Socket.IO

Contents

Before Socket.IO

Pre-SocketIO

- Socket IO is sitting on top of a whole bunch of stuff namely the web socket API.
- Socket IO uses web sockets when it can which is most of the time. But if it can't, it has fail over mechanisms. Whether it's the client or the server, socket IO is doing that behind the scenes.
- In order to fully understand Socket.IO, one needs to know -
    1. TCP/IP
    2. (Network) Socket
    3. Web Sockets

TCP/UDP and Networking 101

- Cloud does not actually exist. The cloud is really just a whole bunch of actual computers somewhere else in the world.
- A cloud computer isn't really a cloud computer. It's really just someone else's computer.
- E.g. AWS, GCP, Azure
- So cloud is actual computers that just don't belong to you as quick for instance.
- It's just a network of computers that are talking to each other. And what's actually getting passed around are called packets, which is little streams of data.
- Node handles pretty much everything underlying related to Network, packets, etc. nd on top of that we can directly use Socket.io.
- Socket IO is all about networking.
- Refer below packet structure.
- As developers, we are going to be interested primarily in Application, Transport and Network layers, specifically application layer is going to be where we're going to be spending most of our time at least usually the Web sockets use TCP, so we're going to be going back and forth between HTTP and TCP a lot.
- HTTP uses TCP as its transport layer.
- When you have a computer with some kind of internet connection, the transport layer creates 2 raised to 16th ports on your computer or roughly 65000 **ports**.
- Whenever you start a node app for instance and maybe you start your node app on port 3000. The reason you have that 3000 is you're using one of the 65000 ports the transport layer creates.
- TCAP and IP together get two computers ready to talk to each other. They create an environment that will allow two machines to talk to each other.

Packet Structure

- Each packet structure has five basic layers. Starting from the bottom, there is the **physical** layer, above that we have the **link** layer, above that we have the **network** layers sometimes called the **Internet** layer, the next layer is called the **transport** layer and above that is the **application** layer.
- These are the layers that make up a packet as data passes back and forth between server and client.
- Physical Layer
    - E.g. cables.
    - Actual physical cables that are connecting stuff together.
    - This is related to Hardware.
- Link layer
    - E.g. WiFi, Ethernet connection.
    - This is related to Hardware.
- Network Layer / Internet Layer
    - E.g. IP protocol
- Transport Layer
    - E.g. TCP, UDP
    - Transport layer plus the network layer together form the **Internet Protocol suite** or TCP/ IP.
- Application Layer
    - The top layer the application layer is the layer that we are going to be working with as developers most of the time.
    - E.g. HTTP, FTP, SSH, SMTP, etc.

UDP (User Datagram Protocol)

- UDP is incredibly light weight. It's only 8 bytes for a header so it's really tiny.
- UDP requires very little overhead.
- It's also **connectionless**. What it means is when you want to send data. So let's say you have a client and a server. And the client wants to talk to the server, you don't have to create a connection first. You can just start talking and even if the other computer/server doesn't want to hear from you that's OK. You don't have to wait for the connection to be established. You can just start going. That can be a big advantage whenever you're ready to start talking, just start sending data.
- There's also a consistency that you can count on. UDP will send data no matter what that might be seen as a very good thing. But it can in fact be a very troubling thing or a frustrating thing. E.g.

    - What happens if there's packet loss. UDP does not care, it will keep on sending packets. It doesn't make any difference.
    - What if the network is very congested, it doesn't care, it will just keep on sending packets and just making the network congested.
    - What happens if the packets are out of order? It doesn't care, it's not UDP's problem. They're going to show up out of order that will be the other side's problem.
- Because of all above reasons, UDP is very **fast** but it's also incredibly **unreliable**.
- It's used primarily for things like maybe video games or real time communication.
    - E.g. Let's say you're playing real time strategy game and you're playing and suddenly your character seems to move back in time five seconds. That's UDP has been back over here on the right has been sending data over just screaming and the data hasn't been getting to the server. And suddenly the server updates your machine with "oh i guess i'm actually way behind. I'm going to start sending some different data."
    - Or if you've been doing a video chat on say FaceTime or Skype or Google chat, where you're talking and it looks like everything's normal and then all of a sudden the video chat catches up and you're seeing live a live feed again.
- UDP will be used for things like gaming or live communication things that you want it to feel like a live experience and you do not ever want to introduce latency, that you want something fast but can be unreliable.

