sse-ws-http-tcp

Hi

Hello! How can I assist you today?

1. What are "server push events"?
2. When was the first type of this technology invented?

1. Server-Sent Events (SSE), also known as Server Push Events, is a technology used for real-time communication between a web server and a client. With SSE, the server can initiate a stream of data to the client, which is continuously transmitted to the client as updates occur on the server-side. This allows the client to receive real-time updates without having to constantly poll the server for new information.
2. Server-Sent Events were first introduced as part of the HTML5 specification in 2011. The technology is a part of the HTML5 EventSource API and is supported by most modern web browsers, including Chrome, Firefox, Safari, and Edge. However, it's important to note that SSE is different from other real-time communication technologies such as WebSockets, which were also introduced as part of the HTML5 specification.

Do I need to use a library to have server-sent events, on the client?

No, you do not necessarily need to use a library to use Server-Sent Events (SSE) on the client-side. SSE is supported natively by most modern web browsers, so you can use the built-in EventSource API to establish a connection to the server and receive real-time updates.

Here's an example code snippet that shows how to use SSE without any additional libraries:

javascript Copy code

```javascript
const eventSource = new EventSource('/path/to/server');

eventSource.addEventListener('message', function(event) {
  console.log('Received message:', event.data);
});

eventSource.addEventListener('error', function(event) {
  console.error('Error:', event);
});
```

In this example, `new EventSource('/path/to/server')` creates a new SSE connection to the specified server. The `addEventListener` method is used to define two event listeners: one for incoming messages, and another for error events. When a message is received from the server, the `message` event listener is called and the data from the message is logged to the console.

Of course, there are also libraries available that provide additional functionality and convenience when working with SSE, such as automatically reconnecting to the server if the connection is lost. However, these libraries are not strictly necessary to use SSE on the client-side.

What's the server code?

The server-side code required to implement Server-Sent Events (SSE) will depend on the programming language and framework that you are using. However, the basic idea is that the server will need to establish an HTTP connection with the client and continuously send data as new events occur.

Here is an example implementation of an SSE server using Node.js and the Express

Node.js

Sure! Here's an example implementation of an SSE server using Node.js and the Express framework:

javascript Copy code

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

app.get('/sse', function(req, res) {
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  // Send an initial "hello" message to the client
  res.write('data: hello\n\n');

  // Periodically send updates to the client
  setInterval(function() {
    res.write('data: ' + new Date() + '\n\n');
  }, 1000);

  // Clean up the connection when the client disconnects
  req.on('close', function() {
    console.log('Client closed SSE connection');
    clearInterval(intervalId);
  });
});

app.listen(3000, function() {
  console.log('SSE server running on port 3000');
});
```

In this example, we define a new route for SSE connections at the path '/sse'. When a client connects to this route, we set the response headers to indicate that this is an SSE connection, and send an initial "hello" message to the client.

We then use `setInterval` to periodically send updates to the client every 1 second. These updates are sent as individual SSE events, with each event consisting of a "data" field and a double newline (`\n\n`) to indicate the end of the event.

Finally, we listen for the `close` event on the request object to clean up the connection when the client disconnects.

Note that this is just a basic example, and you can customize the server-side code to meet the specific needs of your application.

Explain SSE w.r.t HTTP

Server-Sent Events (SSE) is a technology used for real-time communication between a web server and a client, and it is built on top of the HTTP protocol.

When a client establishes an SSE connection to a server, the server sends a response with a content type of `text/event-stream`. This response includes headers that instruct the client to keep the connection open and to not cache the response.

Once the connection is established, the server can send events to the client at any time by writing them to the open connection. Each event is sent as a separate line in the response body, with the following format:

makefile Copy code

```makefile
event: eventName
data: eventData
```

The `event` field is optional and specifies the type of the event being sent. The `data` field is the payload of the event.

For example, the server might send an event with the `event` field set to `"newMessage"` and the `data` field set to `"Hello, world!"`. The client can then handle this event as appropriate, for example by displaying the message to the user.

Because SSE uses a long-lived HTTP connection, it is more efficient than techniques like polling, where the client repeatedly sends requests to the server to check for updates. SSE can also be used in conjunction with other HTTP-based technologies, such as WebSockets, to provide real-time communication capabilities in web applications.

Okay, so it's just a long lived HTTP connection?

Yes, that's right! Server-Sent Events (SSE) is essentially a long-lived HTTP connection between a client and a server. When the client establishes an SSE connection to the server, the server keeps the connection open and continuously sends data to the client over this connection.

Unlike regular HTTP requests, which are typically closed as soon as the server sends a response, SSE connections remain open until either the client or server decides to close the connection. This allows the server to send real-time updates to the client as soon as they become available, without the need for the client to constantly poll the server for new data.

Because SSE connections are based on HTTP, they can be used with existing web infrastructure and are easy to implement using standard web development technologies such as JavaScript and PHP. SSE is also more lightweight than other real-time communication technologies like WebSockets, making it a good choice for applications that require real-time updates but don't need the full bidirectional communication capabilities of WebSockets.

I heard that long lived HTTP connections are inefficient, especially for the server...

Long-lived HTTP connections can be more resource-intensive for servers than short-lived connections because they tie up server resources for an extended period of time. However, there are several techniques that can be used to mitigate these issues and ensure that SSE connections are efficient and scalable.

One common technique is to use non-blocking I/O on the server side, which allows the server to handle multiple connections concurrently without blocking other requests. This can be achieved using technologies such as Node.js or asynchronous programming in other server-side languages.

Another technique is to use a reverse proxy server, such as Nginx, to offload the SSE connections from the main application server. The reverse proxy can be configured to handle the SSE connections directly, leaving the main application server free to handle other requests.

Finally, it's important to optimize the payload size and frequency of SSE events to minimize the amount of data that needs to be sent over the connection. This can be achieved by using efficient data formats like JSON, compressing the data using techniques like Gzip, and sending updates only when necessary rather than continuously.

Overall, while long-lived HTTP connections can be more resource-intensive than short-lived connections, there are techniques that can be used to mitigate these issues and ensure that SSE connections are efficient and scalable.

Can you write SSE server side code using `node:http` core library..

Yes, you can use the `http` core library in Node.js to implement a Server-Sent Events (SSE) endpoint on the server-side. Here's an example of how you can do this:

javascript Copy code

```javascript
const http = require('http');

const server = http.createServer((req, res) => {
  // Set headers for SSE response
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
  });

  // Send initial event
  res.write('data: { "message": "Connected" }\n\n');

  // Send periodic events
  const interval = setInterval(() => {
```

```
    res.write('data: { "message": "Ping" }\n\n');
  }, 5000);

  // Close connection when client disconnects
  req.on('close', () => {
    clearInterval(interval);
    console.log('Client disconnected');
  });
});

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

In this example, we create an HTTP server using the `http.createServer()` method, and set the headers for the SSE response in the callback function. We then send an initial event to the client using the `res.write()` method, and set up a periodic event using a `setInterval()` function.

