Live Interaction: Websockets & Socket.io

CIS 1962 (Spring 2026)
April 9nd, 2026

Penn Engineering

Lesson Plan

4	Communication & WebSockets
10	Socket.io
29	Socket.io Best Practices

Logistics

Project Proposals approved!

Happy coding!

(Socket.io will still count as extra credit despite being covered this lecture)

Real-Time Communication
& WebSockets
Why is there a need for real-time communication on the web?

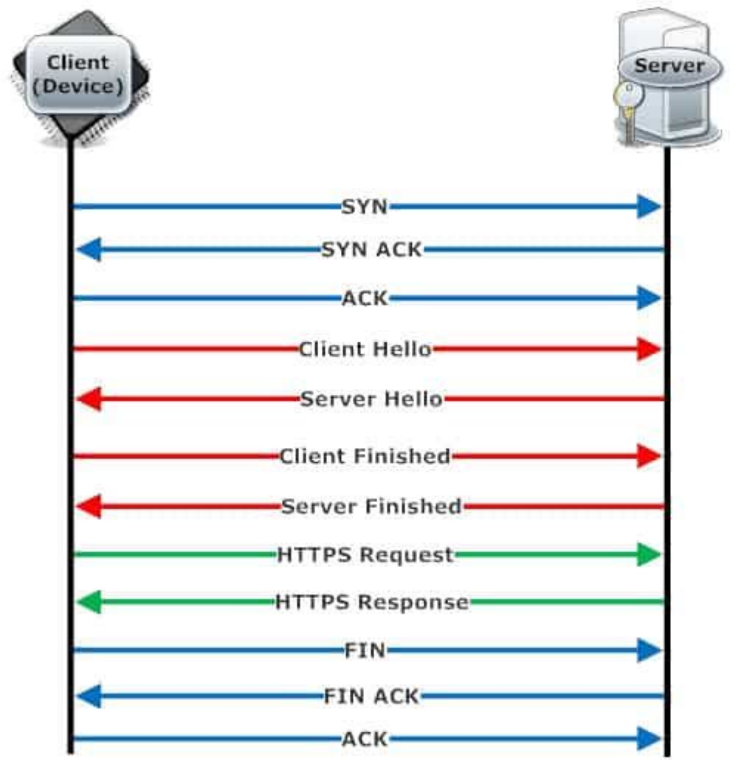
HTTP Communication

So far, with HTTP, we've communicated between server/client as such:

- Client/Frontend sends an HTTP(S) request
- Server/Backend sends an HTTP(S) reponse for Client to recieve
- Connection closes

This is called a **stateless request-response model**.

HTTP Behind the Scenes



Link

Apps with Real-Time Communication

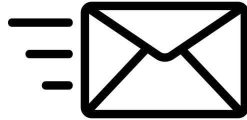
Imagine building:

- A Chat app
- A Stock price dashboard
- A Multiplayer game

These types of apps require **persistent, efficient, two-way** connections.

WebSockets

A WebSocket is a full-duplex (two-way), stateful communication between client and server.



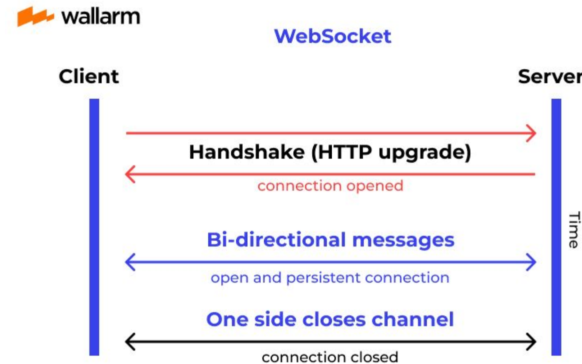
HTTP

WebSocket

WebSockets still make use of TCP (for delivery) and SSL (for security) protocols

How WebSockets Work

1. Client sends a WebSocket handshake to establish the connection
2. Server upgrades connection to WebSocket
3. The connection is now open and both sides can send data
4. Heartbeat messages can be sent to check if connection is alive
5. Either side can send a close message to end the connection



Link

Socket.io

How do we use Socket.io for WebSockets?

Socket.io

Server: `npm install socket.io`

Client: `npm install socket.io-client`

Docs

Socket.io is a JS library for real-time communication, not just WebSockets.

It has fallbacks for when WebSockets aren't supported!