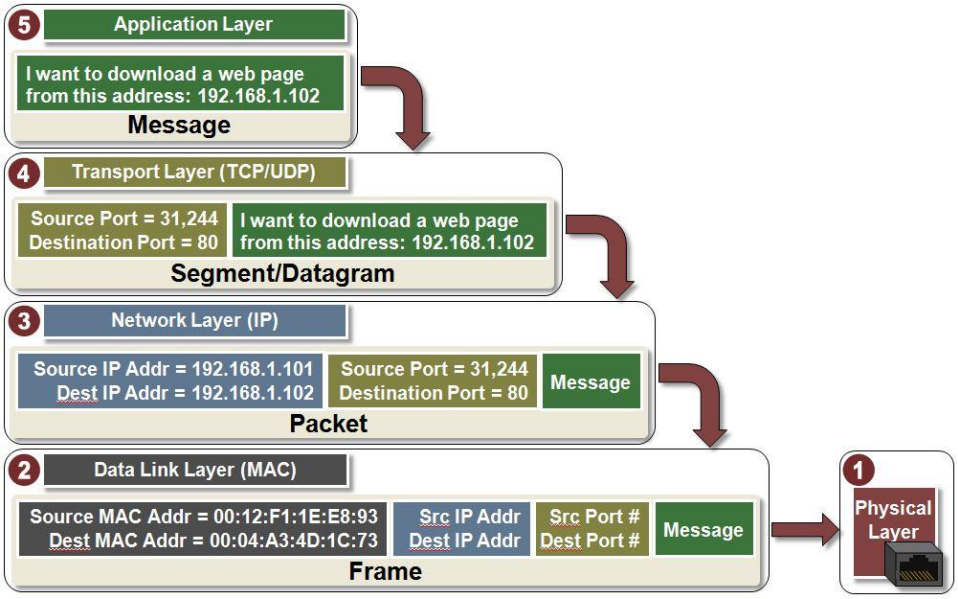
TCP (Transmission Control Protocol)

- TCP is **connection based**. It means unlike UDP if you're a client and you sit down and you want to start talking to a computer via TCP, you don't just start screaming, you actually go through what's called a **3-way handshake** because before you're going to actually transmit any data, you have to initiate a connection.
- It's incredibly **reliable** because you actually know that the connection is going to is going to happen.
- It also have has **delivery acknowledgements** which just means every time data comes through the server over client, it will let the client know that I got your data and vice versa.
- TCAP is very reliable because you can count on the data coming through because you actually get acknowledgements that the data was received.
- There's also **retransmission of data.** With UDP, UDP may not even know that whether it was received or not because there isn't data acknowledgement. But in a TCP connection, if the data isn't received, the server can let the client know I didn't get something and then the client can send it again.

- TCP also guarantees that packets arrive in the **correct order** regardless of what happens with the network.
- One of the thing in TCP that happens is you also have **congestion control** so that if the network is overwhelmed let's say you're at a concert or a sporting event or something where the network is totally overwhelmed where you're trying to connect, TCP may intentionally introduce latency to try and keep packet loss to a minimum so that it doesn't have to make the problem even worse.
- For web application, we need TCP because we need all of the reliability that TCAP has to offer.

