MultiTurn: A Turn-Based Multiplayer Game Framework

Markus Feng

2019-03-09

1 Introduction

The goal of this project is to develop a distributed networking framework, specifically designed for turn-based multiplayer games. The final deliverable of this project will be a library to provide an implementation of a high level networking framework, which that can be used by developers to create turn-based multiplayer games and other similar projects. In addition, I will develop a simple multiplayer game that runs on this framework, both as a proof of functionality and as a platform for testing the usage of the library.

One of the current issues in game development is that extending a single client system to multiple clients is very difficult, and the networking code of many games tend to require solutions to similar problems. These problems include ensuring correctness of concurrent code, handling disconnections, ensuring consistency, data validation, and data security. My goal is to create a system that implements solutions to some of these common problems and integrate this system into a package that will allow developers to focus more on other aspects of design and development.

Even though this project specifically targets networking code for turn-based multiplayer games, the same solutions can apply for a diverse range of applications. Examples include, online chat programs, quizzes, or any software that requires user input and needs to ensure that the results are consistent among multiple clients. Additionally, if real-time functionality ends up being feasible to implement, the potential range of usage grows even wider.

2 Design Principles

The guiding design principles behind this design are correctness, ease of use, generality, and safety. When used correctly, the framework should produce results that are consistently correct, without usually needing to deal with edge cases and other special conditions. Additionally, the framework should be easy to use under the condition that the developer understands the principles behind the technologies used in this framework. Therefore, an experienced developer should be able to use this framework to develop the network aspect of a game in less time, but to the same level of completeness, when compared to developing such a system from scratch. Finally, the framework should handle aspects of safety and security of network communications such that the developer does not need to do so manually.

3 Layers

The framework is divided into several layers, each of which has multiple possible implementations. However, certain features may differ depending on whether the runner is the client or the server. This spec will describe each layer's specification, along with at least one protocol implementation at each layer.

- Network layer
  - Handles the communication to a client's Network layer, most likely using some existing network protocol such as TCP or socket.io

- Authentication layer
  - Handles the authentication of the client's identity, ensuring that the client is who they claim they are
  - Handles keep-alive with the client, ensuring that delivery completes even if the client loses or changes connection
- Synchronization layer
  - Ensure that the client is up to date with the current state of the server
  - Ensure that the client can only access the information that they have access to
- Validation layer
  - Validates requests, both the data type and the content
  - Automatically generates validators based on type definitions
  - Typesafe interface
- Interface layer
  - Exposes an easy to use API to allow the server to send requests to the client and handle requests and responses from the client

4 Network Layer

(Note: `S` stands for the serialize command, which combines two strings into one.)

The purpose of the network layer is to allow messages to be sent from one endpoint to another. It does so by using the "request/response" model of networking. In this model, an endpoint sends a request to another (a pair `<Key>`, `<Value>`, and the other may respond with a `<Response>`). This allows the higher layers to abstract away much of the networking

Connection refused

- `A: key: '_refused'`

Connection closed

- `A: key: '_close'`

A to B request

- `A: key: '_request', message: S[<Key>, <Value>]`
- `B: key: '_response', message: S[<Key>, <Response>]`

Interface

```
export interface NetworkLayer {
  listen(): void;
  addConnectionListener(callback: (e: ConnectionEvent) => void): void;
}

export interface ConnectionEvent {
  accept(): Socket;
  reject(): void;
}

export interface Socket {
  addRequestListener(callback: (e: RequestEvent) => void): void;
  request(key: string, message: string): CancelablePromise<string>;
  close(): void;
}

export interface RequestEvent {
  readonly key: string;
  readonly message: string;
  respond(message: string): void;
}
```

5 Authentication Layer

The purpose of the authentication layer is to allow ”users” to persist across multiple connections. It does so by requiring an authentication token from the client on each request. The authentication token identifies the player, rather than the connection. The client is responsible for storing the authentication locally so it is maintained across multiple sessions.