Socket.io Demo

Follow along with the coming slides in the code in this demo:

https://edstem.org/us/courses/91614/lessons/163811/slides/963385

Socket.io: Server Connection

Server:

```
import { createServer } from 'http';
import { Server } from 'socket.io';

const server = createServer(app);
const PORT = 3000;
const CLIENT_PORT = 5173;

const io = new Server(server, {
  cors: {
    origin: `http://localhost:${CLIENT_PORT}`,
  },
});

io.on('connection', (socket) => {
  console.log('user connections');
});
```

Socket.io: Client Connection

Client:

```
import { io } from 'socket.io-client';

const socket = io('http://localhost:3000');

socket.on('connect', () => {
  console.log('connected from socket');
});

socket.on('disconnect', () => {
  console.log('disconnected from socket');
});
```

Socket.io: Events

Socket.io revolves around named events.

The client and server can emit and listen to each other's events to pass payloads between each other.

Server | Client

```
socket.emit("my-event-a");
```

```
socket.on("my-event-a", () => {
  // ...
});
```

```
socket.on("my-event-b", () => {
  // ...
});
```

```
socket.emit("my-event-b");
```

Socket.io: Events

Server listens for "chat_message" which is emitted by client when the button is pressed

```
socket.on("chat_message", (msg) => {
    console.log("Message received:", msg);

    io.emit("chat_message", msg);
});
```

Client listens for "chat_message" which is emitted by server when "chat_message" is detected

```
socket.on("chat_message", (msg) => {
    setMessages((prev) => [...prev, msg]);
});
...
socket.emit("chat_message", msg);
```

Socket.io: One-time Events

`socket.once()` listens for an event only once.

The handler is removed after firing once.

```
socket.once("welcome", (msg) => {
    console.log("Server says:", msg);
});
```

Socket.io: Remove Event Listener

`socket.off()` removes a specific callback or all listeners for an event.

```
useEffect(() => {
    socket.on("chat_message", (msg) => {
        setMessages((prev) => [...prev, msg]);
    });

    return () => {
        socket.off("chat_message");
    };
}, []);
```

Socket.io: Catch-all Listener

`socket.onAny()` fires on every event.

This is useful for logging or debugging.

```
io.on("connection", (socket) => {
  console.log("User connected:", socket.id);

  socket.onAny((event, ...args) => {
    console.log(`Event received: ${event}`,
args);
  });
});
```

Socket.io: Scoping

When working with socket.io, it's important to understand at which scope you are emitting events:

- Is the event being emitting by the server ("io", start with everyone) or a socket ("socket", start with a single socket)
- Modify the destination:
  - `to`: filter to a certain group (single socket or room)
  - `except`: exclude a certain group
  - `broadcast`: exclude the sender

Socket.io: Targeting Sockets

You can emit to a specific socket using the `io.to` method.

```
io.on('connection', (socket) => {
  socket.emit('alan', `HELLO SOCKET ${socket.id}`);
  console.log(socket.id)
  io.to(socketId).emit('turing', 'HELLO SPECIFIC SOCKET');
});
```

Socket.io: Joining/Targeting Rooms

Rooms are ways to group sockets into subsets of connected users. This allows you to target and emit to certain people.

You can join a room using `socket.join()`. Then, you can emit to that with with `io.to`.

```
io.on("connection", (socket) => {
  console.log("User connected:", socket.id);

  socket.join("room1");

  io.to("room1").emit("chat_message", "Hello
room1!");
});
```

Socket.io: Multiple Rooms

`io.except` allows you to target every room except the specified room.

Alternatively, you can also chain `io.to` calls to target multiple rooms.

```
io.on('connection', (socket) => {
  const randomInt = Math.floor(Math.random() * 3);
  if (randomInt == 0) {
    socket.join('LivingRoom');
  } else if (randomInt == 1) {
    socket.join('BathRoom');
  } else {
    socket.join('BedRoom');
  }

  io.except('BedRoom').emit('announcement_1', 'Hello
everyone, except the sleepyheads!');

  io
    .to("LivingRoom")
    .to("BathRoom")
    .emit("announcement_2", "Hello again, please wake
up bedroom people!!");
});
```

Socket.io: Exclude Self in Room

socket.to(room) (as opposed to io.to) will emit to everyone except oneself in a room.

```
io.on('connection', (socket) => {
  const randomInt = Math.floor(Math.random() * 3);
  console.log(randomInt)
  if (randomInt == 0) {
    socket.join('LivingRoom');
  } else if (randomInt == 1) {
    socket.join('BathRoom');
    socket
        .to("BathRoom")
        .emit("occupied", "This bathroom stall is
occupied!")
  } else {
    socket.join('BedRoom');
  }
});
```

