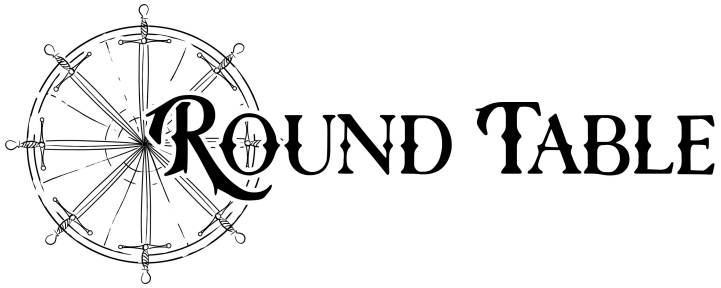

Figure 1: alt text

Introduction

RoundTable is an infrastructure project built on libp2p and Solana

It provides a browser compatible library, server and smart contract to manage a simple decentralised P2P messaging system.

The aim of RoundTable is to provide a messaging layer allowing browser to browser messaging between users of Dapps

But what is it?

A RoundTable deployment

A RoundTable deployment consists of a set of bootstrap servers, a configuration managed by the network operators, a set of user accounts created by users, and user clients running in the users browsers. Note that whilst a basic front end is included in the `round-table-react` project, the general intention is for people to build the `round-table` library into their own applications backends.

Bootstrap servers

Bootstrap servers must be setup with https certs and domains. They must run the `Server` application, specifying their local hostname and the `Owner` address of the deployment. The server fetches the network configuration from the blockchain based on the `Owner` address, starts sharing the specified `channels` and connects to the other `bootstrap` servers. The server is a thin wrapper around `libp2p` integrating with some solana functionality via `web3.js`.

Clients

Clients are an instance of `RoundTable` running in any context, they use the `Owner` address of the deployment to connect to to get the `bootstrap` server list and connect to the network. The library is suitable to run in browsers. The core functionality provided by `RountTable` is a basic chat system, presence indication showing who is connected to the network, and a matching engine, to allow groups on the network looking for a particular thing to find each other

But why is it?

Blockchains are very good at storing state and having rules governing the modification of state. They aren't very good for sending messages. Even with Solana transactions being significantly faster than many other chains, managing processes which involve a "back and forth" between parties can be very slow. The natural answer is to pair a fast messaging layer with a safe blockchain layer for settlement, aiming to strike a balance where applications are both fast and secure.

There are very easy answers to this problem, simply use a centralised server, and the server posts the results to the chain. This is a tried and tested method, but it is not decentralised. If the server goes down the application will no longer function. In the worst case, the centralised server can act dishonestly such as preventing messages being propogated, and if messages themselves are not signed manipulate these too.

In general the centralised messaging pattern does not mesh well with the decentralised nature of the blockchain, at best erasing some of its benefits. This makes it desirable to solve the problem in a more decentralised manner.

But why is RoundTable more decentralised?

Whilst a roundtable deployment still relies on bootstrap servers, the servers themselves and bootstrap list can be managed by a community or a DAO. The network management contract can be called from another contract via CPI allowing the parent contract to put further rules on managing bootstrap servers, channels and user accounts, and to use multi sig for some actions.

Because the servers and clients work by fetching the network configuration from the smart contract the `centralised` element of the network is stored on the `decentralised` blockchain.

And because the server itself is open code, anyone can run a bootstrap server, and it is down to the community managing the deployment to add them to the bootstrap list or not. The bootstraps can be removed to, should one go offline, or perhaps not have been performing or behaving well.

Ok but how is it?

Almost all the heavy lifting in RoundTable has already been done by the creators of `Solana` and `libp2p` (Spun off from IPFS by Protocol Labs)

RoundTable just provides a thin management shim and exposes a particular set of functionality, tying the two together and adding a couple of protocols.

All the messaging is performed using `libp2p`s `pubsub` functionality, with protocols running on their own channel[s].

Current State

Features

P2P

- ☒ Basic chat function
- ☒ Presence broadcasts and manager
- ☒ Track and provide API for present users
- ☒ Make messaging ID tied to solana wallet
- ☒ Build matching protocol and engine
  - ☐ Allow matches with greater than 2 participants
- ☒ Get bootstrap list from smart contract
- ☐ Check pubkey in messages matches channel key
  - (The libp2p key and solana wallet key are separate. Must at least verify keys match accounts, should probably build an address book from the chain data?)