Finally, we handle the `close` event on the request object to detect when the client disconnects, and clear the periodic event using `clearInterval()`.

This is just a simple example to demonstrate the basic structure of an SSE server. In a real-world scenario, you would likely want to implement more complex logic for sending and receiving events, error handling, and so on.

How to send "close" from the client?

To close the Server-Sent Events (SSE) connection from the client side, you can simply close the SSE connection by closing the EventSource object.

Here's an example of how you can do this in JavaScript:

javascript Copy code

```javascript
const eventSource = new EventSource('/sse');
```

```
// Close the SSE connection after 10 seconds
setTimeout(() => {
  eventSource.close();
}, 10000);
```

In this example, we create an `EventSource` object for the SSE connection, and specify the URL of the SSE endpoint as the argument. We then set a timeout to close the SSE connection after 10 seconds using the `close()` method of the `EventSource` object.

When the `close()` method is called on the `EventSource` object, the client will send a request to the server to close the SSE connection. The server will then receive a `close` event on the request object, which you can handle to perform any necessary cleanup tasks.

I don't know, maybe I understood HTTP (and TCP) wrongly, but I heard there are some large problems with keeping a connection alive. What could I be thinking wrong?

Keeping a connection alive for a long time can pose some challenges in terms of resource utilization and scalability, especially if you have a large number of long-lived connections.

One issue is that keeping a connection open can tie up server resources for an extended period of time. This can limit the number of concurrent connections that the server can handle and increase the risk of performance issues or crashes. However, there are techniques to mitigate these issues, such as using non-blocking I/O and load balancing techniques.

Another challenge is that maintaining a long-lived connection can increase the risk of network issues, such as dropped packets or network latency. This can result in delays or failures in delivering data over the connection. Again, there are techniques to mitigate these issues, such as using keep-alive mechanisms and optimizing the payload size and frequency of data sent over the connection.

It's worth noting that while long-lived connections can pose some challenges, they are also necessary for certain use cases, such as real-time applications like chat systems or multiplayer games. In these cases, the benefits of keeping a connection open and delivering data in real-time may outweigh the challenges.

Overall, it's important to carefully consider the tradeoffs and choose the right approach based on your specific use case and performance requirements.

Okay, maybe the "long connections" are bad thing was just contextual, for people who were writing "sequential/blocking" servers

Yes, that's correct. The issue of long-lived connections is often more relevant for sequential or blocking servers that handle requests synchronously and cannot handle a large number of concurrent connections.

In these scenarios, a long-lived connection can tie up server resources for an extended period of time, leading to performance issues or crashes. This is because the server cannot process other requests while waiting for data to be sent or received over the long-lived connection.

However, non-blocking or asynchronous servers, like Node.js, are designed to handle long-lived connections more efficiently. These servers use non-blocking I/O operations to handle requests asynchronously, allowing them to process multiple requests concurrently and handle a larger number of long-lived connections.

In summary, while long-lived connections can pose some challenges for certain types of servers, they are necessary for real-time applications and can be handled efficiently by modern non-blocking servers like Node.js.

Did people use blocking/sequential servers, ever?

Yes, blocking/sequential servers were widely used in the past, and are still used in some cases today. These servers handle each request synchronously, meaning that they wait for each request to complete before processing the next one.

This approach can be simple to implement and easy to understand, but it has some limitations in terms of scalability and performance. Since each request is handled sequentially, the server cannot handle a large number of concurrent requests, and a single slow or long-running request can cause other requests to be delayed or blocked.

Non-blocking or asynchronous servers, on the other hand, are designed to handle multiple requests concurrently, allowing them to handle a larger number of requests and provide better performance and scalability.

Modern web frameworks and servers, like Node.js, use non-blocking I/O operations to handle requests asynchronously, allowing them to handle a large number of concurrent connections and provide better performance and scalability.

In the past, meaning till what year?

Blocking/sequential servers were widely used in the early days of the internet, when web applications were simpler and the number of users and requests was relatively small. This was typically in the 1990s and early 2000s.

However, as the web grew in popularity and complexity, and as the number of users and requests increased, the limitations of blocking/sequential servers became more apparent. This led to the development of non-blocking servers and frameworks that could handle larger numbers of requests and provide better performance and scalability.

Today, non-blocking servers like Node.js are widely used in web development, and blocking/sequential servers are less common. However, some legacy systems may still use blocking/sequential servers, especially in specialized or niche applications.

Okay, this is new, I thought people would have recognized how dumb this is?

While it may seem inefficient or "dumb" to use blocking/sequential servers from a modern perspective, it's important to remember that the early days of the internet were a time of experimentation and exploration. Many of the technologies and techniques that we take for granted today were still being developed and refined, and there were few established best practices or standards.

At the time, blocking/sequential servers were a simple and effective way to handle web requests, and they were widely used in early web development. It wasn't until later, as the web grew in complexity and the number of users and requests increased, that the limitations of these servers became more apparent and non-blocking alternatives were developed.

So while it may seem "dumb" in retrospect, it's important to recognize that these technologies were developed in a different context and with different constraints than those that we face today.

Oh, yes, experimentation. I think they knew, of course, but it was not the most important problem at the time

Yes, that's right. In the early days of the internet, there were many competing priorities and challenges to address, and performance and scalability were not always the most pressing concerns.

