AIRA Documentation

Alexander Krupenkin

Apr 28, 2020

Contents

CHAPTER 1

Getting Started

AIRA

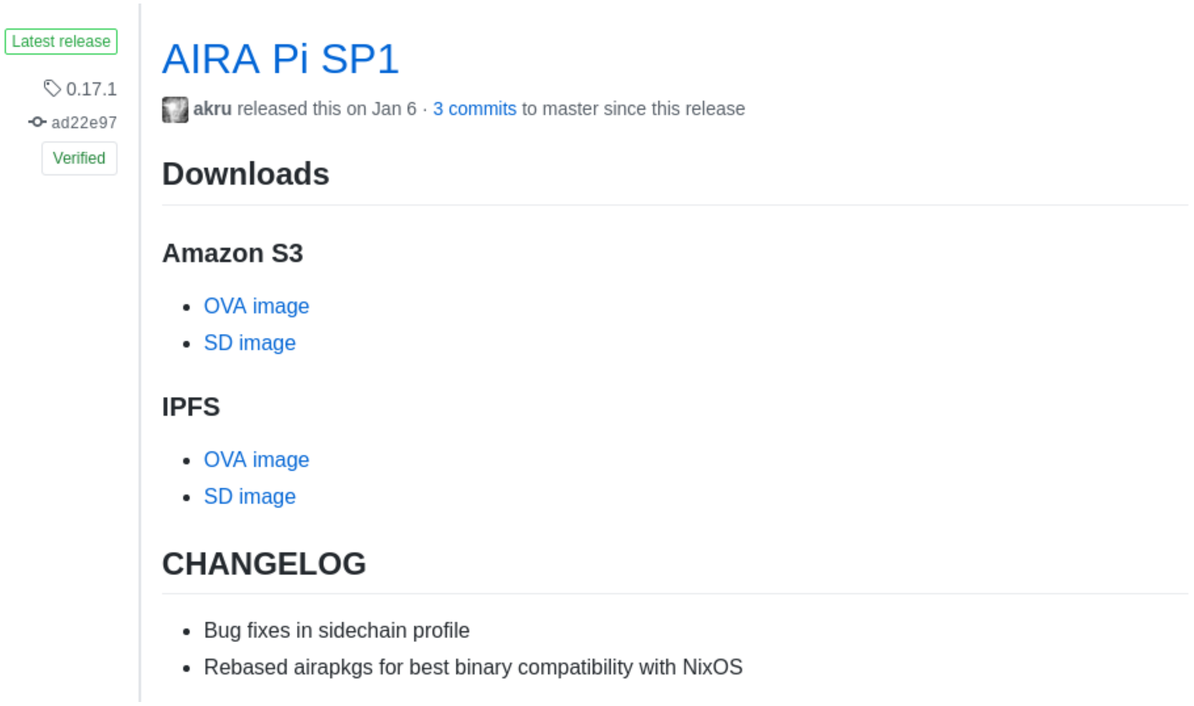
AIRA (Autonomous Intelligent Robot Agent) project implements the standard of economic interaction between human-robot and robot-robot via liability smart contract. AIRA makes it possible to connect a variety of different robots to the market of robot liabilities existing on Ethereum for the direct sale of data from robot sensors, ordering of logistics services, and organization ordering of personalized products at fully automated enterprises.

1.1 Useful links

- AIRA's official site
- The Team
- Robonomics Network

1.2 Quick Start

The first thing to do is to get the last image of AIRA. You can find it here.



AIRA Pi SP1

akru released this on Jan 6 · 3 commits to master since this release

Downloads

Amazon S3

- OVA image
- SD image

IPFS

- OVA image
- SD image

CHANGELOG

- Bug fixes in sidechain profile
- Rebased airapkgs for best binary compatibility with NixOS

Another option is to build the image from the source:

```
$ git clone https://github.com/airalab/airapkgs
$ cd airapkgs
$ nix build -f nixos/release-aira.nix ova_image
```

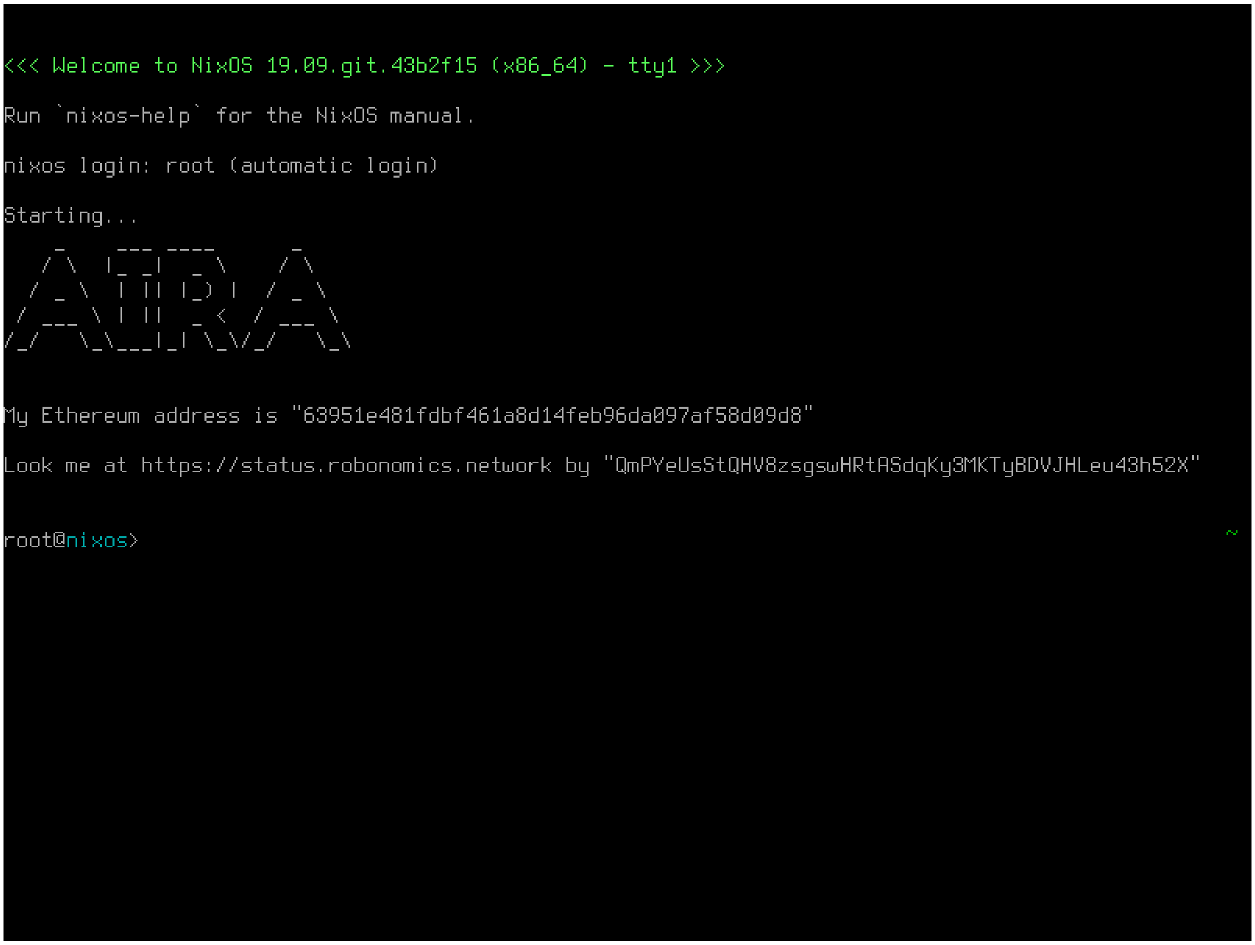
After this the image could be found in the `result` folder.

