FullStack.Cafe - Kill Your Tech Interview

Q1: What is WebSockets? ☆

Topics: WebSockets

Answer:

WebSocket is a technology that allows a client to establish two-way *full-duplex* communication with the server.

The key word in that definition is two-way: with WebSocket, both the client and the server can trigger communication with one another, and both can send messages, at the *same time*. By a contrast In a traditional HTTP system communication can only be initiated in one direction: from the client to the server.

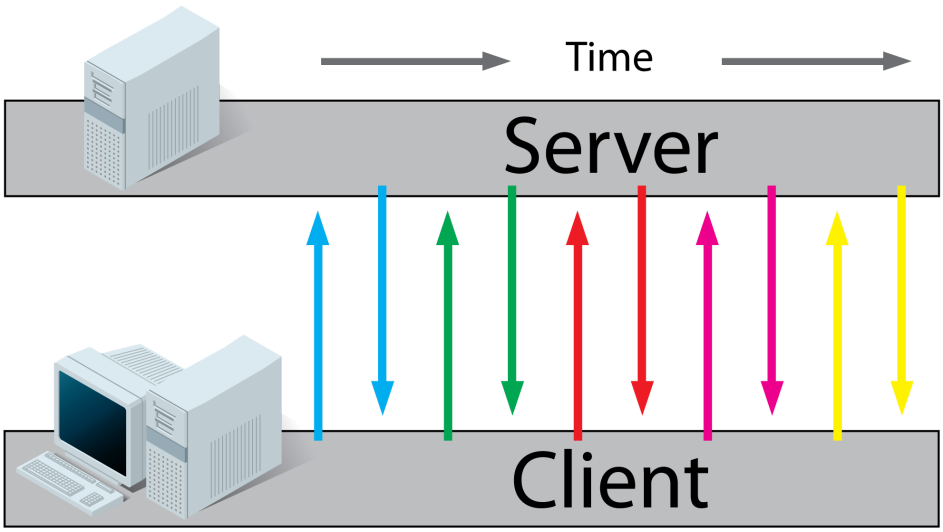
Q2: What is *Short Polling* and what problems do we have with it? ☆☆

Topics: WebSockets

Answer:

Short Polling or Ajax Polling is a technique when we have the client ping the server repeatedly, say, every 500ms (or over some fixed delay). That way, you get new data every 500ms:

1. client requests a webpage from a server using regular HTTP.
2. The client receives the requested webpage and executes the JavaScript on the page which requests a file from the server at regular intervals (e.g. 0.5 seconds).
3. The server calculates each response and sends it back, just like normal HTTP traffic.



There are a few obvious downsides to this:

- there's a 500ms delay,
- it consumes server resources with a barrage of requests, and most requests will return empty if the data isn't frequently updated.

Q3: Explain what is *Server-Sent Events (SSE) / EventSource*? ☆☆

Topics: WebSockets

Answer:

Alongside with short and long polling, **SSE** is a technique for sending messages is the Server-Sent Events API, which allows the server to push updates to the client by leveraging the JavaScript `EventSource`. EventSource opens a persistent, one-directional connection with the server *over HTTP* using a special text/event-stream header and listens for messages, which are treated like JavaScript events by your code.

Server-Sent Events (SSE) are great for apps where you don't need to send the server any data—for example, a Twitter-style news feed or a real-time dashboard of stock quotes. Another pro is that Server-Sent Events work over HTTP and the API is relatively easy to use.

However:

- SSE is not supported by older browsers,
- most browsers limit the number of SSE connections you can make at the same time.

Q4: What do you mean by *lower latency interaction*? ☆☆

Topics: WebSockets Software Architecture

Answer:

Low latency means that there is very little delay between the time you request something and the time you get a response. As it applies to webSockets, it just means that data can be sent quicker (particularly over slow links) because the connection has already been established so no extra packet roundtrips are required to establish the TCP connection.

Q5: Why use WebSocket over HTTP? ☆☆

Topics: WebSockets Software Architecture

Answer:

A **WebSocket** is a continuous connection between client and server. That continuous connection allows the following:

1. Data can be sent from server to client at any time, without the client even requesting it. This is often called server-push and is very valuable for applications where the client needs to know fairly quickly when something happens on the server (like a new chat messages has been received or a new price has been udpated). A client cannot be pushed data over http. The client would have to regularly poll by making an http request every few seconds in order to get timely new data. Client polling is not efficient.