Networking 201 – What is Socket and why should we care

- Web sockets is the native technology. That's actually part of JavaScript and web sockets will allow you to connect to a server's socket via TCP rather than HTTP. So Web sockets would be the native browser protocol.
- Socket IO on the other hand does the same thing. It allows you to try and connect to a socket on the other computer. But it comes with a bunch of advantages so that if this doesn't support web sockets, socket IO can fail over to other things specifically long polling.
- The socket is the networking part and we are going to send data through that socket by connecting to it via some means.
- A network socket is an open stream of data that gets below data passing in and out of it.



Client-Server model with Socket.IO

- We're going to have our client and our server. Since our server will be on Node, we need to use Socket.io. (There's actually a python version of socket IO, there's a C-sharp version

etc.) We're going to have a node server most of the time and usually the client is going to be a browser.

- The client will initially make a request to port 80 or 443 secure or insecure it doesn't matter. That will establish an initial HTTP connection.
- Once we've got everything set up the node server will specify a port on your computer. It will be an arbitrary port. It makes no difference. It'll just be a TCP or UDP port that we can use.
- And then we will open up a socket from our computer to that computer that is all TCP and data can freely flow back and forth.
- Unlike HTTP, for every request a new HTTP connection is created and once that request is done, the HTTP connection is disconnected. However with Web sockets and socket IO, TCP connection is going to remain open and you've got all those layers working together to make this magic happen.

HTTP vs Web sockets

- Web sockets is the native JavaScript technology.
- Web sockets is just a precursor for socket Io which is going to use the web socket API.
- When we use web sockets, we're going to have two different players.
- Client – We're going to have a client and the client is going to be just HTML, JavaScript and CSS. This is where we have web socket API. The Web socket API is the JavaScript portion.
- Server – For that, we will have Node. The server has no idea that web socket API exists. **(Server doesn't know about web sockets). It just knows about sockets.** It just knows how to deal with the transmission of network traffic.
- So something has to act as a translator between client and the server because the connection between the two of them is going to use a protocol called WS instead of HTTP.
- The WS protocol needs something at server side to be able to translate between the web socket (client) way of communicating and the (node) server way of communicating. The server way of communicating is going to be either the web socket module or the socket IO module.
- The node module at server side is going to be the interpreter between the (network) socket and the JavaScript web socket.

Short Overview of Native Web Sockets

- Documentation: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

- The WebSocket API is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. Before the web socket API this was not possible but it is now built into JavaScript to be able to communicate at a socket level with a browser.
- With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.
- Web sockets are connected all the time right through the TCP channel rather than through the HTTP channel which disconnects as soon as everything is complete.
- 3rd party package – WS: https://www.npmjs.com/package/ws
    - You would choose socket.io over ws package for more flexibility and more power.
    - If you just want speed this will keep as close to the metal as possible but then you have to implement a lot of stuff yourself which is probably not realistic.
- Refer – https://github.com/sameerbhilare/Socket.IO/tree/main/workspace/01-native-ws

Socket.IO 101

The Basics and Socket.IO vs WS

- Socet.IO official website: https://socket.io/
- Socket.IO enables **real-time, bidirectional and event-based** communication.
- It works on every platform, browser or device, focusing equally on reliability and speed.
- To install Socket.IO
  npm install socket.io --save
- From npm site, we can see that ws (WebSocket) is far more heavily downloaded than socket IO. That's also because when we install socket.io, ws is also installed as socket.io is based on ws only.

Why Socket.io

- Since socket.io uses ws, why on earth would I use socketio, when I could use web sockets (ws)?
- Socket.io takes care of all the problems when using plain WebSocket(ws) packahe APIs.

Problems with using plain WebSocket(ws)

- We need to take care of below things by our own –
- The ability for the user to reconnect
- Binary support so that we could send blob or buffer back and forth
- Multi room support so that a browser could be connected to essentially multiple sockets.
- Deal with blocker issues. Meaning dealing with Antivirus, Firewall, Load balancer, proxy.