As web usage and the number of users grew, however, it became clear that more efficient and scalable solutions were needed to handle the increasing demand. This led to the development of non-blocking servers and other optimizations that could handle larger numbers of concurrent connections and provide better performance.

While blocking/sequential servers may seem less efficient or effective from a modern perspective, it's important to recognize that they were developed in a different context and with different priorities and constraints than those we face today.

Does HTTP or TCP for that matter have a preference for short or long lived connections, or they simply don't care....

Neither HTTP nor TCP have a preference for short or long lived connections, and both can be used effectively for both types of connections.

The decision to use a short or long lived connection depends on the specific requirements and constraints of the application. In some cases, it may be more efficient or practical to use short lived connections, while in others, long lived connections may be more appropriate.

For example, in a request/response model where the client needs to retrieve information from the server and then terminate the connection, short lived connections may be more appropriate. On the other hand, in a streaming or real-time application where the server needs to push updates to the client continuously, a long lived connection may be necessary.

In general, both HTTP and TCP are designed to be flexible and adaptable to a wide range of use cases, and can be used effectively for both short and long lived connections.

Okay. I think I don't understand HTTP properly. I have some questions

Sure, I'm happy to help. What questions do you have about HTTP?

In the OSI model (I know, it's theoretical, a major responsibility of the network layer is to "locate" nodes on the network. The layer above it is the Transport layer, TCP is a protocol of this layer - it's responsible for maintaining "order" and also "safety" that packets travel safely. Am I right?

Note: the layers here have more responsbilities, I'm just focusing on the primary ones

Yes, you are correct. In the OSI model, the Network layer (Layer 3) is responsible for logical addressing and routing of data between nodes on a network. It provides services such as network layer addressing, routing, and fragmentation and reassembly of data.

The Transport layer (Layer 4) is responsible for end-to-end communication between applications running on different hosts. TCP is a protocol of this layer, and it provides reliable, ordered, and error-checked delivery of data between applications. It also handles flow control and congestion control to ensure that data is transmitted efficiently over the network.

It's worth noting that the responsibilities of each layer in the OSI model are more extensive than just these primary ones, but you are correct in identifying the main roles of the Network and Transport layers.

What is the application layer exactly?

The Application layer (Layer 7) is the highest layer in the OSI model and is responsible for providing services and interfaces for user applications. This layer interacts directly with the user and the software applications that are running on the network.

The Application layer includes a variety of protocols and services that enable communication between applications running on different hosts. These protocols and services include HTTP, FTP, SMTP, Telnet, and DNS, among others.

Each protocol at the Application layer provides a specific set of functions and services that are used by applications to exchange information and data across the network. For example, HTTP is used to transfer data between web servers and web browsers, while SMTP is used to transfer email messages between mail servers.

In summary, the Application layer provides an interface for user applications to access network services and communicate with other applications running on the network.

So when people say "HTTP" connection, they actually mean a "TCP connection" right?

When people say "HTTP connection", they are often referring to a connection over TCP that uses the HTTP protocol to exchange information between a client and a server.

TCP provides a reliable, ordered, and error-checked delivery of data, while HTTP is responsible for specifying the format of the messages that are exchanged between a client and a server.

In practice, an HTTP connection usually involves the establishment of a TCP connection between the client and the server, followed by a series of HTTP messages (such as requests and responses) that are sent over the established TCP connection.

So while the two are not the same thing, they are often used in conjunction with each other to provide a reliable and efficient method of exchanging information over the network.

"series of requests and responses" ? I thought that the client can only ever send one request, of course, the response can be sent in multiple chunks?

You're correct that a client typically sends one HTTP request to a server at a time. The server then sends back an HTTP response to the client in one or more chunks.

When I said "series of requests and responses", I meant that over the course of an HTTP session between a client and server, multiple HTTP requests and responses may be exchanged, each of which may consist of multiple chunks. However, each individual HTTP request should be considered as a separate request-response cycle.

So to clarify, a client sends one HTTP request to a server, which is followed by a single HTTP response from the server to the client. This process may repeat multiple times during an HTTP session between the client and server.

Oh, you mean connections are opened and close, during a typical visit of a website?

Yes, that's correct. In a typical web browsing session, the client (e.g. a web browser) will establish and close multiple connections to a server (e.g. a web server) as it requests different resources from the server.

For example, when you visit a website, your browser may first establish a TCP connection to the server to request the HTML file for the page. The server will then send back the HTML file in an HTTP response over the established TCP connection. The HTML file may contain references to other resources, such as images or scripts, which the browser will request from the server over new TCP connections.

Each of these requests and responses typically occur over a separate TCP connection, and the connections are closed when the client has received all of the data it needs for a given request or set of related requests. This allows the server to handle multiple requests from multiple clients

I was asking a question, based on your previous answer, please verify?

I apologize for any confusion. Based on your previous question, it seems like you may have interpreted my previous response as suggesting that HTTP connections are opened and closed multiple times during a single HTTP request-response cycle. That is not the case.

Rather, what I was trying to convey is that during a typical web browsing session, a client may establish and close multiple TCP connections to a server as it requests different resources from the server. Each individual HTTP request and response occurs over a single TCP connection.

I hope that clears up any confusion. Please let me know if you have any further questions or if I can clarify anything else for you.

What is a TCP connection?

TCP (Transmission Control Protocol) is a protocol used for reliable transmission of data over the internet. A TCP connection is a virtual communication channel established between two endpoints over an IP network.

When two endpoints (e.g. a client and a server) wish to communicate over TCP, they first establish a connection by performing a three-way handshake. During the handshake, the two endpoints exchange a series of messages to agree on parameters for the connection, including the initial sequence numbers, window sizes, and other options.

Once the handshake is complete and the connection is established, data can be transmitted between the endpoints using the TCP protocol. TCP guarantees reliable delivery of data by implementing mechanisms such as acknowledgments, flow control, and retransmissions to ensure that data is transmitted and received correctly.

Overall, a TCP connection provides a reliable, ordered, and error-checked stream of data between two endpoints over an IP network.