2. Data can be sent either way very efficiently. Because the connection is already established and a webSocket data frame is very efficiently organized, one can send data a lot more efficiently that via an HTTP request that necessarily contains headers, cookies, etc...

Q6: What is the difference between *WebSockets* vs. *Server-Sent Events/EventSource*? ☆☆☆

Topics: WebSockets

Answer:

Websockets and SSE (Server Sent Events) are both capable of pushing data to browsers, however they are not competing technologies.

- **Websockets** connections can both send data to the browser and receive data from the browser. A good example of an application that could use websockets is a chat application.

- **SSE** connections can only push data to the browser. Online stock quotes, or twitters updating timeline or feed are good examples of an application that could benefit from SSE.

It can be overkill for some types of application to use WebSockets, and the backend could be easier to implement with a protocol such as SSE. Furthermore SSE can be polyfilled into older browsers that do not support it natively using just JavaScript.

Q7: Mention some advantages of *SSE* over WebSockets ☆☆☆

Topics: WebSockets

Answer:

Advantages of SSE over Websockets:

- Transported over simple HTTP instead of a custom protocol
- Can be poly-filled with javascript to "backport" SSE to browsers that do not support it yet.
- Built in support for re-connection and event-id
- Simpler protocol
- No trouble with corporate firewalls doing packet inspection

Ideal use cases of SSE:

- Stock ticker streaming
- twitter feed updating
- Notifications to browser

Q8: Explain what is *Long Polling*? ☆☆☆

Topics: WebSockets

Answer:

Using **Long polling** (or Ajax Long Polling) method the server receives a request, but doesn’t respond to it until it gets **new data** from another request.

Long polling is more *efficient* than pinging the server repeatedly since it saves the hassle of parsing request headers, querying for new data, and sending often-empty responses.

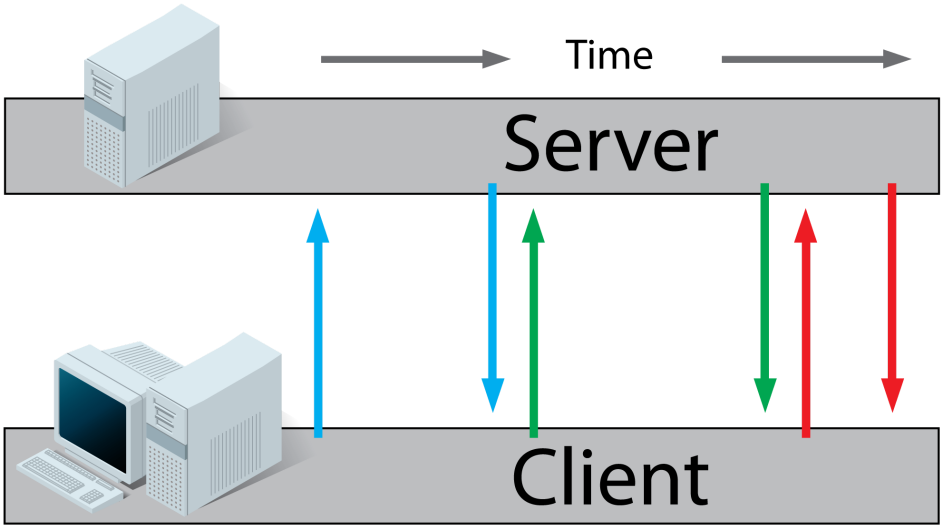
However

- the server must now keep track of multiple requests and their order,
- requests can time out, and new requests need to be issued periodically.

Steps:

1. A client requests a webpage from a server using regular HTTP (see HTTP above).

2. The client receives the requested webpage and executes the JavaScript on the page which requests a file from the server.
3. The server does not immediately respond with the requested information but waits until there's **new** information available.
4. When there's new information available, the server responds with the new information.
5. The client receives the new information and immediately sends another request to the server, re-starting the process.