AIRA is distributed as virtual machine image. To launch the client you need to import .ova file to VirtualBox. You can use a convenient `Ctrl+I` shortcut.

It's recommended to set:

- RAM to 2Gb at least
- At least 40 Gb SSD

When the image is imported, launch the machine.



```
<<< Welcome to NixOS 19.09.git.43b2f15 (x86_64) - tty1 >>>

Run `nixos-help` for the NixOS manual.

nixos login: root (automatic login)

Starting...
    _    ___ ____      _
   / \  |_ _|  _ \    / \
  / _ \  | || |_) |  / _ \
 / ___ \ | ||  _ <  / ___ \
/_/   \_\___|_| \_\/_/   \_\


My Ethereum address is "63951e481fdbf461a8d14feb96da097af58d09d8"

Look me at https://status.robonomics.network by "QmPYeUsStQHV8zsgswHRtASdqKy3MKTyBDVJHLeu43h52X"


root@nixos>                                                                                        ~
```

To make your work with the machine easier, try to connect via SSH.

There are some helpful commands on FAQ page.

CHAPTER 2

Basic Usage

To get familiar with AIRA, let's see what is under the hood.

Once you launch the client several ros nodes will already be on the run. Here's a list of robonomics communication stack nodes:

```
$ rosnode list
/eth/erc20_token
/eth/eth_node
/graph/aira_graph
/liability/executor
/liability/infochan/eth/signer
/liability/infochan/ipfs_channel
/liability/persistence
/liability/listener
/rosout
```

- `/eth/erc20_token`, `/eth/eth_node` - proved services for Ethereum blockchain and ERC20 tokens
- `/graph/aira_graph` - service node for exploring other AIRA instances
- `/liability/executor` - gets rosbag file from IPFS and plays it
- `/liability/infochan/ipfs_channel` - is responsible for offer, demand and result messages. It catches messages from the channel and sends signed messages back
- `/liability/infochan/eth/signer` - offers services for signing offer, demand and result messages
- `/liability/listener` - watches for a new liability contracts. When the event is received the node calls executor node
- `/liability/persistence` - helps to store incoming liabilities and restart them after shutdown

And here's a list of robonomics stack topics.

```
$ rostopic list
/eth/event/approval
/eth/event/transfer
```

(continues on next page)

(continued from previous page)

```
/graph/greetings
/liability/complete
/liability/finalized
/liability/incoming
/liability/infochan/eth/sending/demand
/liability/infochan/eth/sending/offer
/liability/infochan/eth/sending/result
/liability/infochan/eth/signing/demand
/liability/infochan/eth/signing/offer
/liability/infochan/eth/signing/result
/liability/infochan/incoming/demand
/liability/infochan/incoming/offer
/liability/infochan/incoming/result
/liability/persistence/add
/liability/persistence/del
/liability/persistence/update_timestamp
/liability/ready
/liability/result
/rosout
/rosout_agg
```

The most important topics for us are:

- `/liability/incoming` - when a new liability is created, this topic publishes Ethereum address of the contract
- `/liability/result` - this topic is for publishing results. But don't publish a result directly to this topic! Use a service instead
- `/liability/infochan/incoming/*` - a CPS gets information about offer, demand or result from corresponding topics
- `/liability/infochan/eth/signing/*` - a CPS sends offer, demand or result messages to corresponding topics

For the details check out the API page.

Let's start with greetings - say hello to AIRA!

You should just launch a pre-installed package `hello_aira`:

```
$ rosrun hello_aira hello_aira
```

We've launched our agent. It will wait for a demand message. Now it's time to send the message. Go to dapp and press Order. Now go back to the console and see the result!

