



Trystero

Build instant multiplayer web apps, no server required

□ TRY THE DEMO (<https://oxism.com/trystero>) □

Trystero manages a clandestine courier network that lets your application's users talk directly with one another, encrypted and without a server middleman.

The net is full of open, decentralized communication channels: torrent trackers, IoT device brokers, boutique file protocols, and niche social networks.

Trystero piggybacks on these networks to automatically establish secure, private, P2P connections between your app's users with no effort on your part.

Peers can connect via □ BitTorrent, □ Nostr, □ MQTT, ↵ Supabase, □ Firebase, or □ IPFS – all using the same API.

Besides making peer matching automatic, Trystero offers some nice abstractions on top of WebRTC:

- □□ Rooms / broadcasting
- □□ Automatic serialization / deserialization of data
- □□ Attach metadata to binary data and media streams
- ↵□ Automatic chunking and throttling of large data
- □□ Progress events and promises for data transfers
- □□ Session data encryption
- □↖ Runs server-side
- ⚡□ React hooks

You can see what people are building with Trystero [here \(https://github.com/jeremyckahn/awesome-trystero\)](https://github.com/jeremyckahn/awesome-trystero).

Contents

- [How it works](#)
- [Get started](#)
- [Listen for events](#)
- [Broadcast events](#)
- [Audio and video](#)
- [Advanced](#)
 - [Binary metadata](#)
 - [Action promises](#)
 - [Progress updates](#)
 - [Encryption](#)
 - [React hooks](#)

- [Connection issues](#)
 - [Running server-side \(Node, Deno, Bun\)](#)
 - [Supabase setup](#)
 - [Firebase setup](#)
 - [API](#)
 - `joinRoom(config, roomId, [onJoinError])`
 - `selfId`
 - `getRelaySockets()`
 - `getOccupants(config, roomId)`
 - [Strategy comparison](#)
 - [How to choose](#)
-

How it works

□ If you just want to try out Trystero, you can skip this explainer and [jump into using it](#).

To establish a direct peer-to-peer connection with WebRTC, a signalling channel is needed to exchange peer information ([SDP](#) (https://en.wikipedia.org/wiki/Session_Description_Protocol)). Typically this involves running your own matchmaking server but Trystero abstracts this away for you and offers multiple "serverless" strategies for connecting peers (currently BitTorrent, Nostr, MQTT, Supabase, Firebase, and IPFS).

The important point to remember is this:



Beyond peer discovery, your app's data never touches the strategy medium and is sent directly peer-to-peer and end-to-end encrypted between users.



You can [compare strategies here](#).

Get started

You can install with npm (`npm i trystero`) and import like so:

```
import {joinRoom} from 'trystero'
```

Or maybe you prefer a simple script tag? You can just import Trystero from a CDN or download and locally host a JS bundle from the [latest release](#) (<https://github.com/dmotz/trystero/releases/latest>):

```
<script type="module">
  import {joinRoom} from 'https://esm.run/trystero'
</script>
```

By default, the Nostr strategy is used. To use a different one, just use a deep import like this:

```
import {joinRoom} from 'trystero/mqtt' // (trystero-mqtt.min.js with a local file)
// or
import {joinRoom} from 'trystero/torrent' // (trystero-torrent.min.js)
// or
import {joinRoom} from 'trystero/supabase' // (trystero-supabase.min.js)
// or
import {joinRoom} from 'trystero.firebaseio' // (trystero-firebase.min.js)
// or
import {joinRoom} from 'trystero/ipfs' // (trystero-ipfs.min.js)
```

Next, join the user to a room with an ID:

```
const config = {appId: 'san_narciso_3d'}
const room = joinRoom(config, 'yoyodyne')
```

The first argument is a configuration object that requires an `appId`. This should be a completely unique identifier for your app¹. The second argument is the room ID.

Why rooms? Browsers can only handle a limited amount of WebRTC connections at a time so it's recommended to design your app such that users are divided into groups (or rooms, or namespaces, or channels... whatever you'd like to call them).