Socket.IO

- Socket.io uses engine.io. Underneath, engine.io, there is web sockets as well as some other options.
- Engine.io isn't just using web sockets, it's using a whole bunch of stuff.
- So it's not like you have a choice between using web sockets or socket IO. Socket IO will actually use web sockets whenever it possibly can.
- But if you want the benefit of reconnection, passing binary data, this also detects disconnection, multi-room or what's called multiplexing and then reliability by way of getting through or around blockers, then use socket.io
- May be native WebSockets would get better in future, but at least for now and really since the advent of web sockets, you don't just use the native API. You want something that's going to provide a whole bunch of extra stuff.
- So you still get to use web sockets but you get the benefit of some of the best developers in the world working on socket.io who understand networking way better

than you do and probably understand Node and JavaScript way better than you do, that's handling all the stuff before you actually get to web sockets.

Quick analogy to understand difference between Socket.io and WebSocket

- Electric cars are awesome for lots of reasons but I don't have one because the infrastructure is not there. I need to be able to drive sometimes long distances and if there isn't an electric charging station, I am toast and sometimes that I would have to go way out of the way. It's getting better but it's not there. Instead of an electric car, I drive a hybrid and my hybrid goes on electricity quite always. But the stable infrastructure is there for gasoline. So I always want to use electricity if I can but if I can't I can fall back on the gas engine. That is the difference.
- Web sockets are like electric cars that are awesome. But the infrastructure just isn't there yet for them to always be the right choice. Maybe someday it will be maybe technology will get to the point where the native thing does everything that we wanted to do with none of the drawbacks. But it's not right now. Socket IO is like the hybrid where you want to use electricity if you can. But if you can't you've got a gasoline engine to fall back on because gasoline is sold everywhere.

Use cases for Socket.io

- Real-time analytics
    - Push data to clients that gets represented as real-time counters, charts or logs.
- Instant messaging and chat
    - Socket.IO's "Hello world" is a chat app in just a few lines of code.
- Binary streaming
    - Starting in 1.0, it's possible to send any blob back and forth: image, audio, video.
- Document collaboration
    - Allow users to concurrently edit a document and see each other's changes.

Main Features of Socket.io

- https://socket.io/docs/v3#Features

What Socket.io is NOT

- Socket.IO is NOT a WebSocket implementation. Although Socket.IO indeed uses WebSocket as a transport when possible, it adds additional metadata to each packet. That is why a WebSocket client will not be able to successfully connect to a Socket.IO server, and a Socket.IO client will not be able to connect to a plain WebSocket server either.

In a nutshell

- Socket IO is a node server as well as a client for either the browser or a node server, which connects to network sockets.
- It will try and use web sockets but it comes with all other things reliability, auto reconnection, disconnect detection, binary support, multiplexing and room support and all of those things packed together along with the power of just socket connection is socket IO.
- Vanilla Web socket by itself does not have any of the stuff, it just uses web sockets which if that's all you're going for, you should be using web sockets. But if you are going for a solid, real-time, socket based connection between javascript and node, then Socket IO is what you want.
- Refer: https://github.com/sameerbhilare/Socket.IO/tree/main/workspace/02-socket.io-101

Small Chat App

Server API

- Server API Docs: https://socket.io/docs/v3/server-api/
- Refer Server API docs to see different ways of creating a socket server, etc.
- Server APIs include APIs for different classes like Server, Namespace, Socket and Client.
    - Object of Server class represents socketio server.
    - Namespace represents a pool of sockets connected under a given scope identified by a pathname (eg: /chat), like different chat rooms.
    - A Socket is the fundamental class for interacting with browser clients. A Socket belongs to a certain Namespace (by default /) and uses an underlying Client to communicate.
    - A Client class represents an incoming transport (engine.io) connection. A Client can be associated with many multiplexed Sockets that belong to different Namespaces.