Smart contract

- ☒ Allow creation of a "table"
- ☒ Allow table to have a list of bootstrap servers
- ☒ Allow table to have a list of channels
- ☒ Allow users to create a "seat" at a table
- ☒ Optionally require the signature of a table when adding users (so parent contract can choose who can join)
- ☒ CLI tool for managing contracct

Relay server

- ☒ js-libp2p bootstrap server
- ☒ server can pin channels from smart contract
- ☒ server can get bootstrap list from smart contract
- ☐ server can filter connections so only users with account can join
- ☐ Docker image and instructions to bring up round-table servers, or create a sibling network

Examples

- ☐ Clone network bringup tutorial
- ☐ Managing RoundTable via another smart contract via CPI (make a closed network)

round-table-react

Repo here

- ☒ Chat function demo
- ☒ Presence demo
- ☒ Matching demo
- ☐ Make Beautiful
  - ☐ (Learn to actually be OK at web front end design to be able to do this)
    - (pls help)

Future features

- ☐ POMS? (Proof of message sent, rollup all messages into blocks, post merkle root to chain and ?)
- ☐ Build an actual L2?

Smart contract examples

Interact with the smart contract via the tool in `util/contract_api`. My apologies there is no proper documentation in this tool yet and the CLI is not nice

For the following examples the participants are:

Table owner: `69GoySbK6vc9QyWsCYTMUjpQXCocbDJansszPTEaEtMp` User: `DRSRtpAcN9emERXqjVLtLb5iCWDWt9VJ2LzVpUfuuKZ8` Users P2P ID: ‘`And the table and user keys are stored at`contract/keys/table1.json`and`contract/keys/user1.json‘

Fetching data

```
## Get Bootstrap and channels data
npx ts-node util/contract_api.ts getTableData 69GoySbK6vc9QyWsCYTMUjpQXCocbDJansszPTEaEtMp

## Get the data for a user (seat)
npx ts-node util/contract_api.ts getSeatData 69GoySbK6vc9QyWsCYTMUjpQXCocbDJansszPTEaEtMp DR
```

Updating data

```
# Create a new deployment (Table)
npx ts-node util/contract_api.ts createTable contract/keys/table1.json 69GoySbK6vc9QyWsCYTMU
```

```
# Add a bootstrap server
npx ts-node util/contract_api.ts deleteTable contract/keys/table1.json 69GoySbK6vc9QyWsCYTMU

# Add a bootstrap server
npx ts-node util/contract_api.ts addBootstrap contract/keys/table1.json 69GoySbK6vc9QyWsCYTM

# Delete a bootstrap server
npx ts-node util/contract_api.ts deleteBootstrap contract/keys/table1.json 69GoySbK6vc9QyWsC

# Add a channel
npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj

# Delete a channel
npx ts-node util/contract_api.ts deleteChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYT

# Add a user (seat)
npx ts-node util/contract_api.ts addSeat contract/keys/user1.json 69GoySbK6vc9QyWsCYTMUjpQXC

# Remove a user
npx ts-node util/contract_api.ts deleteSeat contract/keys/user1.json 69GoySbK6vc9QyWsCYTMUjp
```

Actually bringing up a network

Same keys as before ^^

```
# Generate bootstrap server IDs
/// tbd

# Add bootstrap servers
npx ts-node util/contract_api.ts addBootstrap contract/keys/table1.json 69GoySbK6vc9QyWsCYTM
npx ts-node util/contract_api.ts addBootstrap contract/keys/table1.json 69GoySbK6vc9QyWsCYTM
npx ts-node util/contract_api.ts addBootstrap contract/keys/table1.json 69GoySbK6vc9QyWsCYTM

# Add core channels

npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj
npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj
npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj
npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj
npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj
npx ts-node util/contract_api.ts addChannel contract/keys/table1.json 69GoySbK6vc9QyWsCYTMUj

# Now bringup bootstrap servers
# ....
# ....
```

```
# Clients can now join the network, using ....
# TODO finish client bootstrap import and show example
```

Note that a project adding further protocols would add more channels here for their protocols

Examples

For now check out the tests in `test/` which excercise the core interfaces for the `Matcher`, `Chat`, and `Presence` protocols. Also refer to their implementations,`pub` and `sub` are exposed via the `RoundTable` instace to make it easy to build extra protocols on top of a RoundNet deployment.