CHAPTER 3

Connecting via SSH

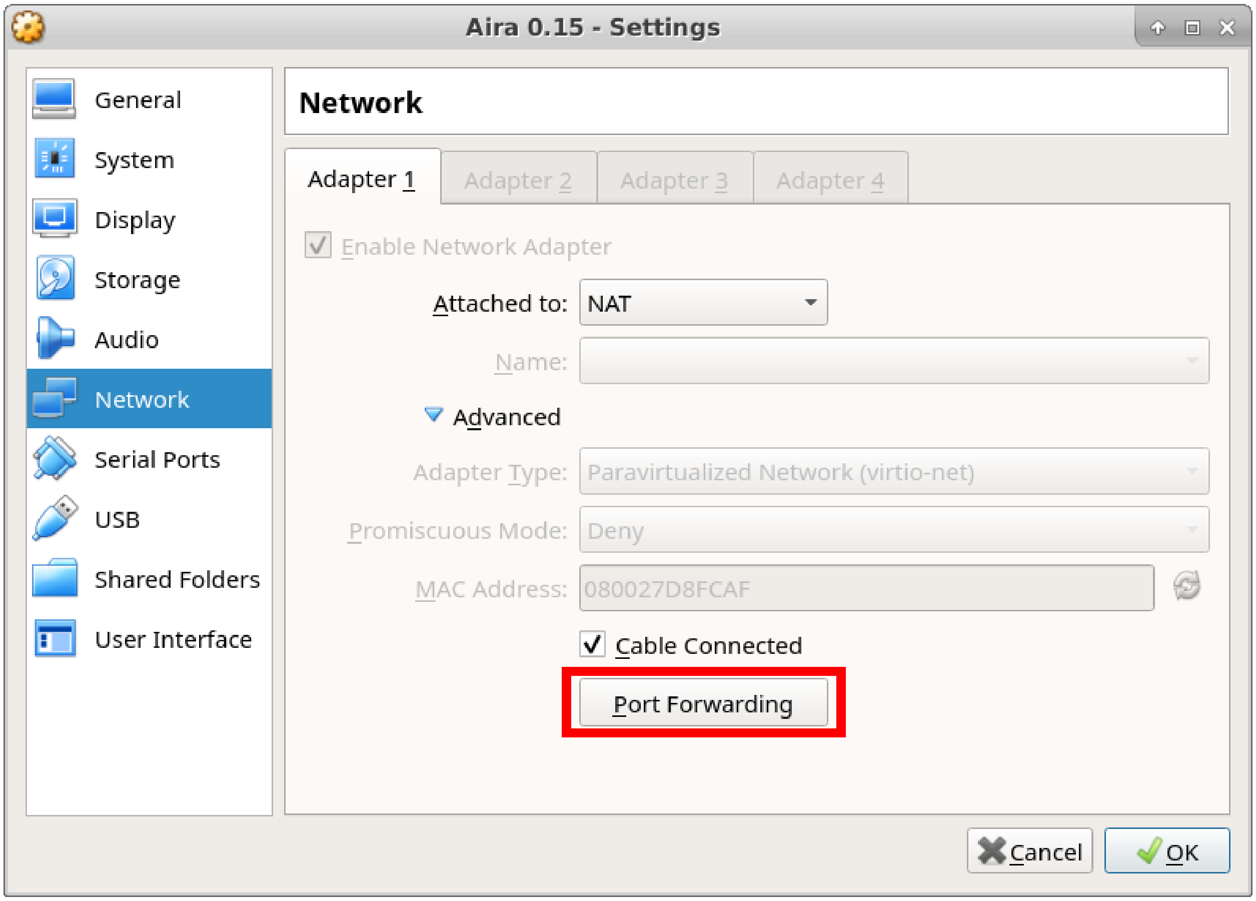
It is more convenient to work with virtual machine via ssh connection. In this section we will configure VM.

Attention: It's required to have your ssh public key on Github.com In case you don't have one, please follow the link

First, launch AIRA client and run a command replacing <username> with your own:

```
$ mkdir .ssh
$ chmod 700 .ssh
$ curl -sSL https://github.com/<username>.keys >> .ssh/authorized_keys
```

Now go to machine settings, network, open Advanced and then Port Forwarding



☑ Enable Network Adapter

Attached to: NAT

Name:

Adapter Type: Paravirtualized Network (virtio-net)

Promiscuous Mode: Deny

MAC Address: 080027D8FCAF

☑ Cable Connected

Add a new rule:

Host IP	Host Port	Guest IP	Guest Port
127.0.1.1	2202	10.0.2.15	22

Reboot the machine and you are able to connect to AIRA client via ssh:

```
$ ssh -p 2202 root@127.0.1.1
```

CHAPTER 4

Frequently Asked Questions

4.1 How to see logs from main services?

IPFS in real time:

```
journalctl -u ipfs -f
```

and Liability:

```
journalctl -u liability -f
```

4.2 How to check the quantity of IPFS peers?

```
ipfs pubsub peers airalab.lighthouse.5.robonomics.eth
```

4.3 IPFS can’t connect to the daemon, what should I do?

Try to specify `--api` option

```
ipfs swarm peers --api=/ip4/127.0.0.1/tcp/5001/
```

CHAPTER 5

Contributing

5.1 Main Airalab repositories

- aira - AIRA client.
- robonomics_comm - Robonomics communication stack
- robonomics_contracts - smart contracts of Robonomics network

Please choose a corresponding repository for reporting an issue!

5.2 Found a bug?

- **Make sure the bug was not already reported** - check GitHub Issues.
- If there is no open issue addressing the problem, open a new one. Be sure to include a **title and clear description**, as much relevant information as possible.

Also, you can open an issue if you have a proposal for improvements.

5.3 Wrote a patch that fixes a bug?

- Open a new GitHub pull request with the patch.
- Make sure the PR description clearly describes the problem and the solution. Include the relevant issue number if applicable.

Please don't fix whitespace, format code, or make a purely cosmetic patch

Thanks!