Q9: Name and explain what different communication techniques on the web do you know? ☆☆☆

Topics: WebSockets

Answer:

- **AJAX** - `request` → `response`. Creates a connection to the server, sends request headers with optional data, gets a response from the server, and closes the connection. *Supported in all major browsers.*

- **Long poll** - `request` → `wait` → `response`. Creates a connection to the server like AJAX does, but maintains a keep-alive connection open for some time (not long though). During connection, the open client can receive data from the server. The client has to reconnect periodically after the connection is closed, due to timeouts or data eof. On server side it is still treated like an HTTP request, same as AJAX, except the answer on request will happen now or some time in the future, defined by the application logic.

- **WebSockets** - `client` ↔ `server`. Create a TCP connection to the server, and keep it open as long as needed. The server or client can easily close the connection. The client goes through an HTTP compatible handshake process. If it succeeds, then the server and client can exchange data in both directions at any time. It is efficient if the application requires frequent data exchange in both ways. WebSockets do have data framing that includes masking for each message sent from client to server, so data is simply encrypted.

- **WebRTC** - `peer` ↔ `peer`. Transport to establish communication between clients and is transport-agnostic, so it can use UDP, TCP or even more abstract layers. This is generally used for high volume data transfer, such as video/audio streaming, where reliability is secondary and a few frames or reduction in quality progression can be sacrificed in favour of response time and, at least, some data transfer. Both sides (peers) can push data to each other independently. While it can be used totally independent from any centralised servers, it still requires some way of exchanging endPoints data, where in most cases developers still use centralised servers to "link" peers. This is required only to exchange essential data for establishing a connection, after which a centralised server is not required.

- **Server-Sent Events** - `client` ← `server`. Client establishes persistent and long-term connection to server. Only the server can send data to a client. If the client wants to send data to the server, it would require the use of another technology/protocol to do so. This protocol is HTTP compatible and simple to implement in most server-side platforms. This is a preferable protocol to be used instead of Long Polling.

Q10: WebSockets vs Rest API for real time data? Which to choose? ☆☆☆

Topics: API Design Software Architecture WebSockets REST & RESTful

Problem:

I need to constantly access a server to get real time data of financial instruments. The price is constantly changing so I need to request new prices every 0.5 seconds. Which kind if API would you recommend?

Solution:

The most efficient operation for what you're describing would be to use a webSocket connection between client and server and have the server send updated price information directly to the client over the webSocket ONLY when the price changes by some meaningful amount or when some minimum amount of time has elapsed and the price has changed.

Here's a comparison of the networking operations involved in sending a price change over an already open webSocket vs. making a REST call.

webSocket

1. Server sees that a price has changed and immediately sends a message to each client.
2. Client receives the message about new price.

Rest/Ajax

1. Client sets up a polling interval
2. Upon next polling interval trigger, client creates socket connection to server
3. Server receives request to open new socket
4. When connection is made with the server, client sends request for new pricing info to server
5. Server receives request for new pricing info and sends reply with new data (if any).
6. Client receives new pricing data
7. Client closes socket
8. Server receives socket close

As you can see there's a lot more going on in the Rest/Ajax call from a networking point of view because a new connection has to be established for every new call whereas the webSocket uses an already open call. In addition, in the webSocket cases, the server just sends the client new data when new data is available - the client doens't have to regularly request it.

A webSocket can also be faster and easier on your networking infrastructure simply because fewer network operations are involved to simply send a packet over an already open webSocket connection versus creating a new connection for each REST/Ajax call, sending new data, then closing the connection. How much of a difference/improvement this makes in your particular application would be something you'd have to measure to really know.

Q11: Explain key features of Socket.io ☆☆☆

Topics: WebSockets

Answer:

Socket.io is a library which enables real-time and full duplex communication between the Client and the Web servers.

- It helps in broadcasting to multiple sockets at a time and handles the connection transparently.
- It works on all platform, server or device ensuring the equality, reliability, and speed.
- It automatically upgrades the requirement to WebSocket if needed.
- It is a custom real-time transport protocol implementation on top of other protocols.
- It requires both libraries to be used Client side as well as a server-side library.
- IO works on work-based events. there are some reserved events which can be accessed using the Socket on server side like Connect, message, Disconnect, Ping and Reconnect.
- There are some Client based reserved events like Connect, connect- error, connect-timeout and Reconnect etc.