¹ When using Firebase, `appId` should be your `databaseURL` and when using Supabase, it should be your project URL.

Listen for events

Listen for peers joining the room:

```
room.onPeerJoin(peerId => console.log(`${peerId} joined`))
```

Listen for peers leaving the room:

```
room.onPeerLeave(peerId => console.log(`${peerId} left`))
```

Listen for peers sending their audio/video streams:

```
room.onPeerStream(  
  (stream, peerId) => (peerElements[peerId].video.srcObject = stream)  
)
```

To unsubscribe from events, leave the room:

```
room.leave()
```

You can access the local user's peer ID by importing `selfId` like so:

```
import {selfId} from 'trystero'  
  
console.log(`my peer ID is ${selfId}`)
```

Broadcast events

Send peers your video stream:

```
const stream = await navigator.mediaDevices.getUserMedia({  
  audio: true,  
  video: true  
)  
room.addStream(stream)
```

Send and subscribe to custom P2P actions:

```

const [sendDrink, getDrink] = room.makeAction('drink')

// buy drink for a friend
sendDrink({drink: 'negroni', withIce: true}, friendId)

// buy round for the house (second argument omitted)
sendDrink({drink: 'mezcal', withIce: false})

// listen for drinks sent to you
getDrink((data, peerId) =>
  console.log(
    `got a ${data.drink} with${data.withIce ? '' : 'out'} ice from ${peerId}`
  )
)

```

You can also use actions to send binary data, like images:

```

const [sendPic, getPic] = room.makeAction('pic')

// blobs are automatically handled, as are any form of TypedArray
canvas.toBlob(blob => sendPic(blob))

// binary data is received as raw ArrayBuffer so your handling code should
// interpret it in a way that makes sense
getPic(
  (data, peerId) => (imgs[peerId].src = URL.createObjectURL(new Blob([data])))
)

```

Let's say we want users to be able to name themselves:

```

const idsToNames = {}

const [sendName, getName] = room.makeAction('name')

// tell other peers currently in the room our name
sendName('Oedipa')

// tell newcomers
room.onPeerJoin(peerId => sendName('Oedipa', peerId))

// listen for peers naming themselves
getName((name, peerId) => (idsToNames[peerId] = name))

room.onPeerLeave(peerId =>
  console.log(`#${idsToNames[peerId]} || 'a weird stranger' left`)
)

```

Actions are smart and handle serialization and chunking for you behind the scenes. This means you can send very large files and whatever data you send will be received on the other side as the same type (a number as a number, a string as a string, an object as an object, binary as binary, etc.).

Audio and video

Here's a simple example of how you could create an audio chatroom:

```
// this object can store audio instances for later
const peerAudios = {}

// get a local audio stream from the microphone
const selfStream = await navigator.mediaDevices.getUserMedia({
  audio: true,
  video: false
})

// send stream to peers currently in the room
room.addStream(selfStream)

// send stream to peers who join later
room.onPeerJoin(peerId => room.addStream(selfStream, peerId))

// handle streams from other peers
room.onPeerStream((stream, peerId) => {
  // create an audio instance and set the incoming stream
  const audio = new Audio()
  audio.srcObject = stream
  audio.autoplay = true

  // add the audio to peerAudios object if you want to address it for something
  // later (volume, etc.)
  peerAudios[peerId] = audio
})
```

Doing the same with video is similar, just be sure to add incoming streams to video elements in the DOM:

```

const peerVideos = {}

const videoContainer = document.getElementById('videos')

room.onPeerStream((stream, peerId) => {
  let video = peerVideos[peerId]

  // if this peer hasn't sent a stream before, create a video element
  if (!video) {
    video = document.createElement('video')
    video.autoplay = true

    // add video element to the DOM
    videoContainer.appendChild(video)
  }

  video.srcObject = stream
  peerVideos[peerId] = video
})