CHAPTER 6

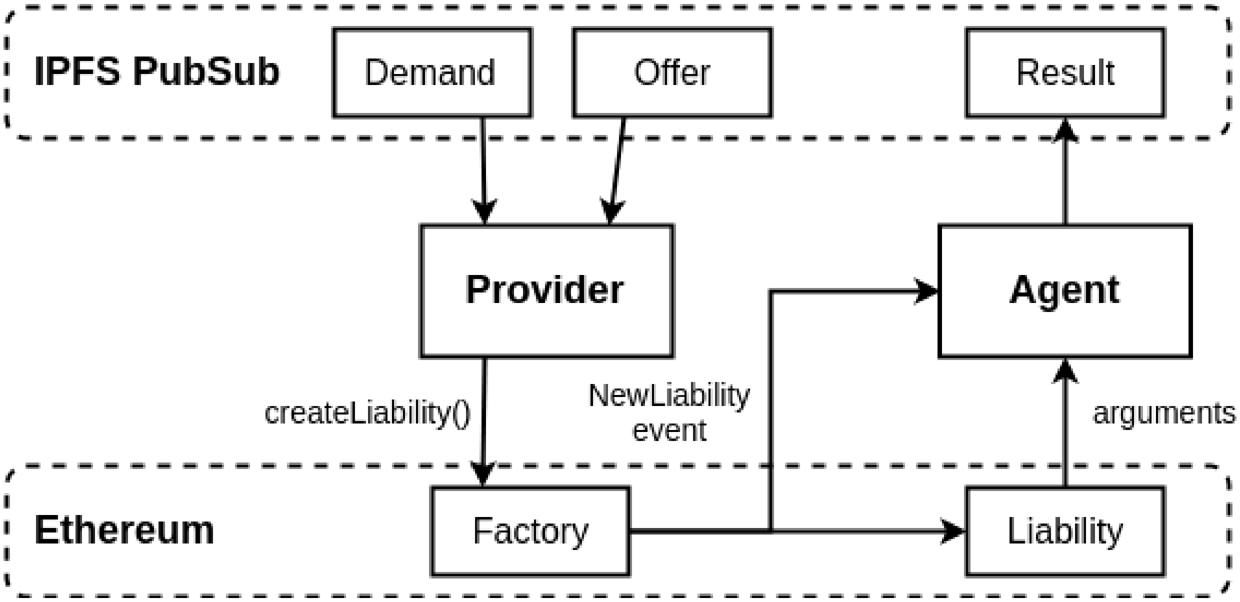
How It Works

In this section we will discuss the Robonomics Network scenario.

There are few main parts in the Robonomics network:

- IPFS for the messages exchanging
- the Ethereum blockchain for storing new liability contracts
- a provider that is responsible for matching messages
- an agent

Let's have a look at the following diagram that describes the scenario without any additional details:



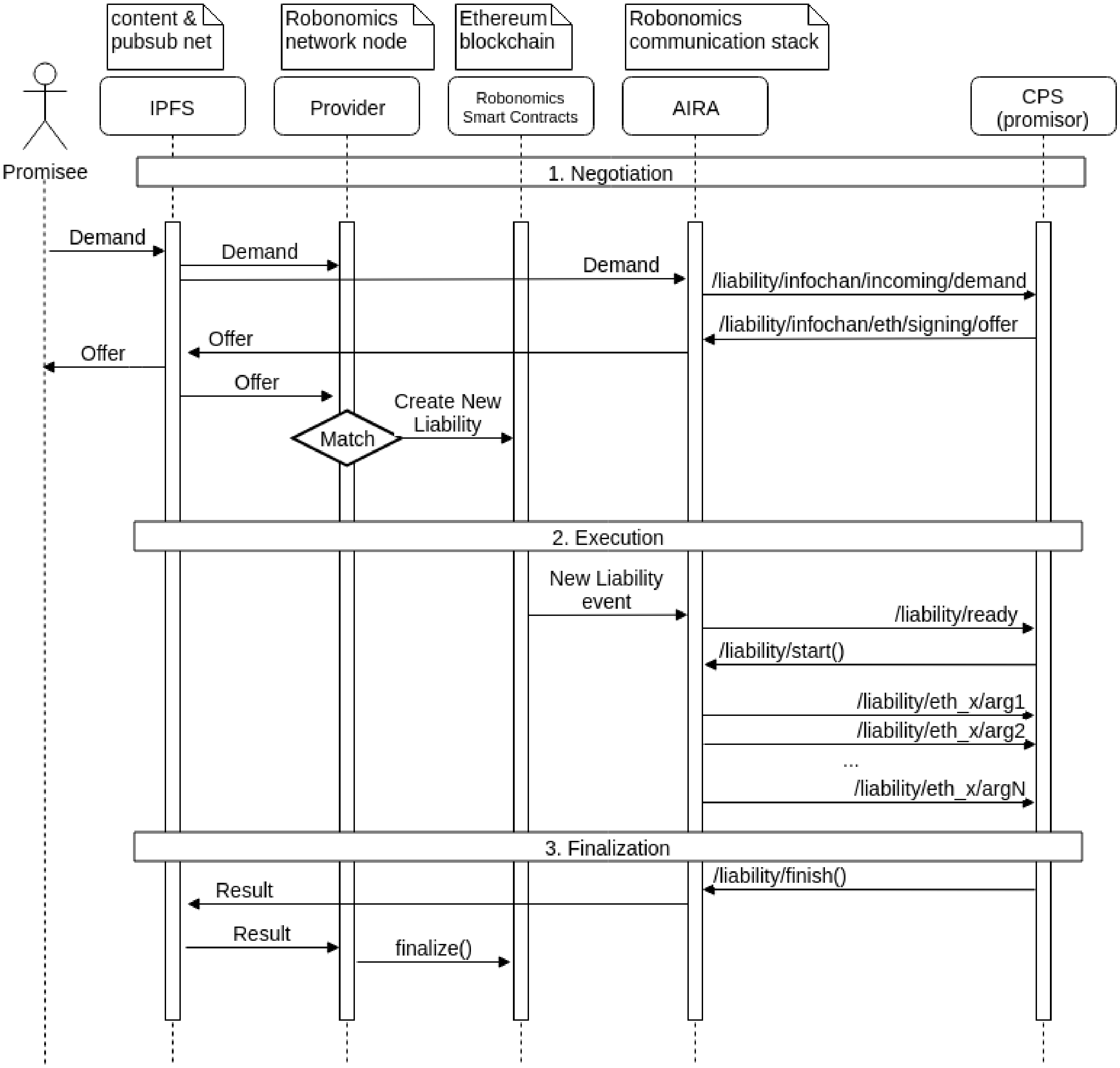
There are three types of messages in IPFS: Demand, Offer, Result.