Q12: Why would you choose *Server-Sent Events* over *WebSockets*?

☆☆☆☆

Topics: WebSockets

Answer:

One reason SSEs have been kept in the shadow is because later APIs like WebSockets provide a richer protocol to perform bi-directional, full-duplex communication. Having a two-way channel is more attractive for things like games, messaging apps, and for cases where you need near real-time updates in both directions. However, in some scenarios data doesn't need to be sent from the client. You simply need updates from some server action. A few examples would be friends' status updates, stock tickers, news feeds, or other automated data push mechanisms (e.g. updating a client-side Web SQL Database or IndexedDB object store). If you'll need to send data to a server, XMLHttpRequest is always a friend.

SSEs are sent over traditional HTTP. That means they do not require a special protocol or server implementation to get working. WebSockets on the other hand, require full-duplex connections and new Web Socket servers to handle the protocol. In addition, Server-Sent Events have a variety of features that WebSockets lack by design such as automatic reconnection, event IDs, and the ability to send arbitrary events.

Q13: Explain how does WebSockets protocol work under the hood

☆☆☆☆

Topics: WebSockets

Answer:

WebSocket is another protocol for sending and receiving messages like HTTP.

Both HTTP and WebSockets send messages over a TCP (Transmission Control Protocol) connection, which is a transport-layer standard that ensures streams of bytes, sent over in packets, are delivered reliably and predictably from one computer to another. So, HTTP and WebSockets use the same delivery mechanism at the packet/byte level, but the protocols for structuring the messages are different.

In order to establish a WebSocket connection with the server, the client first sends an HTTP “handshake” request with an upgrade header, specifying that the client wishes to establish a WebSocket connection. The request is sent to a **ws:** or **wss::** URI (analogous to http or https). If the server is capable of establishing a WebSocket connection and the connection is allowed (for example, if the request comes from an authenticated or whitelisted client), the server sends a successful handshake response, indicated by HTTP code 101 Switching Protocols.

Once the connection is upgraded, the protocol switches from HTTP to WebSocket, and while packets are still sent over TCP, the communication now conforms to the WebSocket message format. Since TCP, the underlying protocol that transmits data packets, is a full-duplex protocol, both the client and the server can send messages at the same time. Messages can be fragmented, so it's possible to send a huge message without declaring the size beforehand. In that case, WebSockets breaks it up into frames. Each frame contains a small header that indicates the length and type of payload and whether this is the final frame.

Q14: What is *Sec-WebSocket-Key* for? ☆☆☆☆

Topics: WebSockets

Answer:

The `Sec-WebSocket-Key` header field is used in the WebSocket opening handshake. It is sent from the client to the server to provide part of the information used by the server to prove that it received a valid WebSocket opening handshake. This helps ensure that the server does not accept connections from non-WebSocket clients (e.g., HTTP clients) that are being abused to send data to unsuspecting WebSocket servers.

It does not provide any security (secure websockets - `wss://` - does), it just ensures that server understands websockets protocol.

Q15: What is WebSockets *Frame*? ☆☆☆☆

Topics: WebSockets

Answer:

Websocket is not a stream based protocol like TCP, it's message based. The websocket protocol communicates with **frames**. Frames are a header + application data. The frame header contains information about the frame and the application data. The application data is any and all stuff you send in the frame "body".

In its most basic form the websocket protocol has three non-control frames and three control frames. These are

- Non-control:
  - 'text' - denotes we are sending 'utf-8' encoded bytes
  - 'binary' - denotes we are sending raw bytes
  - 'continue' - denotes this message is a continuation fragment of the previous message.
- Control:
  - 'close' - says we either want to close, or are responding to a close
  - 'ping' - pings!
  - 'pong' - pongs!

