const \_ = global.\_;

const fs = require('fs');

const Q = require('bluebird');

const spawn = require('child\_process').spawn;

const { dialog } = require('electron');

const Windows = require('./windows.js');

const Settings = require('./settings');

const log = require('./utils/logger').create('EthereumNode');

const logRotate = require('log-rotate');

const EventEmitter = require('events').EventEmitter;

const Sockets = require('./socketManager');

const ClientBinaryManager = require('./clientBinaryManager');

const DEFAULT\_NODE\_TYPE = 'geth';

const DEFAULT\_NETWORK = 'main';

const UNABLE\_TO\_BIND\_PORT\_ERROR = 'unableToBindPort';

const UNABLE\_TO\_SPAWN\_ERROR = 'unableToSpan';

const PASSWORD\_WRONG\_ERROR = 'badPassword';

const NODE\_START\_WAIT\_MS = 3000;

/\*\*

\* Etheruem nodes manager.

\*/

class EthereumNode extends EventEmitter {

constructor() {

super();

this.STATES = STATES;

this.\_loadDefaults();

this.\_node = null;

this.\_type = null;

this.\_network = null;

this.\_socket = Sockets.get('node-ipc', Settings.rpcMode);

this.on('data', \_.bind(this.\_logNodeData, this));

}

get isOwnNode() {

return !!this.\_node;

}

get isExternalNode() {

return !this.\_node;

}

get isIpcConnected() {

return this.\_socket.isConnected;

}

get type() {

return this.isOwnNode ? this.\_type : null;

}

get network() {

return this.isOwnNode ? this.\_network : null;

}

get isEth() {

return this.\_type === 'eth';

}

get isGeth() {

return this.\_type === 'geth';

}

get isMainNetwork() {

return this.network === 'main';

}

get isTestNetwork() {

return this.network === 'test';

}

get state() {

return this.\_state;

}

get stateAsText() {

switch (this.\_state) {

case STATES.STARTING:

return 'starting';

case STATES.STARTED:

return 'started';

case STATES.CONNECTED:

return 'connected';

case STATES.STOPPING:

return 'stopping';

case STATES.STOPPED:

return 'stopped';

case STATES.ERROR:

return 'error';

}

}

set state(newState) {

this.\_state = newState;

this.emit('state', this.state, this.stateAsText);

}

get lastError() {

return this.\_lastErr;

}

set lastError(err) {

return this.\_lastErr = err;

}

/\*\*

\* This method should always be called first to initialise the connection.

\* @return {Promise}

\*/

init() {

return this.\_socket.connect(Settings.rpcConnectConfig)

.then(() => {

this.state = STATES.CONNECTED;

this.emit('runningNodeFound');

})

.catch((err) => {

log.warn('Failed to connect to node. Maybe it\'s not running so let\'s start our own...');

log.info(`Node type: ${this.defaultNodeType}`);

log.info(`Network: ${this.defaultNetwork}`);

// if not, start node yourself

return this.\_start(this.defaultNodeType, this.defaultNetwork)

.catch((err) => {

log.error('Failed to start node', err);

throw err;

});

});

}

restart(newType, newNetwork) {

return Q.try(() => {

if (!this.isOwnNode) {

throw new Error('Cannot restart node since it was started externally');

}

log.info('Restart node', newType, newNetwork);

return this.stop()

.then(() => {

Windows.loading.show();

})

.then(() => {

return this.\_start(newType || this.type, newNetwork || this.network);

})

.then(() => {

Windows.loading.hide();

})

.catch((err) => {

log.error('Error restarting node', err);

throw err;

});

});

}

/\*\*

\* Stop node.

\*

\* @return {Promise}

\*/

stop() {

if (!this.\_stopPromise) {

return new Q((resolve, reject) => {

if (!this.\_node) {

return resolve();

}

this.state = STATES.STOPPING;

log.info(`Stopping existing node: ${this.\_type} ${this.\_network}`);

this.\_node.stderr.removeAllListeners('data');

this.\_node.stdout.removeAllListeners('data');

this.\_node.stdin.removeAllListeners('error');

this.\_node.removeAllListeners('error');

this.\_node.removeAllListeners('exit');

this.\_node.kill('SIGINT');

// after some time just kill it if not already done so

const killTimeout = setTimeout(() => {

if (this.\_node) {

this.\_node.kill('SIGKILL');

}

}, 8000 /\* 8 seconds \*/);

this.\_node.once('close', () => {

clearTimeout(killTimeout);

this.\_node = null;

resolve();

});

})

.then(() => {

this.state = STATES.STOPPED;

this.\_stopPromise = null;

});

} else {

log.debug('Disconnection already in progress, returning Promise.');

}

return this.\_stopPromise;

}

getLog() {

return Settings.loadUserData('node.log');

}

/\*\*

\* Send Web3 command to socket.

\* @param {String} method Method name

\* @param {Array} [params] Method arguments

\* @return {Promise} resolves to result or error.

\*/

send(method, params) {

return this.\_socket.send({

method,

params,

});

}

/\*\*

\* Start an ethereum node.