```

Advanced

Binary metadata

Let's say your app supports sending various types of files and you want to annotate the raw bytes being sent with metadata about how they should be interpreted. Instead of manually adding metadata bytes to the buffer you can simply pass a metadata argument in the sender action for your binary payload:

```

const [sendFile, getFile] = room.makeAction('file')

getFile((data, peerId, metadata) =>
  console.log(
    `got a file ${metadata.name} from ${peerId} with type ${metadata.type}`,
    data
  )
)

// to send metadata, pass a third argument
// to broadcast to the whole room, set the second peer ID argument to null
sendFile(buffer, null, {name: 'The Courier\'s Tragedy', type: 'application/pdf'})

```

Action promises

Action sender functions return a promise that resolves when they're done sending. You can optionally use this to indicate to the user when a large transfer is done.

```
await sendFile(amplePayload)
console.log('done sending to all peers')
```

Progress updates

Action sender functions also take an optional callback function that will be continuously called as the transmission progresses. This can be used for showing a progress bar to the sender for large transfers. The callback is called with a percentage value between 0 and 1 and the receiving peer's ID:

```
sendFile(
  payload,
  // notice the peer target argument for any action sender can be a single peer
  // ID, an array of IDs, or null (meaning send to all peers in the room)
  [peerIdA, peerIdB, peerIdC],
  // metadata, which can also be null if you're only interested in the
  // progress handler
  {filename: 'paranoids.flac'},
  // assuming each peer has a loading bar added to the DOM, its value is
  // updated here
  (percent, peerId) => (loadingBars[peerId].value = percent)
)
```

Similarly you can listen for progress events as a receiver like this:

```
const [sendFile, getFile, onFileProgress] = room.makeAction('file')

onFileProgress((percent, peerId, metadata) =>
  console.log(
    `${percent * 100}% done receiving ${metadata.filename} from ${peerId}`
  )
)
```

Notice that any metadata is sent with progress events so you can show the receiving user that there is a transfer in progress with perhaps the name of the incoming file.

Since a peer can send multiple transmissions in parallel, you can also use metadata to differentiate between them, e.g. by sending a unique ID.

Encryption

Once peers are connected to each other all of their communications are end-to-end encrypted. During the initial connection / discovery process, peers' SDPs (https://en.wikipedia.org/wiki/Session_Description_Protocol) are sent via the chosen peering strategy medium. By default the SDP is encrypted using a key derived from your app ID and room ID to prevent plaintext session data from appearing in logs. This is fine for most use cases, however a relay strategy operator can reverse engineer the key using the room and app IDs. A more secure option is to pass a `password` parameter in the app configuration object which will be used to derive the encryption key:

```
joinRoom({appId: 'kinneret', password: 'MuchoMaa$'}, 'w_a_s_t_e_v_i_p')
```

This is a shared secret that must be known ahead of time and the password must match for all peers in the room for them to be able to connect. An example use case might be a private chat room where users learn the password via external means.

React hooks

Trystero functions are idempotent so they already work out of the box as React hooks.

Here's a simple example component where each peer syncs their favorite color to everyone else:

```
import {joinRoom} from 'trystero'
import {useState} from 'react'

const trysteroConfig = {appId: 'thurn-und-taxis'}

export default function App({roomId}) {
  const room = joinRoom(trysteroConfig, roomId)
  const [sendColor, getColor] = room.makeAction('color')
  const [myColor, setMyColor] = useState('#c0ffee')
  const [peerColors, setPeerColors] = useState({})

  // whenever new peers join the room, send my color to them:
  room.onPeerJoin(peer => sendColor(myColor, peer))

  // listen for peers sending their colors and update the state accordingly:
  getColor((color, peer) =>
    setPeerColors(peerColors => ({...peerColors, [peer]: color}))
  )

  const updateColor = e => {
    const {value} = e.target

    // when updating my own color, broadcast it to all peers:
    sendColor(value)
    setMyColor(value)
  }
}

return (
  <>
  <h1>Trystero + React</h1>

  <h2>My color:</h2>
  <input type="color" value={myColor} onChange={updateColor} />

  <h2>Peer colors:</h2>
  <ul>
    {Object.entries(peerColors).map(([peerId, color]) => (
      <li key={peerId} style={{backgroundColor: color}}>
        {peerId}: {color}
      </li>
    )))
  </ul>
</>
```

```
)  
}
```