A frame looks like this:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-------+-+-------------+-------------------------------+
|F|R|R|R| opcode|M| Payload len |    Extended payload length    |
|I|S|S|S|  (4)  |A|     (7)     |             (16/64)           |
|N|V|V|V|       |S|             |   (if payload len==126/127)   |
| |1|2|3|       |K|             |                               |
+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +
|     Extended payload length continued, if payload len == 127  |
+ - - - - - - - - - - - - - - - +-------------------------------+
```

```
|                               |Masking-key, if MASK set to 1  |
+-------------------------------+-------------------------------+
| Masking-key (continued)       |          Payload Data         |
+-------------------------------- - - - - - - - - - - - - - - - +
:                     Payload Data continued ...                :
+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +
|                     Payload Data continued ...                |
+---------------------------------------------------------------+
```

Q16: Would WebSockets be able to handle 1,000,000 concurrent connections? ☆☆☆☆

Topics: WebSockets

Answer:

Yes, but it's expensive.

This question is not unique to WebSockets since WebSockets are fundamentally long-lived TCP sockets with a HTTP-like handshake and minimal framing for messages.

The real question is: could a single server handle 1,000,000 simultaneous socket connections and what server resources would this consume? The answer is complicated by several factors, but 1,000,000 simultaneous active socket connections is possible for a properly sized system (lots of CPU, RAM and fast networking) and with a tuned server system and optimized server software.

Q17: When to use WebRTC over WebSockets? ☆☆☆☆

Topics: WebSockets

Problem:

I'm looking to build a chat app that will allow video, audio, and text. How to to decide which to use, WebRTC or WebSockets?

Solution:

- **WebRTC** is designed for high-performance, high quality communication of video, audio and arbitrary data. In other words, for apps exactly like what you describe.
- WebRTC is mainly UDP. Thus main reason of using WebRTC instead of Websocket is latency. With WebRTC you may achive low-latency and smooth playback which is crucial stuff for VoIP communications.
- With WebRTC the data is end-to-end encrypted and does not pass through a server (except sometimes TURN servers are needed, but they have no access to the body of the messages they forward).
- WebRTC apps need a service via which they can exchange network and media metadata, a process known as signaling.

- **WebSocket** on the other hand is designed for bi-directional communication between client and server. It is possible to stream audio and video over WebSocket (see here for example), but the technology and APIs are not inherently designed for efficient, robust streaming in the way that WebRTC is.
- Websockets use TCP protocol. With websocket streaming you will have either high latency or choppy playback with low latency.
- With Websockets the data has to go via a central webserver which typically sees all the traffic and can access it.

Q18: What are the differences between Socket.io and WebSockets? ☆☆☆☆

Topics: WebSockets

Answer:

- **WebSocket** is the communication Protocol which provides bidirectional communication between the Client and the Server over a TCP connection, WebSocket remains open all the time so they allow the real-time data transfer. When clients trigger the request to the Server it does not close the connection on receiving the response, it rather persists and waits for Client or server to terminate the request.

- **Socket.IO** is a library which enables real-time and full duplex communication between the Client and the Web servers. It uses the WebSocket protocol to provide the interface. Generally, it is divided into two parts, both WebSocket vs Socket.io are event-driven libraries

No.	WebSocket	Socket.io
1	It is the protocol which is established over the TCP connection	It is the library to work with WebSocket
2	It provides full duplex communication on TCP connections.	Provides the event-based communication between browser and server.
3	Proxy and load balancer is not supported in WebSocket.	A connection can be established in the presence of proxies and load balancers.
4.	It doesn’t support broadcasting.	It supports broadcasting.
5.	It doesn’t have a fallback option.	It supports fallback options.

Q19: Can you suggest how to *load balance* Web Sockets? ☆☆☆☆

Topics: WebSockets

Answer:

Put a **L3** load-balancer that distributes IP packets based on source-IP-port hash to your WebSocket server farm. Since the L3 balancer maintains no state (using hashed source-IP-port) it will scale to wire speed on low-end hardware (say 10GbE). Since the distribution is deterministic (using hashed source-IP-port), it will work with TCP (and hence WebSocket).

Also note that you can do **L7** load-balancing on the HTTP path announced during the initial WebSocket handshake. In that case the load balancer has to maintain state (which source IP-port pair is going to which backend node). It will probably scale to millions of connections nevertheless on decent setup.