User register

- `User Request: key: '_register'`
- `Server Response: <Token>`

User login

- `User Request: key: '_login', message: <Token>`
- Login success:
  - `Server Response: '_login_success'`
- Login fail:
  - `Server Response: '_login_fail'`

User to server request

- `User Request: key: <token>, message: S[<Key>, <Value>]`
- `Server Response: <Response>`

Server to user request

- `Server Request: key: <Key>, message: <Value>`
- `User Response: <Response>`

6 Synchronization layer

The purpose of the synchronization layer is to ensure that the server and all clients maintain the same state, that no requests are lost, and that requests and responses are processed in order.

Server Interface

```
type IdType = string;
type StateType = string;

export interface ServerSyncLayer {

  state: StateManager;

  listen(): void;

  // Send state update to all players without a request
  update(): SyncResponse;
  getUser(id: IdType): SyncUser | undefined;
  getUsers(): SyncUser[];
  requestAll(key: string, value: string): SyncResponse;
}

export interface SyncUserEvent {
  accept(): SyncUser;
  reject(): void;
}

export interface SyncUser {
  readonly id: IdType;
  // On request, send a state update to all players
  request(key: string, value: string, timeout?: number): SyncResponse;
  close(): void;
}

export interface SyncStateEvent {
  readonly id: IdType;
  readonly key: string;
  readonly message: string;
}

export interface SyncResponse {
  readonly result?: CancelablePromise<string>;
  readonly updates: Map<IdType, CancelablePromise<void>>;
  cancel(): void;
}

export interface StateManager {
  onNewUser(e: SyncUserEvent): void;
  getState(id: IdType): StateType;
}
```

Client Interface

```
export interface ClientSyncLayer {
  responder: ClientSyncResponder;
  listen(): void;
}

export interface ClientSyncResponder {
  onUpdateState(e: ClientSyncStateEvent): void;
  onRequest(e: ClientSyncRequestEvent): Promise<string>;
}

export interface ClientSyncStateEvent {
  readonly state: string;
}

export interface ClientSyncRequestEvent {
  readonly key: string;
  readonly message: string;
}

export interface ClientSyncCombinedEvent {
  readonly key: string;
  readonly message: string;
  readonly state: string;
}
```

7 Validation Layer

The purpose of the validation layer is to ensure that data passed to the server is valid. This is because invalid data can be a security vulnerability, especially when using a weakly typed runtime language like Javascript. This layer uses existing type definitions and comments to automatically generate validators for remote method calls from the server to the client, all while maintaining type safety and reducing duplicate code.

Technical Details

Input validation is completed by using typescript-json-schema, a package that converts a type definition of a typescript file to a JSON schema object. To enable validation, the client class method needs to have the `@remote(<type>)` where type is optionally the type of object returned by the method (if one cannot be inferred directly by the return arguments, such as when using a Promise). Then, calls to the method need to be wrapped in a remote call wrapper. When a remote call request is sent by the server, the client sends a response as a JSON encoded string. The string is parsed with JSON and then verified against the JSON schema of the type definition as specified by the return type of the remote call, using ajv. If this verification passes, the response is accepted.

Server Interface

```

// @remote() decorator
function remote(namedType?: {name: string});

interface Validator {

    // For calling standard functions remotely
    public call<T>(t: (() => T)): () => Promise<T>;

    // For calling promises remotely
    public flatCall<T>(t: (() => Promise<T>)): () => Promise<T>;
}

function setupRemote<T>(t: T, validator: RemoteValidator): T;
```