Astute readers may notice the above example is simple and doesn't consider if we want to change the component's room ID or unmount it. For those scenarios you can use this simple `useRoom()` hook that unsubscribes from room events accordingly:

```
import {joinRoom} from 'trystero'  
import {useEffect, useRef} from 'react'  
  
export const useRoom = (roomConfig, roomId) => {  
  const roomRef = useRef(joinRoom(roomConfig, roomId))  
  const lastRoomIdRef = useRef(roomId)  
  
  useEffect(() => {  
    if (roomId !== lastRoomIdRef.current) {  
      roomRef.current.leave()  
      roomRef.current = joinRoom(roomConfig, roomId)  
      lastRoomIdRef.current = roomId  
    }  
  
    return () => roomRef.current.leave()  
  }, [roomConfig, roomId])  
  
  return roomRef.current  
}
```

Connection issues

WebRTC is powerful but some networks simply don't allow direct P2P connections using it. If you find that certain user pairings aren't working in Trystero, you're likely encountering an issue at the network provider level. To solve this you can configure a TURN server which will act as a proxy layer for peers that aren't able to connect directly to one another.

1. If you can, confirm that the issue is specific to particular network conditions (e.g. user with ISP X cannot connect to a user with ISP Y). If other user pairings are working (like those between two browsers on the same machine), this likely confirms that Trystero is working correctly.
2. Sign up for a TURN service or host your own. There are various hosted TURN services you can find online like [Cloudflare](https://developers.cloudflare.com/calls/turn/) (<https://developers.cloudflare.com/calls/turn/>) (which offers a free tier with 1,000 GB traffic per month) or [Open Relay](https://www.metered.ca/stun-turn) (<https://www.metered.ca/stun-turn>). You can also host an open source TURN server like [coturn](https://github.com/coturn/coturn) (<https://github.com/coturn/coturn>), [Pion TURN](https://github.com/pion/turn) (<https://github.com/pion/turn>), [Violet](https://github.com/paullouisageneau/violet) (<https://github.com/paullouisageneau/violet>), or [eturnal](https://github.com/processone/eturnal) (<https://github.com/processone/eturnal>). Keep in mind data will only go through the TURN server for peers that can't directly connect and will still be end-to-end encrypted.
3. Once you have a TURN server, configure Trystero with it like this:

```

const room = joinRoom(
  {
    // ...your app config
    turnConfig: [
      {
        // single string or list of strings of URLs to access TURN server
        urls: ['turn:your-turn-server.ok:1979'],
        username: 'username',
        credential: 'password'
      }
    ],
    'roomId'
  }
)

```

Running server-side (Node, Deno, Bun)

Trystero works wherever JS runs, including server-side like Node, Deno, or Bun. Why would you want to run something that helps you avoid servers on a server? One reason is if you want an always-on peer which can be useful for remembering the last state of data, broadcasting it to new users. Another reason might be to run peers that are lighter weight and don't need a full browser running, like an embedded device or Raspberry Pi.

Running server-side uses the same syntax as in the browser, but you need to import a polyfill for WebRTC support:

```

import {joinRoom} from 'trystero'
import {RTCPeerConnection} from 'node-datachannel/polyfill'

const room = joinRoom(
  {appId: 'your-app-id', rtcPolyfill: RTCPeerConnection},
  'your-room-name'
)

```

Supabase setup

To use the Supabase strategy:

1. Create a [Supabase \(<https://supabase.com>\)](https://supabase.com) project or use an existing one
2. On the dashboard, go to Project Settings -> API
3. Copy the Project URL and set that as the `appId` in the Trystero config, copy the `anon public` API key and set it as `supabaseKey` in the Trystero config

Firebase setup

If you want to use the Firebase strategy and don't have an existing project:

1. Create a [Firebase](https://firebase.google.com/) (<https://firebase.google.com/>) project
2. Create a new Realtime Database
3. Copy the `databaseURL` and use it as the `appId` in your Trystero config

► Optional: configure the database with security rules to limit activity:

API

`joinRoom(config, roomId, [onJoinError])`

Adds local user to room whereby other peers in the same namespace will open communication channels and send events. Calling `joinRoom()` multiple times with the same namespace will return the same room instance.