Below there is the specification for a Demand message:

Field	Type	Description	Example
model	*ipfs_common/Multihash*	CPS behavioral model Identifier	QmfXHZ2YkNC5vRjp1oAaRoDHD8H3zZznfhBPasTu348eWC
objective	*ipfs_common/Multihash*	CPS behavioral model parameters in rosbag file	QmUo3vvSXZPQaQWjb3cH3qQo1hc8vAUqNnqbdVABbSLb6
token	*ethereum_common/Address*	Operational token address	0xbD949595eE52346c225a19724084cE517B2cB735
cost	*ethereum_common/UInt256*	CPS behavioral model implementation cost	1
lighthouse	*ethereum_common/Address*	Lighthouse address	0xa1b60ED40E5A68184b3ce4f7bEf31521A57eD2dB1
validator	*ethereum_common/Address*	Observing network address	0x0000000000000000000000000000000000000000
validatorFee	*ethereum_common/UInt256*	Observing network commission	0
deadline	*ethereum_common/UInt256*	Deadline block number	6393332
sender	*ethereum_common/Address*	Message sender address	0x0000000000000000000000000000000000000000
signature	std_msgs/UInt8[]	Sender's digital signature	0x23bc…c617

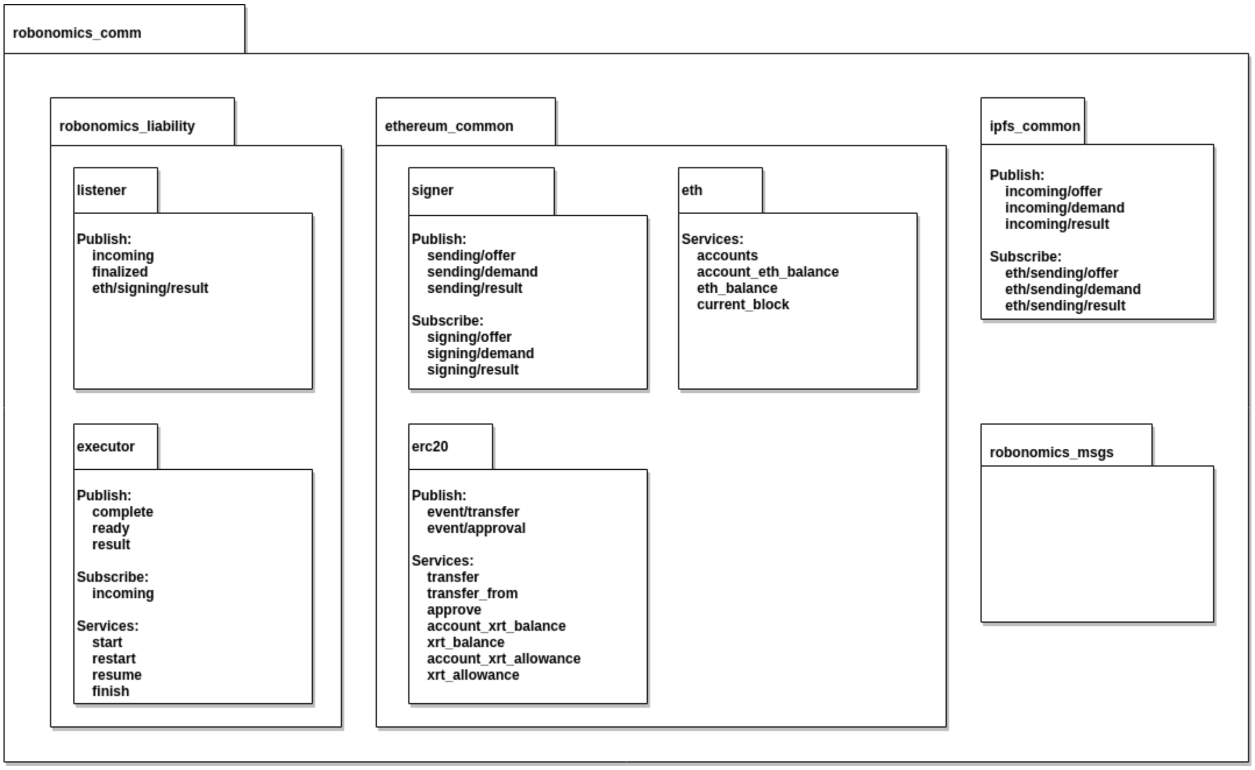
An Offer message has the same fields but instead of `validatorFee` there is a `lighthouseFee` field. This field determines the amount of fee for a lighthouse.

Now let's have a look at the following diagram and walk step by step from the moment of publishing messages to a liability finalization.



A liability contract is created only if the following fields match: `model`, `objective`, `token`, `cost`. A provider of Robonomics Network watches every message and finds those ones that have a match. After the match is found the provider calls `createLiability(demand, offer)` method from the contract factory where `demand` and `offer` are serialized.

Below is the package diagram for the Robonomics communication stack:



The factory deserializes arguments and recovers *promisee* and *promisor* addresses from signatures.

Next step is token transfer. The factory transfers **cost** tokens from the *promisee* address and **validatorFee** and **lighthouseFee** from the *promisor* address to the new liability address.

Note: You should approve sufficient amount of tokens for the factory.

Note: It's not required to approve tokens from the *promisor* address if fees are null.

Now the factory emits a NewLiability event with the liability address. An agent gets the address, reads fields, perform a task and at the same time writes a log file in rosbag format.

When the work is done the agent sends a Result message with the following fields: hash of the rosbag file, a success flag, a signature. If the **validator** field is not null it means that only validator is able to finalize the liability.

After the successful liability finalization the agent gets **cost** tokens. Otherwise, the *promisee* gets tokens back.

CHAPTER 7

Contracts Deployment

Robonomics network works on top of the existing Ethereum network. The protocol is implemented by smart contracts. A source code is on Github. Airalab team deploys new version of contracts and supports a current one.

In this lesson we are going to learn more about these contracts. To do this we will deploy our test copy. Also we are going to use these contracts in the future lessons.

You need a client running Ethereum node. You can use either one of existing network (e.g. Mainnet, Ropsten, Kovan) or your local one. For testing purpose we suggest to use this docker container

```
$ docker run --rm -d -p 9545:8545 -p 9546:8546 foamspace/cliquebait:latest
```

Next step is obtain a copy of robonomics contracts source code:

```
$ git clone --recursive https://github.com/airalab/robonomics_contracts
```

A file truffle.js contains available networks for migration. We will work with development network. When you are in `robonomics_contracts` directory install dependencies and run a migration:

```
npm install // to install dependencies
truffle migrate --network development
```

It's time to learn how to create a new lighthouse. For more information about Robonomics network and Lighthouse in particular read white paper. Briefly lighthouse o distributes the running time of providers. Every lighthouse serves its own broadcast channel. Ask and Bid messages come into this channel. XRT tokens are used as a payment.