Example

```
export default class Player {

  // The decorator indicates that the method is a remote method
  @remote()
  public getMove(): Move {
    return new Move(1, 1);
  }

  // The type is needed because the resulting type to be
  // transmitted should be Move, not Promise<Move>
  @remote(Move)
  public getDelayedMove(): Promise<Move> {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        resolve(new Move(1, 0));
      }, 1000);
    });
  }
}

// Usage
function main() {
  const remote = new Validator(...);
  const getMove = remote.call(Player.prototype.getMove);
  const getDelayedMove = remote.flatCall(Player.prototype.getDelayedMove);

  const move1 = await getMove();
  console.log(move1);
  const move2 = await getDelayedMove();
  console.log(move2);
}

class Move {
    /**
     * The x-coordinate of the move.
     * @minimum 0
     * @maximum 2
     * @TJS-type integer
     */
    public x: number;

    /**
     * The y-coordinate of the move.
     * @minimum 0
     * @maximum 2
     * @TJS-type integer
     */
    public y: number;
}
```

8 Interface layer

The purpose of the interface layer is to provide an easy-to-use publicly accessible interface to ease the creation of correct and secure games. The interface layer exposes an API that allows the developer to focus on developing game logic, ignoring as much of the common implementation details as possible. Ideally, using the interface layer, writing a multi-system turn-based game would be as straightforward as writing the same game to work on a single system.

Server Interface

```

// Type of player: R
// Type of state: T
class Server<R, T> {

  public readonly maxPlayers: number;

  public constructor(
    private mainLoop: (server: Server<R, T>) => Promise<void>,
    remoteGenerator: new () => R,
    state: T,
    options: ServerOptions<R, T>
  );

  public async start(): Promise<void>;

  public getPlayers(): Array<Player<R>>;

  public getCurrentPlayer(): Player<R>;

  public getTurn(): number;

  public getMaxPlayers(): number;

  // Warning: standard turns already increases turn count,
  // setTurn disables next turn increment
  public setTurn(turn: number);

  public gameOver(message: string);
}
```

```
type StateMask<R, T> = (state: T, player: Player<R>) => string;

export interface ServerOptions<R, T> {
  syncLayer: ServerSyncLayer;
  stateMask: StateMask<R, T>;
  maxPlayers: number;
  typePath: string;
  standardTurns: boolean;
}
```

Example: Tic Tac Toe

For the implementation of the `Board` and `Remote` class, check the Appendix.

```
import * as express from 'express';
import * as http from 'http';
import * as socketio from 'socket.io';
import { fillDefault } from '../../multiturn/game/default';
import Server from '../../multiturn/game/server';
import Board from './board';
import Remote from './remote';

function getRunner(state: Board) {
  return async function runner(game: Server<Remote, Board>): Promise<void> {
    const player = game.getCurrentPlayer();
    const board = state;
    const validator = (possibleMove: Move) => !board.occupied(possibleMove);
    let move;
    do {
      move = await player.remote.getMove();
    } while (!validator(move));
    console.log(`Valid move: {${move.x}, ${move.y}}`);
    board.occupy(move, player.num);
    const win = board.checkVictory();
    if (win >= 0) {
      game.gameOver(player.num.toString());
      return;
    }
    const full = board.checkFull();
    if (full) {
      game.gameOver((-1).toString());
    }
  };
}

export default function main() {
  const app = express();

  const clientPath = `${__dirname}/../../../dist`;
  app.use(express.static(clientPath));

  const server = http.createServer(app);

  const io = socketio(server);

  const options = fillDefault({
    typePath: './src/server/tictactoe-new/game.ts'
  }, io);
  const state = new Board();
  const gameServer = new Server<Remote, Board>(
    getRunner(state), Remote, state, options);
  gameServer.start().then(() => {
    console.log('Closing server.');
    server.close();
  });

  const port = process.env.PORT || 8080;
  server.listen(port, () => {
    console.log('Server started on ' + port);
  });

}


class Move {
    /**
     * The x-coordinate of the move.
     * @minimum 0
     * @maximum 2
     * @TJS-type integer
     */
    public x: number;

    /**
     * The y-coordinate of the move.
     * @minimum 0
     * @maximum 2
     * @TJS-type integer
     */
    public y: number;
}
```

9 Current Progress

Layer	Design	Implementation	Integration	Testing
Network Layer	✓	✓	✓	✓
Authentication Layer	✓	✓	✓	✓
Synchronization Layer	✓	...	✓	
Validation Layer	✓	✓	✓	✓
Interface Layer	✓	...	✓	
Sample Game	✓	...		