- `config` - Configuration object containing the following keys:

- `appId` - **(required)** A unique string identifying your app. When using Supabase, this should be set to your project URL (see [Supabase setup instructions](#)). If using Firebase, this should be the `databaseURL` from your Firebase config (also see `firebaseApp` below for an alternative way of configuring the Firebase strategy).
- `password` - **(optional)** A string to encrypt session descriptions via AES-GCM as they are passed through the peering medium. If not set, session descriptions will be encrypted with a key derived from the app ID and room name. A custom password must match between any peers in the room for them to connect. See [encryption](#) for more details.
- `relayUrls` - **(optional, □ BitTorrent, □ Nostr, □ MQTT only)** Custom list of URLs for the strategy to use to bootstrap P2P connections. These would be BitTorrent trackers, Nostr relays, and MQTT brokers, respectively. They must support secure WebSocket connections.
- `relayRedundancy` - **(optional, □ BitTorrent, □ Nostr, □ MQTT only)** Integer specifying how many torrent trackers to connect to simultaneously in case some fail. Passing a `relayUrls` option will cause this option to be ignored as the entire list will be used.
- `rtcConfig` - **(optional)** Specifies a custom [RTCCConfiguration](#) (<https://developer.mozilla.org/en-US/docs/Web/API/RTCCConfiguration>) for all peer connections.
- `turnConfig` - **(optional)** Specifies a custom list of TURN servers to use (see [Connection issues](#) section). Each item in the list should correspond to an [ICE server config object](#) (<https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/RTCPeerConnection#iceservers>). When passing a TURN config like this, Trystero's default STUN servers will also be used. To override this and use both custom STUN and TURN servers, instead pass the config via the above `rtcConfig.iceServers` option as a list of both STUN/TURN servers — this won't inherit Trystero's defaults.

- `rtcPolyfill` - **(optional)** Use this to pass a custom [RTCPeerConnection](https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/RTCPeerConnection) (<https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/RTCPeerConnection>)-compatible constructor. This is useful for running outside of a browser, such as in Node (still experimental).
- `supabaseKey` - **(required, ✎ Supabase only)** Your Supabase project's `anon public` API key.
- `firebaseApp` - **(optional, ✎ Firebase only)** You can pass an already initialized Firebase app instance instead of an `appId`. Normally Trystero will initialize a Firebase app based on the `appId` but this will fail if you've already initialized it for use elsewhere.
- `rootPath` - **(optional, ✎ Firebase only)** String specifying path where Trystero writes its matchmaking data in your database ('`__trystero__`' by default). Changing this is useful if you want to run multiple apps using the same database and don't want to worry about namespace collisions.
- `manualRelayReconnection` - **(optional, ✎ Nostr and ✎ BitTorrent only)** Boolean (default: `false`) that when set to `true` disables automatically pausing and resuming reconnection attempts when the browser goes offline and comes back online. This is useful if you want to manage this behavior yourself.
- `roomId` - A string to namespace peers and events within a room.
- `onJoinError(details)` - **(optional)** A callback function that will be called if the room cannot be joined due to an incorrect password. `details` is an object containing `appId`, `roomId`, `peerId`, and `error` describing the error.

Returns an object with the following methods:

- `leave()`

Remove local user from room and unsubscribe from room events.

- `getPeers()`

Returns a map of [RTCPeerConnection](https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection) (<https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection>)s for the peers present in room (not including the local user). The keys of this object are the respective peers' IDs.

- `addStream(stream, [targetPeers], [metadata])`

Broadcasts media stream to other peers.

- `stream` - A `MediaStream` with audio and/or video to send to peers in the room.
- `targetPeers` - **(optional)** If specified, the stream is sent only to the target peer ID (string) or list of peer IDs (array).