Q20: How can WebSockets be better than *Long-Polling* in term of performance? ☆☆☆☆☆

Topics: WebSockets

Answer:

By its very nature, long-polling is a bit of a hack. It was invented because there was no better alternative for server-initiated data sent to the client.

Maintaining an open webSocket connection between client and server is a very *inexpensive* thing for the server to do (it's just a TCP socket). An inactive, but open TCP socket takes no server CPU and only a very small amount of memory to keep track of the socket. Properly configured servers can hold hundreds of thousands of open sockets at a time.

On the other hand a client doing long-polling, even one for which there is no new information to be sent to it, will have to regularly re-establish its connection. Each time it re-establishes a new connection, there's a TCP socket teardown and new connection and then an incoming HTTP request to handle.

HTTP connections, while they don't create open files or consume port numbers for a long period, are more expensive in just about every other way:

- Each HTTP connection carries a lot of baggage that isn't used most of the time: cookies, content type, conetent length, user-agent, server id, date, last-modified, etc. Once a WebSockets connection is established, only the data required by the application needs to be sent back and forth.

- Typically, HTTP servers are configured to log the start and completion of every HTTP request taking up disk and CPU time. It will become standard to log the start and completion of WebSockets data, but while the WebSockets connection doing duplex transfer there won't be any additional logging overhead (except by the application/service if it is designed to do so).

- Typically, interactive applications that use AJAX either continuously poll or use some sort of long-poll mechanism. WebSockets is a much cleaner (and lower resource) way of doing a more event'd model where the server and client notify each other when they have something to report over the existing connection.

- Most of the popular web servers in production have a pool of processes (or threads) for handling HTTP requests. As pressure increases the size of the pool will be increased because each process/thread handles one HTTP request at a time. Each additional process/thread uses more memory and creating new processes/threads is quite a bit more expensive than creating new socket connections (which those process/threads still have to do). Most of the popular WebSockets server frameworks are going the event'd route which tends to scale and perform better.

FullStack.Cafe - Kill Your Tech Interview

Q1: How would you secure WebSockets communication on your project? ☆☆☆☆☆

Topics: WebSockets Web Security

Answer:

You should strongly prefer the secure `wss://` protocol over the insecure `ws://` transport. Like HTTPS, WSS (WebSockets over SSL/TLS) is encrypted, thus protecting against man-in-the-middle attacks.

The WebSocket protocol doesn't handle authorization or authentication. Practically, this means that a WebSocket opened from a page behind auth doesn't "automatically" receive any sort of auth; you need to take steps to also secure the WebSocket connection.

Since you cannot customize WebSocket headers from JavaScript, you're limited to the "implicit" auth (i.e. Basic or cookies) that's sent from the browser. Further, it's common to have the server that handles WebSockets be completely separate from the one handling "normal" HTTP requests. This can make shared authorization headers difficult or impossible.

You can use the `Origin` header as an advisory mechanism—one that helps differentiate WebSocket requests from different locations and hosts, but you *shouldn't* rely on it as a source of authentication.

One pattern I would use that seems to solve the WebSocket authentication problem well is a "ticket"-based authentication system. Broadly speaking, it works like this:

- When the client-side code decides to open a WebSocket, it contacts the HTTP server to obtain an authorization "ticket".
- The server generates this ticket. It typically contains some sort of user/account ID, the IP of the client requesting the ticket, a timestamp, and any other sort of internal record keeping you might need.
- The server stores this ticket (i.e. in a database or cache), and also returns it to the client.
- The client opens the WebSocket connection, and sends along this "ticket" as part of an initial handshake.
- The server can then compare this ticket, check source IPs, verify that the ticket hasn't been re-used and hasn't expired, and do any other sort of permission checking. If all goes well, the WebSocket connection is now verified.

Q2: How to use *CHAP Authentication* (Challenge Response Authentication) for webSockets? ☆☆☆☆☆

Topics: WebSockets Web Security

Answer:

In computing, the Challenge-Handshake Authentication Protocol (CHAP) authenticates a user or network host to an authenticating entity. CHAP is a 3 way handshake:

1. Client Connects to the websocket server, server then sends a `challenge` string (random characters of random or set length) to client.

2. Client responds with the *hash* of the `challenge` + *shared secret*