\* @param {String} nodeType geth, eth, etc

\* @param {String} network network id

\* @return {Promise}

\*/

\_start(nodeType, network) {

log.info(`Start node: ${nodeType} ${network}`);

const isTestNet = (network === 'test');

if (isTestNet) {

log.debug('Node will connect to the test network');

}

return this.stop()

.then(() => {

return this.\_\_startNode(nodeType, network)

.catch((err) => {

log.error('Failed to start node', err);

this.\_showNodeErrorDialog(nodeType, network);

throw err;

});

})

.then((proc) => {

log.info(`Started node successfully: ${nodeType} ${network}`);

this.\_node = proc;

this.state = STATES.STARTED;

Settings.saveUserData('node', this.\_type);

Settings.saveUserData('network', this.\_network);

return this.\_socket.connect(Settings.rpcConnectConfig, {

timeout: 30000, /\* 30s \*/

})

.then(() => {

this.state = STATES.CONNECTED;

})

.catch((err) => {

log.error('Failed to connect to node', err);

if (err.toString().indexOf('timeout') >= 0) {

this.emit('nodeConnectionTimeout');

}

this.\_showNodeErrorDialog(nodeType, network);

throw err;

});

})

.catch((err) => {

// set before updating state so that state change event observers

// can pick up on this

this.lastError = err.tag;

this.state = STATES.ERROR;

// if unable to start eth node then write geth to defaults

if (nodeType === 'eth') {

Settings.saveUserData('node', 'geth');

}

throw err;

});

}

/\*\*

\* @return {Promise}

\*/

\_\_startNode(nodeType, network) {

this.state = STATES.STARTING;

this.\_network = network;

this.\_type = nodeType;

const client = ClientBinaryManager.getClient(nodeType);

let binPath;

if (client) {

binPath = client.binPath;

} else {

throw new Error(`Node "${nodeType}" binPath is not available.`);

}

log.info(`Start node using ${binPath}`);

return new Q((resolve, reject) => {

this.\_\_startProcess(nodeType, network, binPath)

.then(resolve, reject);

});

}

/\*\*

\* @return {Promise}

\*/

\_\_startProcess(nodeType, network, binPath) {

return new Q((resolve, reject) => {

log.trace('Rotate log file');

// rotate the log file

logRotate(Settings.constructUserDataPath('node.log'), { count: 5 }, (err) => {

if (err) {

log.error('Log rotation problems', err);

return reject(err);

}

let args;

// START TESTNET

if (network == 'test') {

args = (nodeType === 'geth')

? ['--testnet', '--fast', '--ipcpath', Settings.rpcIpcPath]

: ['--morden', '--unsafe-transactions'];

}

// START MAINNET

else {

args = (nodeType === 'geth')

? ['--fast', '--cache', '1024']

: ['--unsafe-transactions'];

}

const nodeOptions = Settings.nodeOptions;

if (nodeOptions && nodeOptions.length) {

log.debug('Custom node options', nodeOptions);

args = args.concat(nodeOptions);

}

log.trace('Spawn', binPath, args);

const proc = spawn(binPath, args);

// node has a problem starting

proc.once('error', (err) => {

if (STATES.STARTING === this.state) {

this.state = STATES.ERROR;

log.info('Node startup error');

// TODO: detect this properly

// this.emit('nodeBinaryNotFound');

reject(err);

}

});

// we need to read the buff to prevent node from not working

proc.stderr.pipe(

fs.createWriteStream(Settings.constructUserDataPath('node.log'), { flags: 'a' })

);

// when proc outputs data

proc.stdout.on('data', (data) => {

log.trace('Got stdout data');

this.emit('data', data);

// check for startup errors

if (STATES.STARTING === this.state) {

const dataStr = data.toString().toLowerCase();

if (nodeType === 'geth') {

if (dataStr.indexOf('fatal: error') >= 0) {

const err = new Error(`Geth error: ${dataStr}`);

if (dataStr.indexOf('bind') >= 0) {

err.tag = UNABLE\_TO\_BIND\_PORT\_ERROR;

}

log.debug(err.message);

return reject(err);

}

}

}

});

// when proc outputs data in stderr

proc.stderr.on('data', (data) => {

log.trace('Got stderr data');

this.emit('data', data);

});

this.on('data', \_.bind(this.\_logNodeData, this));

// when data is first received

this.once('data', () => {

/\*

We wait a short while before marking startup as successful

because we may want to parse the initial node output for

errors, etc (see geth port-binding error above)

\*/

setTimeout(() => {

if (STATES.STARTING === this.state) {

log.info(`${NODE\_START\_WAIT\_MS}ms elapsed, assuming node started up successfully`);

resolve(proc);

}

}, NODE\_START\_WAIT\_MS);

});

});

});

}

\_showNodeErrorDialog(nodeType, network) {

let nodelog = this.getLog();

if (nodelog) {

nodelog = `...${nodelog.slice(-1000)}`;

} else {

nodelog = global.i18n.t('mist.errors.nodeStartup');

}

// add node type

nodelog = `Node type: ${nodeType}\n` +

`Network: ${network}\n` +

`Platform: ${process.platform} (Architecure ${process.arch})` + `\n\n${

nodelog}`;

dialog.showMessageBox({

type: 'error',

buttons: ['OK'],

message: global.i18n.t('mist.errors.nodeConnect'),

detail: nodelog,

}, () => {});

}

\_logNodeData(data) {

data = data.toString().replace(/[\r\n]+/, '');

const nodeType = (this.type || 'node').toUpperCase();

log.trace(`${nodeType}: ${data}`);

if (!/^\-\*$/.test(data) && !\_.isEmpty(data)) {

this.emit('nodeLog', data);

}

}

\_loadDefaults() {

log.trace('Load defaults');

this.defaultNodeType = Settings.nodeType || Settings.loadUserData('node') || DEFAULT\_NODE\_TYPE;

this.defaultNetwork = Settings.network || Settings.loadUserData('network') || DEFAULT\_NETWORK;

}

}

const STATES = {

STARTING: 0, /\* Node about to be started \*/

STARTED: 1, /\* Node started \*/

CONNECTED: 2, /\* IPC connected - all ready \*/

STOPPING: 3, /\* Node about to be stopped \*/

STOPPED: 4, /\* Node stopped \*/

ERROR: -1, /\* Unexpected error \*/

};

EthereumNode.STARTING = 0;

module.exports = new EthereumNode();