- metadata - (**optional**) Additional metadata (any serializable type) to be sent with the stream. This is useful when sending multiple streams so recipients know which is which (e.g. a webcam versus a screen capture). If you want to broadcast a stream to all peers in the room with a metadata argument, pass `null` as the second argument.

- `removeStream(stream, [targetPeers])`

Stops sending previously sent media stream to other peers.

- stream - A previously sent `MediaStream` to stop sending.
- targetPeers - (**optional**) If specified, the stream is removed only from the target peer ID (string) or list of peer IDs (array).

- `addTrack(track, stream, [targetPeers], [metadata])`

Adds a new media track to a stream.

- track - A `MediaStreamTrack` to add to an existing stream.
- stream - The target `MediaStream` to attach the new track to.
- targetPeers - (**optional**) If specified, the track is sent only to the target peer ID (string) or list of peer IDs (array).
- metadata - (**optional**) Additional metadata (any serializable type) to be sent with the track. See `metadata` notes for `addStream()` above for more details.

- `removeTrack(track, [targetPeers])`

Removes a media track.

- track - The `MediaStreamTrack` to remove.
- targetPeers - (**optional**) If specified, the track is removed only from the target peer ID (string) or list of peer IDs (array).

- `replaceTrack(oldTrack, newTrack, [targetPeers], [metadata])`

Replaces a media track with a new one.

- oldTrack - The `MediaStreamTrack` to remove.

- newTrack - A `MediaStreamTrack` to attach.
- targetPeers - (**optional**) If specified, the track is replaced only for the target peer ID (string) or list of peer IDs (array).
- `onPeerJoin(callback)`

Registers a callback function that will be called when a peer joins the room. If called more than once, only the latest callback registered is ever called.

- callback(peerId) - Function to run whenever a peer joins, called with the peer's ID.

Example:

```
onPeerJoin(peerId => console.log(`${peerId} joined`))
```

- `onPeerLeave(callback)`

Registers a callback function that will be called when a peer leaves the room. If called more than once, only the latest callback registered is ever called.

- callback(peerId) - Function to run whenever a peer leaves, called with the peer's ID.

Example:

```
onPeerLeave(peerId => console.log(`${peerId} left`))
```

- `onPeerStream(callback)`

Registers a callback function that will be called when a peer sends a media stream. If called more than once, only the latest callback registered is ever called.

- callback(stream, peerId, metadata) - Function to run whenever a peer sends a media stream, called with the peer's stream, ID, and optional metadata (see `addStream()` above for details).

Example:

```
onPeerStream((stream, peerId) =>
  console.log(`got stream from ${peerId}`, stream)
)
```

- `onPeerTrack(callback)`

Registers a callback function that will be called when a peer sends a media track. If called more than once, only the latest callback registered is ever called.

- `callback(track, stream, peerId, metadata)` - Function to run whenever a peer sends a media track, called with the the peer's track, attached stream, ID, and optional metadata (see `addTrack()` above for details).

Example:

```
onPeerTrack((track, stream, peerId) =>
  console.log(`got track from ${peerId}`, track)
)
```

• `makeAction(actionId)`

Listen for and send custom data actions.

- `actionId` - A string to register this action consistently among all peers.

Returns an array of three functions:

1. Sender

- Sends data to peers and returns a promise that resolves when all target peers are finished receiving data.
- `(data, [targetPeers], [metadata], [onProgress])`
 - `data` - Any value to send (primitive, object, binary). Serialization and chunking is handled automatically. Binary data (e.g. `Blob`, `TypedArray`) is received by other peer as an agnostic `ArrayBuffer`.
 - `targetPeers` - **(optional)** Either a peer ID (string), an array of peer IDs, or `null` (indicating to send to all peers in the room).
 - `metadata` - **(optional)** If the data is binary, you can send an optional metadata object describing it (see [Binary metadata](#)).
 - `onProgress` - **(optional)** A callback function that will be called as every chunk for every peer is transmitted. The function will be called with a value between 0 and 1 and a peer ID. See [Progress updates](#) for an example.