When XRT contracts was deployed some tokens were issued on our account. Let's check the balance:

```
$ truffle --network development console
> xrt = XRT.at(XRT.address)
> xrt.balanceOf(web3.eth.accounts[0])
```

And that's how we create a lighthouse:

```
> factory = LiabilityFactory.at(LiabilityFactory.address)
> tx = factory.createLighthouse(1000, 10, "test")
> tx.then(x => {laddress = x.logs[0].args.lighthouse})
> l = LighthouseLib.at(laddress)
```

Instead of deploying a lighthouse contract every time we need a new one, we ask a factory to do this job. A `l` variable contains lighthouse instance. The lighthouse should be able to spend our tokens. Let's make an approve and check everything went well:

```
> xrt.approve(l.address,1000)
> xrt.allowance(web3.eth.accounts[0],l.address)
```

And a very important step is become a worker:

```
> l.refill(1000)
```

Each worker has to put a stake. In this case it's 1000 Wn.

Below is a table of our addresses:

Contract	Address	ENS name
ENSRegistry	0x80c77a7de64a15450bb8cf45ece4fbb7bae6fb49	
XRT	0x673583a369eb3a830a5571208cf6eb7ce83987f8	xrt.3.robonomics.eth
LiabilityFactory	0x1b3190e00c1903266862af1f31714d4b81ef59b2	factory.3.robonomics.eth
Lighthouse	0xd2b78c032b6c8851a8b6cbf950caa02a77618d8e	test.lighthouse.3.robonomics.eth

CHAPTER 8

Become a Provider

This page describes how to create a lighthouse and become a provider in the Robonomics network.

8.1 Prepare an address

First of all, an Ethereum address is required. You must have access to a private key of the address. In case you don't have one, below are steps to create an address via Parity.

```
$ sudo snap install parity
$ parity.ethkey generate random
secret:  15abe71557c07b69537bbe4352ed10a057be89037c69d4b35556112519911539
public:␣
↪38b800bfd90d486c78c646da79bb94b9d038aca8aad221062ce1b148df7764bfef02f6b3cf931786b6997540b798ea226ae
address: c531fa8f141493df3da264a864bdcbec19695b4c
```

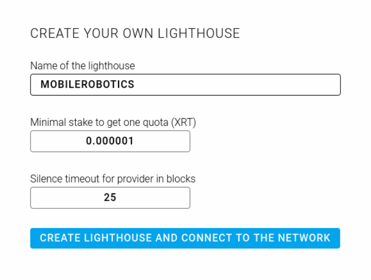
The `secret` field is a private key, you'll need it to run the provider client. Save it to a file:

```
$ echo '0x15abe71557c07b69537bbe4352ed10a057be89037c69d4b35556112519911539' > private.
↪key
```

The next step is to deposit some ethers and XRT tokens to the address which is held in the `address` field.

8.2 Create a lighthouse

Go to the lighthouse dapp and fill in a name in the right side:



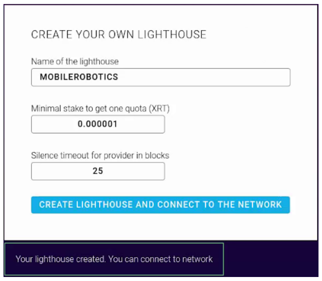
CREATE YOUR OWN LIGHTHOUSE

Name of the lighthouse: MOBILEROBOTICS

Minimal stake to get one quota (XRT): 0.000001

Silence timeout for provider in blocks: 25

Click on the *Create lighthouse and connect to the network* button and sign a transaction. After a while you should see:



CREATE YOUR OWN LIGHTHOUSE

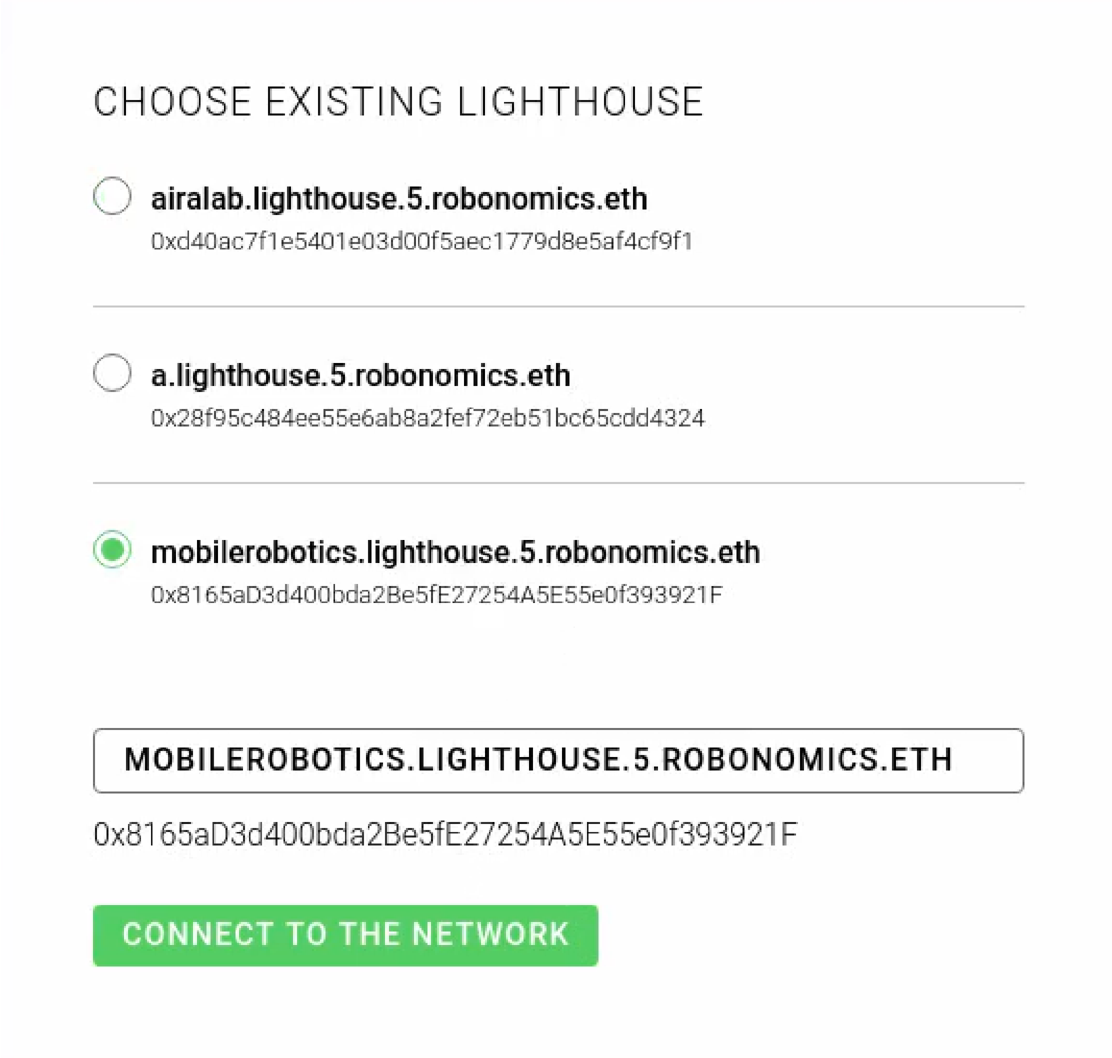
Name of the lighthouse: MOBILEROBOTICS

Minimal stake to get one quota (XRT): 0.000001

Silence timeout for provider in blocks: 25

Your lighthouse created. You can connect to network

Now it's time to put a stake. Select the new lighthouse and click *Connect to the network*:



CHOOSE EXISTING LIGHTHOUSE

○ **airalab.lighthouse.5.robonomics.eth**
0xd40ac7f1e5401e03d00f5aec1779d8e5af4cf9f1

○ **a.lighthouse.5.robonomics.eth**
0x28f95c484ee55e6ab8a2fef72eb51bc65cdd4324

● **mobilerobotics.lighthouse.5.robonomics.eth**
0x8165aD3d400bda2Be5fE27254A5E55e0f393921F

MOBILEROBOTICS.LIGHTHOUSE.5.ROBONOMICS.ETH

0x8165aD3d400bda2Be5fE27254A5E55e0f393921F

On this page in the *Provider* section click the *Approve* button, sign a transaction. When it's mined click the *Refill* button and do the same.

8.3 Install the client

You must install robonomics-tools at least 0.4.2. version. You can build from the source or do the following steps:

Note: Make sure you have Nix and Stack installed:

```
$ curl -sSL https://get.haskellstack.org/ | sh
$ curl https://nixos.org/nix/install | sh
```

- Setup Airalab binary cache at https://aira.cachix.org/
- Import Airalab channel:

  ```
  $ nix-channel --add http://aira.life/channels/aira-unstable/ aira
  $ nix-channel --update
  ```

- Install from the binary cache:

```
$ nix-env -iA aira.robonomics-tools
```

- Run the client:

```
$ xrtd --lighthouse mobilerobotics.lighthouse.5.robonomics.eth --private
↪$(cat private.key)
```

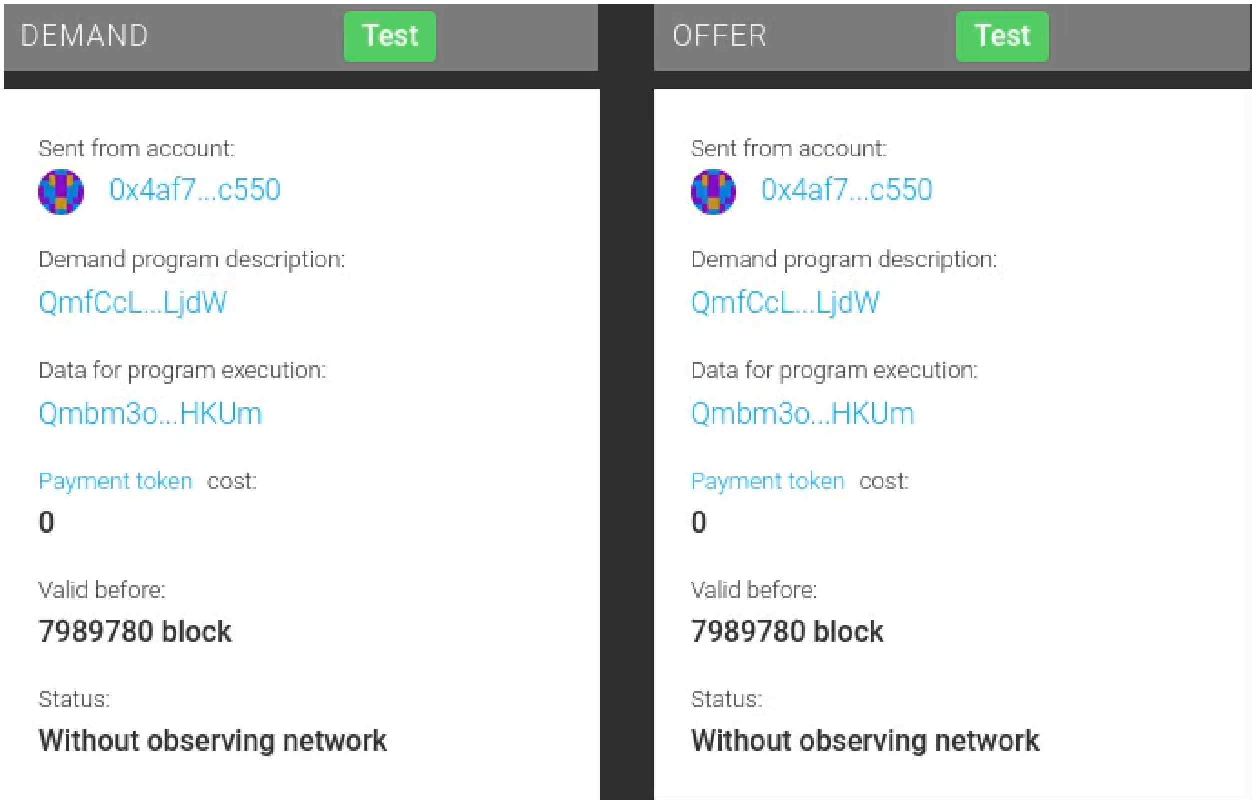
Hint: Get familiar with the *xrtd* options via `xrtd --help`

8.4 Test the provider

To test your provider go again to the lighthouse dapp and connect to the just created lighthouse.