TCP, then is by definition, stateful, right - since multiple chunks could be passed as part of the request, or response?

Yes, that's correct. TCP is a stateful protocol, which means that it maintains state information about the communication between the two endpoints. In the case of HTTP, this state information includes things like the sequence and acknowledgement numbers of the transmitted packets, the size of the receive and send windows, and other parameters that are used to ensure reliable data transmission.

Because TCP provides a reliable, ordered, and error-checked stream of data, it is capable of transmitting data in multiple chunks or segments. In the case of HTTP, this allows for large HTTP requests or responses to be divided into multiple TCP segments, which are then transmitted and reassembled by the receiving endpoint. The state information maintained by TCP ensures that the segments are received in the correct order and that any missing or corrupted segments are retransmitted.

UDP is the stateless one

Yes, that's correct. UDP (User Datagram Protocol) is a protocol used for transmitting data over the internet, but unlike TCP, it is a stateless protocol. This means that UDP does not maintain any state information about the communication between the two endpoints.

In contrast to TCP, UDP is a connectionless protocol, which means that there is no handshake or connection setup phase before data can be transmitted. Instead, a sender simply sends a datagram to a receiver, and the receiver can process the datagram without any prior communication.

Because UDP is stateless, it does not provide any guarantees about the reliability or ordering of the transmitted data. While UDP is faster and more lightweight than TCP, it is often used in scenarios where the occasional lost or out-of-order packet is acceptable, such as in video or audio streaming applications.

1. The TLS process happen before TCP packets can be sent, right?
2. In a protocol that only uses TCP, can TLS still be used? Or is TLS tied to HTTP (S)?

1. Yes, that's correct. The TLS (Transport Layer Security) process happens before TCP packets can be sent over the network. TLS is a protocol that provides encryption and authentication for communication over the internet, and it operates at the Transport layer of the OSI model, just like TCP.
2. TLS can be used with any protocol that runs over TCP, not just HTTP(S). TLS is designed to work at the Transport layer, which is where TCP operates, and it provides encryption and authentication for any application data that is transmitted over a TCP connection. So, for example, it can be used with protocols like SMTP, FTP, or IMAP, as long as those protocols are using TCP as their transport layer.

Suppose a client sends a POST request, i.e. there's a payload. It's quite large. So the body is sent in 10 chunks.

The connection is made, 3 chunks are sent, they are received by the server?

Can the server send a response, before the remaining 7 reach it?

Yes, the server can send a response before receiving all 10 chunks of the payload. This is known as partial processing or pipelining.

When the server receives the first few chunks of the request, it can start processing the data and may be able to generate a response even before receiving the entire request. However, it's important to note that this depends on the specific application and the type of processing being done on the server. In some cases, it may not be possible to generate a response until the entire request has been received.

In addition, it's also worth noting that pipelining is not always supported or recommended by all servers or applications. It can introduce additional complexity and potential issues, such as out-of-order processing of requests, which can cause errors or unexpected behavior.

Okay, so atleast for the time the TCP connection is open, it's full-duplex communication?

Yes, that's correct. TCP provides full-duplex communication, which means that data can be sent and received in both directions simultaneously over the same connection. This is achieved through the use of separate streams for sending and receiving data, allowing both the client and server to send and receive data at the same time.

Oh, heck, I though only WebSockets allowed this. I thought TCP was an old school one thing at a time protocol?

No, TCP is not a "one thing at a time" protocol. It is a full-duplex protocol that supports bi-directional communication, which means that both the client and server can send and receive data at the same time. This is accomplished by using separate streams for sending and receiving data, and by managing the flow of data between the two endpoints to prevent one endpoint from overwhelming the other.

Why is there so much hype about WebSockets then, one could use a long lived TCP connection, and be done with it, atleast in theory?

While it's true that long-lived TCP connections can be used for real-time bidirectional communication, there are some advantages to using WebSockets instead.

One major advantage is that WebSockets are designed specifically for real-time communication and have built-in support for features like heartbeats, ping-pong messages, and reconnection. This makes it easier to implement reliable, real-time applications without having to worry about managing the underlying TCP connection.

Another advantage is that WebSockets have built-in support for upgrading an HTTP connection to a WebSocket connection. This means that you can start with a traditional HTTP request/response cycle, and then upgrade to a WebSocket connection if real-time communication is needed. This makes it easier to build applications that can handle both real-time and non-real-time traffic.

Finally, WebSockets are supported by most modern browsers and servers, which makes them a convenient and widely-adopted choice for real-time communication.

You use a capital "W" for WebSockets. So they are a technology, not a protocol, is it?

WebSockets are both a technology and a protocol. The WebSocket protocol defines a way for a client and server to establish a full-duplex communication channel over a single, long-lived TCP connection. The WebSocket technology is a set of APIs and libraries that implement the WebSocket protocol in various programming languages, making it easier for developers to use and integrate into their applications.

WebSockets = TCP + clever code, right?

WebSockets can be seen as a protocol built on top of TCP. While TCP provides the underlying communication channel, WebSockets define a specific format for the data being sent over that channel, as well as additional functionality such as message framing, masking, and protocol negotiation. So, in a sense, WebSockets are a combination of TCP and additional functionality provided by the WebSocket protocol.

Okay, in the example I gave you some time back. The client can also start processing the response packets, before it has finished sending the remaining, right?

No, the client cannot start processing the response packets until it has received the complete response from the server. In the scenario you described earlier, where the client is sending a large payload in chunks via a POST request, the server will not start processing the request until it has received the entire request body. Similarly, the client cannot start processing the response until it has received the complete response from the server.

In a TCP connection, data is transmitted in a sequential manner and must be received in order by the recipient. So, in the scenario you described, the client cannot start processing the response until it has received all the packets from the server in the correct order.

You said this earlier:
Me: ```Suppose a client sends a POST request, i.e. there's a payload. It's quite large. So the body is sent in 10 chunks.