2. Receiver

- Registers a callback function that runs when data for this action is received from other peers.
- `(data, peerId, metadata)`

- `data` - The value transmitted by the sending peer. Deserialization is handled automatically, i.e. a number will be received as a number, an object as an object, etc.
- `peerId` - The ID string of the sending peer.
- `metadata` - (**optional**) Optional metadata object supplied by the sender if `data` is binary, e.g. a filename.

3. Progress handler

- Registers a callback function that runs when partial data is received from peers. You can use this for tracking large binary transfers. See [Progress updates](#) for an example.
- `(percent, peerId, metadata)`
 - `percent` - A number between 0 and 1 indicating the percentage complete of the transfer.
 - `peerId` - The ID string of the sending peer.
 - `metadata` - (**optional**) Optional metadata object supplied by the sender.

Example:

```
const [sendCursor, getCursor] = room.makeAction('cursormove')

window.addEventListener('mousemove', e => sendCursor([e.clientX, e.clientY]))

getCursor(([x, y], peerId) => {
  const peerCursor = cursorMap[peerId]
  peerCursor.style.left = x + 'px'
  peerCursor.style.top = y + 'px'
})
```

• ping(`peerId`)

Takes a peer ID and returns a promise that resolves to the milliseconds the round-trip to that peer took. Use this for measuring latency.

- `peerId` - Peer ID string of the target peer.

Example:

```
// log round-trip time every 2 seconds
room.onPeerJoin(peerId =>
  setInterval(
    async () => console.log(`took ${await room.ping(peerId)}ms`),
    2000
  )
)
```

selfId

A unique ID string other peers will know the local user as globally across rooms.

getRelaySockets()

(**BitTorrent**, **Nostr**, **MQTT only**) Returns an object of relay URL keys mapped to their WebSocket connections. This can be useful for determining the state of the user's connection to the relays and handling any connection failures.

Example:

```
console.log(trystero.getRelaySockets())
// => Object {
//   "wss://tracker.webtorrent.dev": WebSocket,
//   "wss://tracker.openwebtorrent.com": WebSocket
// }
```

pauseRelayReconnection()

(**Nostr**, **BitTorrent only**) Normally Trystero will try to automatically reconnect to relay sockets unless `manualRelayReconnection: true` is set in the room config. Calling this function stops relay reconnection attempts until `resumeRelayReconnection()` is called.

resumeRelayReconnection()

(**Nostr**, **BitTorrent, only**) Allows relay reconnection attempts to resume. (See `pauseRelayReconnection()` above).

getOccupants(config, roomId)

(**Firebase only**) Returns a promise that resolves to a list of user IDs present in the given namespace. This is useful for checking how many users are in a room without joining it.

- `config` - A configuration object
- `roomId` - A namespace string that you'd pass to `joinRoom()`.

Example:

```
console.log((await trystero.getOccupants(config, 'the_scope')).length)
// => 3
```

Strategy comparison

one-time setup¹ bundle size²

<input type="checkbox"/> Nostr	none	8K
<input type="checkbox"/> MQTT	none	75K
<input type="checkbox"/> BitTorrent	none	5K
<input checked="" type="checkbox"/> Supabase	~5 mins	28K
<input type="checkbox"/> Firebase	~5 mins	45K
<input type="checkbox"/> IPFS	none	119K

¹ All strategies except Supabase and Firebase require zero setup. Supabase and Firebase are managed strategies which require setting up an account.

² Calculated via Terser minification + Brotli compression.

How to choose

Trystero's unique advantage is that it requires zero backend setup and uses decentralized infrastructure in most cases.

This allows for frictionless experimentation and no single point of failure. One potential drawback is that it's difficult to guarantee that the public infrastructure it uses will always be highly available, even with the redundancy techniques Trystero uses. While the other strategies are decentralized, the Supabase and Firebase strategies are a more managed approach with greater control and an SLA, which might be more appropriate for "production" apps.

Trystero makes it trivial to switch between strategies — just change a single import line and quickly experiment:

```
import {joinRoom} from 'trystero/[nostr|mqtt|torrent|supabase|firebase|ipfs]'
```

Trystero by [Dan Motzenbecker \(https://oxism.com\)](https://oxism.com)