Client API

- Client API Docs: https://socket.io/docs/v3/client-api/

Simple Chat Application

- Refer – https://github.com/sameerbhilare/Socket.IO/tree/main/workspace/02-socket.io-101

Socket.IO 201

Namespaces

- Namespaces and Rooms are two different concepts, however for the most part functionally on smaller apps, they will be the same.
- A Namespace is a communication channel that allows you to split the logic of your application over a single shared connection.
- Namespaces are a way to group a bunch of sockets together.
- Socket.io allows us to "namespace" our sockets, which essentially means assigning different endpoints or paths.
- E.g.
  `const socket = io('http://localhost:3000');`
  This points to the '/' namespace or endpoint.
  `const socket = io('http://localhost:3000/admin');`
  This points to the '/admin' namespace or endpoint.
- It's not really an endpoint, it's just internal for socket IO. Remember Web sockets does not support this. So this is a socket IO sort of feature or benefit. This is just for internal management.
- Client requests to join different namespaces, so will have different sockets. And once you're in one namespace, there isn't any cross socket communication meaning the socket can't hop from place to place. So everything that belongs to one socket is just going to belong to that socket. It won't be able to listen to stuff or respond to stuff on other sockets (belonging to other namespaces/endpoints).
- The server can send messages to whatever namespace it wants to.
- Namespace object represents a pool of sockets connected under a given scope identified by a pathname (eg: /chat). A client always connects to / (the main namespace), then potentially connect to other namespaces (while using the same underlying connection – multiplexing).
- From client, even though we can connect to different namespaces, but the underlying connection is the same. So if you print the socket.id for each socket which is connected to different namespaces, you will see different ids but all ends with a common string. So this is multiplexing at work. This is socket IO doing its thing where we have connected to multiple name spaces and we have ultimately different IDs but it's still the same underlying connection.
- E.g. For single client,
  socket.id for '/' namespace=> blyG7L_0LEoXuW8zAAAB
  socket.id for '/admin' namespace => /admin#blyG7L_0LEoXuW8zAAAB

Summary

- Namespaces are just the bucket inside of the socket IO server.
- They're like the workspaces in slack or they're like overall accounts that you have with your one email address and there is no association with them other than you have one main service that's running everything. Socket IO is running everything and socket IO can connect to all the name spaces.
- The client on the other hand can only listen for events on one socket at a time.
- So you can join whatever namespace you want main or admin and whichever one you join or if you join multiple, you'll listen for an event on a particular socket or namespace, a particular connection. And if the server doesn't want to send you anything well there's nothing you can do about it, that's part of being the client. ☺
- The name spaces are very useful and very powerful when you have a clear obvious separation of concerns.
- The main namespace ('/') is only jobs going to be to manage the other namespace because that's where the actual chat rooms are at.
- The main namespace's ('/') job is just to facilitate joining of the other name spaces. And those other namespace is where the chat actually happens.

Rooms

- Within each namespace, we can define arbitrary channels that sockets can join and leave. These are called Rooms.
- If we compare slack to socket.io, then workspace in slack is analogous to namespace in socket.io. And channels in slack is analogous to rooms in socket.io.
- So we have a socket IO server. The socket server is the really big bucket. Inside of the socket IO server, we will have namespaces and inside of the namespaces we have rooms.
- Server Socket: https://socket.io/docs/v3/server-api/#Socket
- Client Socket: https://socket.io/docs/v3/client-api/#Socket
- The client has no idea that it's in a room. The client knows it's connected to a namespace. But it has no idea that it's even in a room.

Namespace Cheatsheet (Server Only)

ALL these are server only

Socket (Server) Class

(Server to one Socket): Send an event from the server to this socket only:

```
1. socket.emit()
2. socket.send() // same as above line, but uses default event name as ‘message’
```

(Socket to Room): Send an event from a socket to a room:
NOTE: remember, this event will not go to the sending socket

```
1. socket.to(roomName).emit()
2. socket.in(roomName).emit() // same as above line
```

(Socket to Socket): Because **each socket has it's own room, named by it's socket.id**, a socket can send a message to another socket: (like private messaging)

```
1. socket.to(anotherSocketId).emit('hey');
2. socket.in(anotherSocketId).emit('hey'); // same as above line
```