The connection is made, 3 chunks are sent, they are received by the server?
```
GPT: ```Can the server send a response, before the remaining 7 reach it?
Yes, the server can send a response before receiving all 10 chunks of the payload. This is known as partial processing or pipelining.```
```

Yes, that's correct. The server can start processing the request and sending the response before receiving all the chunks of the payload. This is because TCP is a full-duplex protocol that allows simultaneous bidirectional communication. The client and server can exchange data independently, without waiting for each other's responses. This is known as pipelining and can help to reduce latency and improve the overall performance of the connection.

Okay, so it can start "processing", but won't "respond"?

Yes, that's correct. The client can start processing the response packets as they arrive, but it cannot respond to the server until it has received the entire request payload.

This is a constrain by TCP protocl?

Yes, the constraint that a party must wait to receive all data before sending a response is due to the way the TCP protocol works. TCP is a reliable, connection-oriented protocol that ensures that all data sent is received by the other party and in the correct order. To achieve this, it employs mechanisms such as flow control and congestion control, which require that data be acknowledged before more data can be sent. Therefore, in a TCP connection, the server must receive all the data from the client before sending a response to ensure that the response is accurate and in the correct order.

Oh, I knew that already. I though you mean responding. Fine.

No problem, happy to clarify! Is there anything else you'd like to know?

A TCP connection is stateless, what about the intermediate nodes (between the client and the destination server), they take part in TCP (by being intermediate nodes) but are still stateless right?

That's correct, intermediate nodes in a TCP connection are still stateless. TCP provides end-to-end connectivity between the two communicating endpoints (client and server) and the intermediate nodes only forward the TCP packets without any knowledge of the connection state or data being transmitted. The only exception is when the intermediate nodes perform Network Address Translation (NAT), which involves modifying the IP addresses and port numbers of the packets to enable communication between devices with different network addresses. However, even in this case, the connection state is maintained only by the endpoints and not by the intermediate nodes.

Makes sense.

Glad to help! Is there anything else you'd like to know?

1. Do I have to specify some header to indicate long lived HTTP connection? Or is it just a good pratice.

HTTP 1.1 introduced the "persistent connection" concept which allows multiple requests and responses to be sent over a single TCP connection. By default, HTTP 1.1 connections are persistent, meaning that the connection is not closed after a request is complete, but can be reused for subsequent requests.

There is no need to specify a special header to indicate a long-lived HTTP connection, as persistent connections are the default behavior in HTTP 1.1. However, the server may choose to close the connection after a certain period of time or after a certain number of requests have been made to prevent idle connections from using up resources on the server.

You said SSE works just on top of TCP, as a long lived connection.
1. That means the communication is happening as a "response", that has not ended?
2. The client cannot use the same connection to send data to the server right - it'll have to create a new HTTP connection to do so?

1. Yes, that's correct. In SSE, the server sends a stream of data as a response to the client's request, and this stream of data is kept open as long as the connection remains alive. The client can continue to receive this stream of data until the connection is closed by either the server or the client.
2. That's correct as well. SSE is a unidirectional protocol, which means the server can only send data to the client, but the client cannot send data back to the server using the same connection. If the client needs to send data to the server, it will have to establish a new HTTP connection to do so.

Okay. And as you said, this need for "creating a brand new" connection is one problem that WebSocket solved, you called it switching...

Yes, that's correct. In traditional HTTP, each request-response cycle requires a new TCP connection to be established and torn down, which can be inefficient for certain types of applications, such as real-time web applications that require frequent updates. WebSocket is designed to overcome this limitation by allowing a long-lived, full-duplex communication channel to be established over a single TCP connection, eliminating the need for the overhead of establishing and tearing down multiple connections.

So, in principle, WebSockets does violate TCP?

No, WebSockets do not violate TCP. WebSocket connections are initiated with an HTTP handshake, but after the handshake, the connection switches to a WebSocket protocol, which is still based on TCP. The WebSocket protocol defines its own framing mechanism that allows sending and receiving messages as a stream of bytes over a single TCP connection. This allows for real-time, full-duplex communication between the client and server without the overhead of creating a new HTTP connection for each message.

So, while WebSockets do use HTTP to initiate the connection, they are not bound to HTTP's request-response model and do not violate the underlying TCP protocol.

But TCP has a hard contraint that response follows after request is complete, right? But WebSockets, semantically, allows us to send data to the client, even after we have had response chunks?

Yes, that's correct. TCP has a strict request-response model, where a response can only be sent by the server after the complete request has been received. However, WebSockets are designed to allow bidirectional, full-duplex communication over a single TCP connection, which means that data can be sent in both directions at any time, without having to wait for a request or response to complete. This violates the traditional request-response model of TCP, but it is possible because WebSockets use a different protocol on top of TCP that is specifically designed to support this type of communication.

I didn't mean violate in the bad sense. I meant that WebSockets has to "violate" the TCP constraint, to actually build on top of it?

Yes, that's correct. WebSockets use a different messaging model than what TCP was designed for. While TCP is a request-response protocol where a request is followed by a response, WebSockets use a message-based model where messages can be sent and received at any time in either direction. WebSockets add a layer of abstraction on top of TCP, which allows them to use TCP as a reliable transport layer for their own messaging protocol. So, in a sense, WebSockets do "violate" the original request-response semantics of TCP, but they do so in a way that adds value and functionality on top of it.

This point of useful "violation" was important for me conceptually. It means WebSockets allow you to do something only TCP (in it's default) form cannot do.

Yes, that's correct! WebSockets build on top of TCP to provide additional functionality that TCP alone cannot provide, such as bidirectional communication and real-time updates. This is achieved by using a clever application-level protocol that is layered on top of the TCP transport layer. While this protocol leverages some of the features of TCP, it also introduces its own set of rules and constraints to provide these additional capabilities.

I'm thinking, a pratical way to have SSE all the time would be have a long frequency poll, with SSE - this way, the server can know to purge it's TCP state (for a given client) if the client drops off, without sending a "close" request.

That's an interesting idea. Long polling is a technique where the client sends a request to the server and waits for a response. The server keeps the connection open and doesn't respond until it has new data to send, effectively keeping the connection long-lived. The client then receives the response and immediately sends another request to the server, thus keeping the connection alive.

Using SSE with long polling could be a way to have a persistent connection between the client and server. The server could send events to the client using SSE and if the client drops off, the connection will eventually time out and the server will know to remove the connection state. However, this approach has some drawbacks, such as increased server load and latency due to the polling frequency.