(✓ : complete, · · · : under progress)

10 Appendix

10.1 Implementation of Board for Tic-Tac-Toe example

```
import Move from './move';

const rows = 3;
const cols = 3;

export default class Board {
  /**
    * The spaces of the board, represented in a two-dimensional array
    * The first coordinate is the x coordinate
    * The second coordinate is the y coordinate
    * A value of -1 indicates unoccupied
    * A nonnegative value indiciates occupied by the specified player
    * Strict validation is not needed because the client never passes an
    * instance to the server
    */
  public spaces: number[][];

  public constructor() {
    const spaces: number[][] = [];
    this.spaces = spaces;
    for (let i = 0; i < cols; i++) {
      const col: number[] = [];
      spaces.push(col);
      for (let j = 0; j < rows; j++) {
        col.push(-1);
      }
    }
  }

  public occupied(move: Move): boolean {
    return this.spaces[move.x][move.y] !== -1;
  }

  public occupy(move: Move, n: number) {
    this.spaces[move.x][move.y] = n;
  }

  public checkFull(): boolean {
    for (let i = 0; i < rows; i++) {
      for (let j = 0; j < cols; j++) {
        if (this.spaces[i][j] === -1) {
          return false;
        }
      }
    }
    return true;
  }

  public checkVictory() {
    let arr: number[];
    const lines: number[][] = [];
    const spaces = this.spaces;
    // Check rows
    for (let i = 0; i < rows; i++) {
        arr = [];
        for (let j = 0; j < cols; j++) {
            const player = spaces[j][i];
            arr.push(player);
        }
        lines.push(arr);
    }
    // Check columns
    for (let i = 0; i < cols; i++) {
        arr = [];
        for (let j = 0; j < rows; j++) {
            const player = spaces[i][j];
            arr.push(player);
        }
        lines.push(arr);
    }
    // Check diagonals
    arr = [];
    for (let i = 0; i < rows; i++) {
        const player = spaces[i][i];
        arr.push(player);
    }
    lines.push(arr);
    arr = [];
    for (let i = 0; i < rows; i++) {
        const player = spaces[cols - i - 1][i];
        arr.push(player);
    }
    lines.push(arr);
    // Returns the number if all numbers are the same for a given line
    for (const line of lines) {
        let match = true;
        const v = line[0];
        for (let j = 1; j < line.length; j++) {
            if (line[j] !== v) {
                match = false;
                break;
            }
        }
        if (match && v >= 0) {
            return v;
        }
    }
    return -1;
  }
}
```

10.2 Implementation of `Remote` for Tic-Tac-Toe example

```
import { Client } from '../../multiturn/game/client';
import { remote } from '../../multiturn/remote/remote';
import { ClientSyncStateEvent } from '../../multiturn/sync/client';
import Board from './board';
import Move from './move';

export default class Remote implements Client<Remote> {
  private playerNum!: number;
  private state!: Board;

  @remote(Move)
  public getMove(): Promise<Move> {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        const move = this.getRandomMove();
        resolve(move);
      }, 1000);
    });
  }

  // Client methods
  public assignNumber(num: number) {
    this.playerNum = num;
  }

  public updateState(e: ClientSyncStateEvent) {
    this.state = JSON.parse(e.state) as Board;
  }

  public getRemote(): Remote {
    return this;
  }

  public gameOver(message: string) {
    console.log(`Game over! ${message}`);
  }

  private getRandomMove(): Move {
    const randomX = Math.floor(Math.random() * 3);
    const randomY = Math.floor(Math.random() * 3);
    return new Move(randomX, randomY);
  }
}
```