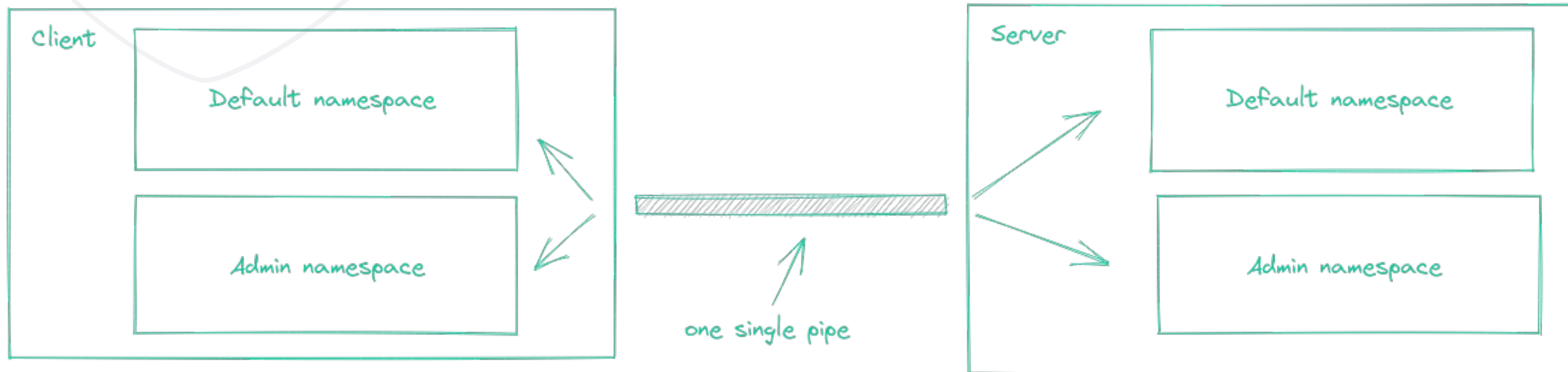
Socket.io: Broadcast

socket.broadcast will send to every socket except socket that sent the event.

```
socket.on("chat_message", (msg) => {
  socket.broadcast.emit("chat_message", msg);
});
```

Socket.io: Namespaces

Namespaces are separate channels over the same connection.
Each namespace has their own events, connections, and logic.
For instance, you can use this to split admin and user logic.



Socket.io: Namespaces

Frontend accesses “localhost:3000/admin”

```
const admin = io.of("/admin");

admin.use((socket, next) => {
  const isAdmin = socket.handshake.auth?.admin === true;

  if (!isAdmin) {
    return next(new Error("Unauthorized"));
  }

  next();
});

admin.on("connection", (socket) => {
  console.log("Admin connected:", socket.id);
});
```

Class Activity

https://edstem.org/us/courses/91614/lessons/163811/slides/963389

Try the live version:

https://cis1962sp26-socketchat-client.vercel.app/

What should you keep in mind when you design an app with Socket.io, especially with a deployed application?

Socket.io Security

- Use robust authentication (e.g JWT) before allowing socket events, so that you can reduce attacks like session hijacking
- Use HTTPS, enable CORS to restrict allowed origins
- Validate/Sanitize your data!
- Use rate limiting strategies, especially with constant real time communication! You don't want to be DDOS'd!

Article

Organize your logic

- Use namespaces to separate functionality
  - Users (/) vs admins (/admin)
  - Chat (/chat) vs notifications (/notifications)
- Use rooms within namespaces to group subsets of clients
  - Chat channels
  - Different game rooms

Organizing your sockets well can make it easier to scale apps with more users/functionality

Performance/Scaling

With your final projects, Socket.io may perform well at a prototype level, but if you want to scale up, it will require some planning.

- You may need different Socket.io adapters to allow for multiple servers
  - The Redis adapter is a popular solution
- Socket.io, by default, maxes out at 10,000-30,000 concurrent connections per instance, beyond that performance degrades

Article

Penn Engineering

Errors, Reconnection, Feedback

- Provide user feedback on connection status, in case connections are intermittent
- Handle the `connect_error` event to help log issues to find problems with connections
- Detect the `disconnect` event and inspect the messages and reasons for disconnection

Article

Testing and Monitoring

- Test using testing frameworks like Mocha and Jest
  - Establish test sockets on the client that emit and listens to expected events
  - Simulate connection, disconnection, reconnection, and message handling throughout your app
- Monitor your app using tools like monitor.io
- Monitor user activity in your app in general to see suspicious patterns. This can help detect vulnerabilities!

Internet of Things

Socket.io can help with **Internet of Things** (IoT) as well, since IoT devices need persistent, bi-directional communication.

This can include items like:

- Smart home systems like Lights, thermostats, security cameras
- Healthcare devices
- Industrial IoT/Factories

Example: IoT Real-Time Sensor Dashboard