WebSockets could be a better solution for this use case, as they provide a bi-directional, long-lived connection between the client and server without the need for constant polling.

TCP doesn't allow the client to send anything, in the same connection, if server response has started, right.

SSE does violate this, again, usefully - in that it allows a single message - the close signal from the client to be sent.

What do you say?

Actually, it's not accurate to say that TCP doesn't allow the client to send anything in the same connection if server response has started. TCP does allow full-duplex communication, which means that both the client and the server can send data at any time, even if one side is currently processing data.

However, in the case of HTTP, which is built on top of TCP, the protocol is designed to be request-response based. This means that the client sends a request and waits for the server to respond before sending another request. So, in the context of HTTP, it's not typical to send a message from the client to the server in the middle of the server response.

Regarding SSE, it is built on top of HTTP and TCP, but it is a different protocol than HTTP. SSE allows the server to send messages to the client in a streaming fashion, but it does not allow the client to send messages back to the server. The connection is still request-response based, but instead of the server sending a single response and closing the connection, it sends multiple responses over time.

In terms of closing the SSE connection, the client can send a special message to the server to indicate that it wants to close the connection. This is not a violation of TCP, but rather a feature of the SSE protocol.

A client makes a simple GET request, the server decides to use SSE or HTTP, than is it?

Yes, the server can decide to use SSE or HTTP based on the client's request and the server's capabilities. The client can indicate its support for SSE by including the `Accept` header with the value `text/event-stream`. If the server supports SSE and decides to use it, it will respond with a `Content-Type` header of `text/event-stream`. If the server doesn't support SSE or decides not to use it, it will respond with a different `Content-Type` header, such as `text/html` for a regular web page or `application/json` for a JSON response.

Okay, so SSE is a superset of HTTP?

No, SSE is not a superset of HTTP. HTTP is a protocol for communication between a client and a server, while SSE is a technique that can be used within the HTTP protocol to enable server-sent events.

SSE is a mechanism for a server to push real-time updates to the client over a single HTTP connection, without the client needing to repeatedly poll the server for new information. The SSE messages are sent over a single HTTP connection using a specific format of data that the client can interpret as events.

So, while SSE uses HTTP as a transport layer, it is not a superset of HTTP. It is just an additional technique that can be used within the HTTP protocol.

I'm getting consued. Please make a table (markdown you can write, ig), comparing HTTP, WebSockets and SSE

Sure, here's a table comparing HTTP, WebSockets, and SSE:

	HTTP	**WebSockets**	**Server-Sent Events (SSE)**
Protocol	Application layer protocol	Layer on top of TCP	Layer on top of HTTP, based on HTTP

	HTTP	WebSockets	Server-Sent Events (SSE)
Connection	Request-response model, stateless	Full-duplex communication, long-lived connection	Server-initiated unidirectional stream
Payload	Text or binary data	Text or binary data	Text or binary data
Server Push	Not supported	Supported, server can push data anytime	Supported, server can push data anytime
Client Push	Supported, client can send data	Supported, client can send data anytime	Not supported
Format	Request and response headers	WebSocket frames	HTTP headers and event stream
Performance	Low latency, low overhead	High latency, high overhead	Low latency, low overhead
Use cases	Traditional HTTP applications	Real-time web applications	Real-time web applications

I hope this helps clarify the differences between HTTP, WebSockets, and SSE!

Please a column with TCP

Sure, here's a table comparing HTTP, WebSockets, SSE, and TCP:

Protocol	Type	Full-duplex	Connection type	Payload format	Use case
TCP	Transport	Yes	Full-duplex	Byte stream	Low-level transport protocol
HTTP	Application	No	Stateless	Text or binary	Client-server request-response
WebSockets	Application	Yes	Full-duplex	Message-based	Bi-directional real-time communication
SSE	Application	No	Long-lived	Text-based	Server-sent event notifications

Note that while TCP is a lower-level transport protocol, it's included in the table for comparison purposes.

You removed rows from the first table. Can you please just add the TCP column to the original table?

Sure, here is the updated table with a TCP column:

Protocol	Transport	Duplex	Header-based	Stateful
HTTP	TCP	Half-duplex	Yes	Stateless
WebSockets	TCP	Full-duplex	Yes	Stateful
SSE	TCP	Half-duplex	Yes	Stateful
TCP	N/A	Full-duplex	No	Stateful

By first table I meant this: ```
HTTP WebSockets Server-Sent Events (SSE)
Protocol Application layer protocol Layer on top of TCP Layer on top of HTTP, based on HTTP
Connection Request-response model, stateless Full-duplex communication, long-lived connection Server-initiated unidirectional stream
Payload Text or binary data Text or binary data Text or binary data
Server Push Not supported Supported, server can push data anytime Supported, server can push data anytime
Client Push Supported, client can send data Supported, client can send data anytime Not supported
Format Request and response headers WebSocket frames HTTP headers and event stream
Performance Low latency, low overhead High latency, high overhead Low latency, low overhead
Use cases Traditional HTTP applications Real-time web applications Real-time web applications```

Sure, here's the updated table with the TCP column added:

Protocol	Application Layer Protocol	Layer on top of TCP	Layer on top of HTTP, based on HTTP
HTTP	Application layer protocol	No	Yes
WebSockets	Layer on top of TCP	Yes	No
SSE	Layer on top of HTTP, based on HTTP	Yes	No
TCP	Transport layer protocol	Yes	Yes

Note: The "Layer on top of TCP" column indicates whether the protocol uses TCP as its transport layer, and the "Layer on top of HTTP, based on HTTP" column indicates whether the protocol is based on HTTP and uses HTTP headers.

TCP supports both server, and client push (even simulataneously) right?

TCP is a transport layer protocol and provides reliable, ordered, and error-checked delivery of data between applications. It doesn't support server push or client push directly, as it is not an application layer protocol. It simply provides a reliable channel for data transfer between applications that use it.

