLOGY STACK

Scale cube and microservices :

1. X-AXIS SCALING LOAD BALANCES REQUESTS ACROSS MULTIPLE INSTANCES

2. Z-AXIS SCALING ROUTES REQUESTS BASED ON AN ATTRIBUTE OF THE REQUEST

3. Y-AXIS SCALING FUNCTIONALLY DECOMPOSES AN APPLICATION INTO SERVICES

Benefits of the microservice architecture : The microservice architecture has the following benefits:

- It enables the continuous delivery and deployment of large, complex applications.
- Services are small and easily maintained.
- Services are independently deployable.
- Services are independently scalable. The microservice architecture enables teams to be autonomous.
- It allows easy experimenting and adoption of new technologies.
- It has better fault isolation.

The major drawbacks and issues of the microservice architecture:
Finding the right set of services is challenging.
Distributed systems are complex, which makes development, testing, and deployment difficult.
Deploying features that span multiple services requires careful coordination.
Deciding when to adopt the microservice architecture is difficult.