Namespace

A namespace can send a message to any room: (Namespace to Room)
This is same as above `socket.to(roomName).emit()` but only difference is that, here everyone will get the message. And with `socket.to(roomName).emit()`, everyone except the client socket will get the message.

```
1. io.of(aNamespace).to(roomName).emit()
2. io.of(aNamespace).in(roomName).emit() // same as above line
```

A namespace can send a message to the entire namespace. (Namespace to entire Namespace)

```
1. io.emit()
2. io.of('/').emit()  // same as above line
3. io.of('/admin').emit()
```

Multiplayer Video Game (e.g. agar.io)

- Real Time gaming requires heavy processing.
- Modern videogames need to operate between 30 frames per second and 60 frames per second. That means for 30 fps, we need to send out 1 frame every 0.033 seconds. And if we do 60 fps, we need to send out 1 frame every 0.016 seconds.
- And a frame is going to require updating the client with what's happened everywhere else and it's going to be updating the server with what has happened on this client. So with 30 fps setting, we need to send (33 x 2 x n) messages where 2 because it has to go both ways, that is it has to come from the client to the server and from the server to the client, and 'n' means number of clients.
- For instance, with 4 clients and 30 fps setting, we will have to send 33 x 2 x 4 = 264 messages per second. We're going to have 264 messages per second set back and forth across TCP. Now your internet connection is probably fast and your computer is fast but we have a big job to make sure that those are absolutely as optimal as possible. This is a very different concern than we had in the slack/chat client.
- Gamers are very smart. If you give them a chance, they will exploit your game. We can't give the players access to anything.
- What things need to happen every single frame So 33 times per second what do we actually need to send out?
    - Send out where the orbs are. That information is going to need to be sent out because every player is going to need to draw them every single frame 33 times per second because that's how that's how games work too.
    - We need to know where all of the players are because they unlike the orbs can actually move. So we're going to need to communicate that every single frame because the players are constantly moving and they're moving again at 30 frames per second.
    - We're going to need to know if there were any collisions. Player to player, Player to orb, that is, did somebody run into somebody else or did this player consume this player or did this player consume this orb. We're going to have to check for that every single frame.
    - We're going to need to keep track of which direction a given player wants to go. So that's going to be just a mouse event that's going to happen on the client. But that's information we're going to need to figure out where to draw the player.

Things to keep in mind while designing Multi-player games

- The client is going to have to actually draw all of the players and all of the orbs in their correct spots 30 or 60 frames per second or something in between. And the client is going to have to listen for all mouse movement so that it can change the direction if need be. But we can't have the client deciding where to draw players or checking collisions on the other system. We have to control them in the server because if you had a player with even a little bit of Javascript savvy because start looking through the source code quickly find out where things are happening and all of a sudden everybody quits the game because it's totally unfair.
- The question is in our node's server, to all of our clients what needs to happen from the server and what needs to happen from the client back to the server what we can put, what's the least amount we can put in to reduce bandwidth because we're going to have 264 messages passing back and forth every second. But we cannot trust the players so we can't let the players decide where they're at and we can't let the players decide if they collided the server has to do that.
- So the client's job at the absolute minimum is going to be to draw everything on the screen based on where the server tells it the stuff is at and the server's job is going to be to figure out where all of these things are as infrequently as possible so we can send as little data as possible because we we have to send those messages for every single frame but we want to send as little as possible in there.

Tips and Tricks

- The difference between a good developer and a great developer is that a great developer knows how something works not just how to do something.
- Every socket joins its own room immediately upon a connection to a namespace.
- Handshake happens only one time and that is at the time of joining the main (/) namespace. And the same 'handshake' object is accessible from different namespaces.
- To initialize npm project with default settings for package.json, use –y flag as shown below. It will look at your code and create package.json including name, dependencies, scripts, main, etc.
  npm init –y
-