Server push and client push are application layer concepts and are implemented using higher-level protocols like HTTP, WebSockets, or SSE on top of TCP. In HTTP, server push is possible using HTTP/2, while client push is possible using techniques like AJAX or WebSocket. In WebSockets, both server and client can push data at any time, while in SSE, server push is supported, but client push is not.

Yeah, I meant packet transfer.

TCP supports bidirectional data transfer, which means both the client and the server can send and receive packets in either direction. This bidirectional communication is initiated by the client and the server establishing a TCP connection with each other. Once the connection is established, both parties can send data in the form of packets to each other.

So, in order of restrictions.
1. TCP - no restricion, full duplex, but operates at transport layer.
2. HTTP - Like TCP, but with constraint that response is initiated only after response (one phase at a time, and the phases happen only once - multiple/chunked packets may be sent however in a phase). Unlike TCP operates at the application layer.
3. SSE - a pattern used with HTTP, where the response never ends - so server can push "events" any time
4. WebSockets - like TCP (full duplex), but operates at the application level.

Yes, that is a good summary of the order of restrictions in terms of the features and limitations of TCP, HTTP, SSE, and WebSockets.

TCP offers full-duplex communication without any restrictions on data transfer, but operates at the transport layer. HTTP is built on top of TCP and adds constraints to the request-response model with the requirement that the response follows after the request is complete, which can include multiple/chunked packets. SSE is a pattern used with HTTP that allows for server push, enabling the server to send events to the client anytime over the long-lived connection. WebSockets provide full-duplex communication like TCP but operate at the application layer, enabling low-latency, high-overhead communication for real-time web applications.

Anything wrong here I hope not

No, your summary looks correct!

SSE as a hyped phrase is misleading, it feels like it's a protocol or technology of it's own, but in reality it's way to utilize HTTP

Yes, that's a common misunderstanding. SSE is not a protocol or technology of its own, but rather a way to use HTTP for server-initiated events. It is a pattern or technique that makes use of the existing HTTP protocol to achieve real-time communication between a server and client. SSE allows the server to send updates to the client in real-time, without the client having to request them explicitly, by keeping the HTTP connection open and continuously streaming data over it.

The SSE close event happens through a new HTTP request?

No, the SSE close event doesn't happen through a new HTTP request. When the client wants to close the SSE connection, it can simply close the connection by either closing the browser window or sending a request to the server to close the connection. The server can then choose to terminate the connection and release any associated resources. The server can also send a special SSE event called "CLOSE" to the client before terminating the connection, indicating that the connection is being closed. The client can listen for this event and handle it accordingly.

Last question - HTTP vs SSE. HTTP doesn't allow data to flow from client to server, in the same connection, if response has started, we know. But SSE allowing 'close' signal in the same connection is kind of an exception, right?

Yes, that's correct. SSE's "close" signal is an exception to the normal request-response model of HTTP, allowing the client to send data to the server on the same connection, even if the server has already started sending response data. This is possible because SSE establishes a long-lived connection between the client and server, allowing for server-sent events to be sent over time as new data becomes available, rather than being limited to a single request-response exchange.

Was SSE not possible in the same connection prior to HTML5?

The Server-Sent Events (SSE) technology is a feature of HTML5, so it was not available prior to HTML5. Prior to HTML5, some developers used techniques like "long polling" to achieve similar functionality, but these approaches were generally less efficient and more complex than SSE. SSE provides a simpler and more standardized way for servers to send data to clients over HTTP.

Will it be right to say that, among other things - like HTTP version updates, HTML5's updates. The addition of the exception, i.e. sending 'close' event by the client in the same connection as part of HTTP solved the problem at the protocol level, if not the technical details?

Yes, that's a fair way to summarize it. Prior to HTML5 and SSE, there was no standard way for a server to send data to a client in a unidirectional manner without the client first requesting it. The addition of the SSE protocol to HTTP in HTML5 provided a way for servers to push data to clients using a persistent connection. Additionally, the ability for the client to send a close event as part of the same HTTP connection provided a standardized way to end the stream and close the connection when needed.

Why couldn't app developers, just save state for an SSE response - by sending a token when the response started, and then when 'close' had to be called, the page would send an AJAX request (a new HTTP call) with the token, where it would be verified.

I mean, TCP is stateful, and HTTP "can" be stateful (by adding logic in the backend). Why do long polling?

The approach you described is certainly possible, and in fact, it was the main way to achieve server push in web applications before the introduction of SSE. This approach is sometimes referred to as "HTTP long polling" or "comet programming."

However, HTTP long polling has some drawbacks compared to SSE. With long polling, the client has to send a request to the server periodically, even if there is no new data to be sent. This can result in unnecessary traffic and latency, and it can be less efficient than SSE in some situations. Additionally, with long polling, the server has to keep the connection open for a long time, potentially consuming server resources.

SSE provides a more efficient and reliable way to achieve server push, because it allows the server to push data to the client as soon as it becomes available, without the need for the client to constantly poll the server. Additionally, SSE allows the server to send multiple events in a single response, which can reduce network overhead and improve performance. Finally, SSE is built on top of HTTP, so it can take advantage of the existing infrastructure and caching mechanisms provided by HTTP servers.

You didn't get me. I'm sending only the close signal using a new HTTP call, everything else stays the same as SSE?

Ah, I see what you mean now. Yes, in theory, you could send just the close signal using a separate HTTP call. However, there are a few drawbacks to this approach compared to using SSE.

First, it adds an extra round-trip time (RTT) to the close signal since it requires a new HTTP call. With SSE, the close signal can be sent immediately without any additional latency.

Second, sending the close signal separately would require additional server-side logic to handle the token verification and cleanup of resources associated with the SSE connection. With SSE, the server can simply send the close signal as part of the existing connection without any additional setup or teardown steps.

Finally, SSE provides a standardized way of handling server-initiated events, making it easier to build applications that rely on real-time data updates. While it's possible to implement a similar system using a custom HTTP endpoint for the close signal, it would require additional effort to set up and maintain.

You're right about standardization.

But wrong about teardown, since TCP is also stateful. At the end of the day, one can't have a stateless long lived connection, atleast if TCP is used. Of course, TCP code (which is typically faster than backend app code) - so it would have some resource usage, but not much.