3. Server calculates the challenge it sent and the 'shared secret' it has locally and compares the client's hash to it's own and either authenticates (adds it to approved clients) or drops the client.

You can get a bit further and if the client was dropped we can add him to a blocked list with a timestamp. If the client tries to connect I check if he is in the blocked list and if his timestamp is old enough. Then continue with the CHAP auth for the client again.

Q3: Explain why CDN (in)availability may be a problem for using WebSockets? ☆☆☆☆☆

Topics: WebSockets CDN

Answer:

Today (almost 4 years later), web scaling involves using Content Delivery Network (CDN) front ends, not only for static content (html,css,js) but also your (JSON) application data.

Of course, you won't put all your data on your CDN cache, but in practice, a lot of common content won't change often. I suspect that 80% of your REST resources may be cached... Even a *one minute* (or 30 seconds) CDN expiration timeout may be enough to give your central server a new live, and enhance the application responsiveness a lot, since CDN can be geographically tuned...

To my knowledge, there is no WebSockets support in CDN yet, and I suspect it would never be. WebSockets are statefull, whereas HTTP is stateless, so is much easily cached. In fact, to make WebSockets CDN-friendly, you may need to switch to a stateless RESTful approach... which would not be WebSockets any more.

Q4: What is the *mask* in a WebSocket frame? ☆☆☆☆☆

Topics: WebSockets

Answer:

The reason for the masking is to make websocket traffic look unlike normal HTTP traffic and become completely unpredictable. Otherwise any network infrastructure equipment which is not yet upgraded to understand the Websocket protocol can mistake it for normal http traffic causing various problems.

This is especially a problem for caching proxy servers and leads to possible attack scenarios. Specifically crafted websocket traffic can cause cache poisoning by tricking the proxy servers into mistaking a part of a websocket communication for a request and response for an unrelated URL, cache it like a legitimate response and send it to other users which request that URL.

Basically, WebSockets is unique in that you need to protect the network infrastructure, even if you have hostile code running in the client, full hostile control of the server, and the only piece you can trust is the client browser. By having the browser generate a random mask for each frame, the hostile client code cannot choose the byte patterns that appear on the wire and use that to attack vulnerable network infrastructure.

Q5: What is the fundamental difference between WebSockets and pure TCP? ☆☆☆☆☆

Topics: WebSockets

Problem:

I've read about WebSockets and I wonder why browser couldn't simply open trivial TCP connection and communicate with server like any other desktop application?

Solution:

- It's easier to communicate via TCP sockets when you're working within an intranet boundary, since you likely have control over the machines on that network and can open ports suitable for making the TCP connections.

- Over the internet, you're communicating with someone else's server on the other end. They are extremely unlikely to have any old socket open for connections. Usually they will have only a few standard ones such as port 80 for HTTP or 443 for HTTPS. So, to communicate with the server you are obliged to connect using one of those ports.

- Given that these are standard ports for web servers that generally speak HTTP, you're therefore obliged to conform to the HTTP protocol, otherwise the server won't talk to you. The purpose of web sockets is to allow you to initiate a connection via HTTP, but then negotiate to use the web sockets protocol (assuming the server is capable of doing so) to allow a more "TCP socket"-like communication stream.

- A WebSocket requires some sort of transport protocol to operate over, but that transport layer doesn't have to be TCP (it's almost always going to be TCP in practice though). You could think of WebSockets as a kind of wrapper around TCP, but I don't believe there's any prescriptive link between the two.

- When you send bytes from a buffer with a normal TCP socket, the send function returns the number of bytes of the buffer that were sent. If it is a non-blocking socket or a non-blocking send then the number of bytes sent may be less than the size of the buffer. If it is a blocking socket or blocking send, then the number returned will match the size of the buffer but the call may block. With WebSockets, the data that is passed to the send method is always either sent as a whole "message" or not at all. Also, browser WebSocket implementations do not block on the send call.

- When the receiver does a `recv` (or `read`) on a TCP socket, there is no guarantee that the number of bytes returned corresponds to a single send (or write) on the sender side. It might be the same, it may be less (or zero) and it might even be more (in which case bytes from multiple send/writes are received). With WebSockets, the recipient of a message is event-driven (you generally register a message handler routine), and the data in the event is always the entire message that the other side sent.