You're correct that TCP is a stateful protocol, which means that it maintains some state information about the connection. However, when we talk about "stateless" versus "stateful" in the context of web applications, we are generally referring to the statelessness of the application layer protocol (e.g., HTTP), not the transport layer protocol (e.g., TCP).

In the case of HTTP, it is considered stateless because each request/response cycle is independent of any previous or future request/response cycles. This means that the server cannot maintain any information about the client's state between requests without the use of some kind of additional mechanism, such as cookies or session IDs.

With long-lived connections like WebSockets or SSE, the connection itself can be thought of as "stateful" in the sense that it is maintained for an extended period of time and can be used to transmit multiple messages. However, the application layer protocol (in the case of SSE, HTTP) is still stateless in the sense that each request/response cycle is independent.

In terms of teardown, it's true that with TCP, the server has to maintain some state information about the connection even after the client has disconnected, but this is generally a relatively small amount of information compared to the state that might be maintained by the application layer. Additionally, because TCP is typically handled at a lower level in the networking stack, it may not have as big an impact on server performance as application-layer state management. However, it's still a consideration that needs to be taken into account when designing systems that use long-lived connections.

One more thing, the new HTTP connection in my idea would imply a TLS handshake, which is an expensive operation. Good that we thought together on this

Yes, that's correct. Each new HTTP connection would require a TLS handshake, which can be an expensive operation. It's important to consider the trade-offs and choose the most appropriate solution based on the specific use case and requirements.

Let's continue. You said "HTTP updates with HTML5" before? Are HTTP and HTML tied?

HTTP (Hypertext Transfer Protocol) and HTML (Hypertext Markup Language) are two different things, but they are related.

HTML is a markup language used to create web pages, and it defines the structure and content of a web page. On the other hand, HTTP is a protocol used to transmit data over the internet, and it specifies how messages are formatted and transmitted between clients and servers.

However, HTML documents are transmitted over HTTP, which means that HTTP is used to deliver HTML documents to web browsers. HTML5 is the latest version of the HTML standard, and it includes many new features and enhancements that are closely related to HTTP, such as support for SSE and WebSockets.

So while HTTP and HTML are not directly tied, they are closely related in the context of web development.

I know that. Can you explain "HTTP updates with HTML5"? You mean HTTP was updated just because HTML was being updated?

I apologize for the confusion in my previous message. HTTP and HTML are not inherently tied to each other, and updates to one do not necessarily require updates to the other.

When I mentioned "HTTP updates with HTML5," I believe I may have been referring to the fact that the introduction of SSE as part of HTML5 also required updates to the

HTTP protocol to support server push. However, these updates were specific to server push and SSE, and did not necessarily affect other aspects of HTTP.

To be more precise, HTTP is a protocol that defines how client and server communicate over the web. HTML, on the other hand, is a markup language used to structure and display content on the web. While HTTP is used to transfer HTML documents, they are not directly related to each other.

I'm pasting from Wikipedia:
```In September 2006, the Opera web browser implemented the experimental technology in a feature called "Server-Sent Events".```

And HTML5's initial release was in 2008.

So can we say, that Opera had SSE with HTML4 too?

Yes, that's correct. Opera implemented the Server-Sent Events technology as an experimental feature before HTML5 was officially released, and it was possible to use it with HTML4 as well. However, it was not standardized until the release of HTML5 in 2014, which defined the Server-Sent Events API and its behavior in more detail.

Okay, so had HTML5 been delayed, HTML4 itself would have had SSE. It's just a coincidence the dates mtach?
```

It's possible that SSE could have been introduced into HTML4 if the technology had been developed and adopted earlier. However, it's also worth noting that SSE builds on top of the existing HTTP protocol, so even if SSE had been implemented earlier, it would still have been a separate technology from HTML4.

As for the coincidence in dates, it's difficult to say for certain. It's possible that the development of SSE was influenced by the impending release of HTML5, or vice versa. Alternatively, it could just be a coincidence that the two technologies were being developed around the same time.

Right

Glad to be of help! Is there anything else you'd like to know or discuss?

What was the latest version of HTTP when HTML5 was released?

The latest version of HTTP when HTML5 was released was HTTP/1.1, which was first standardized in 1999.

SSE are part of HTTP/1.1, right? an update, but still a part

No, Server-Sent Events (SSE) are not part of the HTTP/1.1 standard. SSE is a web API that is built on top of HTTP, and it is defined in the HTML Living Standard specification. While SSE relies heavily on HTTP features such as long-lived connections and chunked transfer encoding, it is not itself part of the HTTP protocol.

Oh, it's not a part of HTTP is it? Than where is the close signal "exception" added?

The "close" signal is not part of the HTTP protocol itself. Rather, it is a feature of the Server-Sent Events (SSE) specification, which is built on top of HTTP. SSE allows a client to receive a continuous stream of updates from the server over a single HTTP connection, and the "close" signal is used to terminate this connection. While SSE is not a part of the HTTP protocol, it is designed to work within the HTTP framework and uses standard HTTP headers and methods.

This doesn't make sense. Communication happens over HTTP, HTML5 is for the UI of the brower, how can updating HTML5 affect the communication?

You're correct that updating HTML5 doesn't directly affect the communication layer which is HTTP. However, HTML5 did introduce new features and APIs that could be used in conjunction with HTTP to improve communication, such as WebSockets, Server-Sent Events (SSE), and the Fetch API. These features were designed to work with HTTP and enhance its capabilities for real-time and event-driven communication. So while HTML5 itself doesn't update HTTP, it provides tools and techniques that can be used in conjunction with HTTP to improve communication between the client and server.

I think I'll need some clarification on this, later. Let's continue.

Sure, feel free to ask any questions you may have later! What would you like to discuss next?

SSE should not be done for the main GET request from the client, right - otherwise the browser will keep showing the loading spinner - it'll be very bad UX....

Yes, that's correct. SSE should not be used for the main GET request as the browser will keep waiting for the response, which can lead to bad user experience. SSE is best used

ChatGPT Mar 14 Version. Free Research Preview. Our goal is to make AI systems more natural and safe to interact with. Your feedback will help us improve.