At the bottom you should see the *TEST LIGHTHOUSE* section.

Click on the *Demand* button and then on the *Offer* one. You should see something similar to:



Sent from account:
0x4af7...c550

Demand program description:
QmfCcL...LjdW

Data for program execution:
Qmbm3o...HKUm

Payment token cost:
0

Valid before:
7989780 block

Status:
Without observing network

Sent from account:
0x4af7...c550

Demand program description:
QmfCcL...LjdW

Data for program execution:
Qmbm3o...HKUm

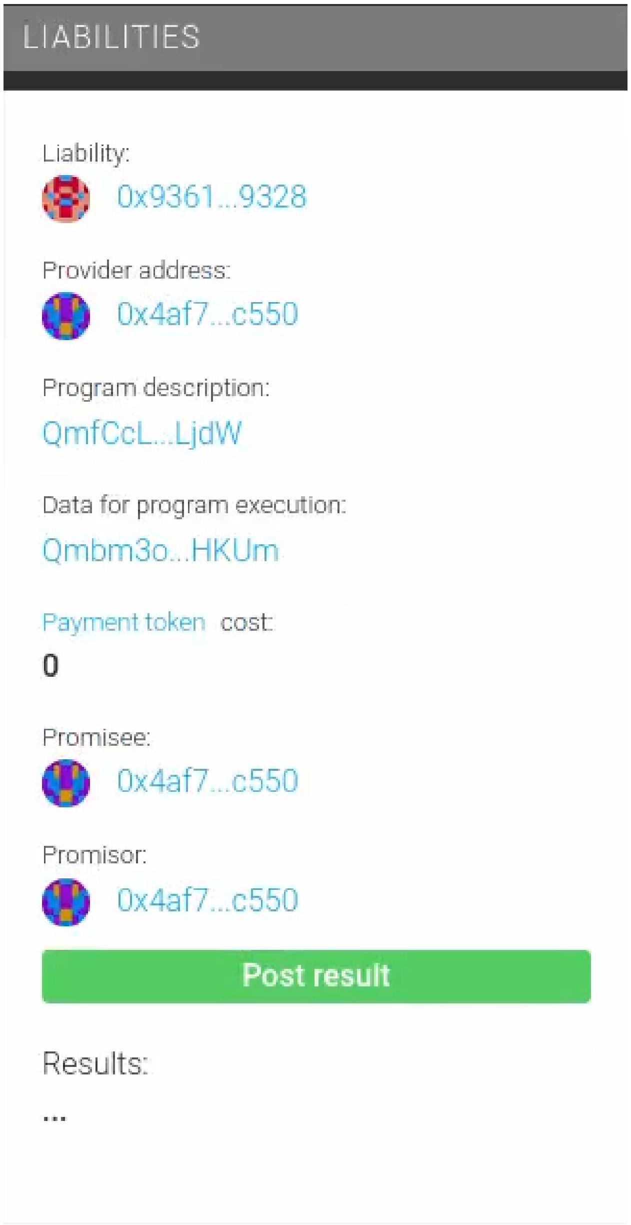
Payment token cost:
0

Valid before:
7989780 block

Status:
Without observing network

Don't forget to sign every message with the MetaMask extension.

Finally you should see a new liability contract created:



LIABILITIES

Liability:
0x9361...9328

Provider address:
0x4af7...c550

Program description:
QmfCcL...LjdW

Data for program execution:
Qmbm3o...HKUm

Payment token cost:
0

Promisee:
0x4af7...c550

Promisor:
0x4af7...c550

Results:
...

CHAPTER 9

Market Messages

Market messages is used for exchange **Demand** and **Offer** information. It also used for delivery **Result** messages with liability execution reports.

Note: This is spec for Robonomics `Generation 5`.

- Currently for message delivery is used IPFS PubSub broadcaster.
- IPFS PubSub **topic** is set according to *Lighthouse* ENS name.

9.1 Messages content

Robonomics market message use JSON data format.

Demand

Field	ROS Type	Description
model	*ipfs_common/Multihash*	CPS behavioral model identifier
objective	*ipfs_common/Multihash*	CPS behavioral model parameters in rosbag file
token	*ethereum_common/Address*	Operational token address
cost	*ethereum_common/UInt256*	CPS behavioral model execution cost
lighthouse	*ethereum_common/Address*	Lighthouse contract address
validator	*ethereum_common/Address*	Observing network address
validatorFee	*ethereum_common/UInt256*	Observing network fee
deadline	*ethereum_common/UInt256*	Deadline block number
nonce	*ethereum_common/UInt256*	Robonomics message counter
sender	*ethereum_common/Address*	Message sender address
signature	std_msgs/UInt8[]	Sender's Ethereum signature

Offer

Field	ROS Type	Description
model	*ipfs_common/Multihash*	CPS behavioral model identifier
objective	*ipfs_common/Multihash*	CPS behavioral model parameters in rosbag file
token	*ethereum_common/Address*	Operational token address
cost	*ethereum_common/UInt256*	CPS behavioral model execution cost
validator	*ethereum_common/Address*	Observing network address
lighthouse	*ethereum_common/Address*	Lighthouse contract address
lighthouseFee	*ethereum_common/UInt256*	Liability creation fee
deadline	*ethereum_common/UInt256*	Deadline block number
nonce	*ethereum_common/UInt256*	Robonomics message counter
sender	*ethereum_common/Address*	Message sender address
signature	std_msgs/UInt8[]	Sender's Ethereum signature

Result

Field	ROS Type	Description
liability	*ethereum_common/Address*	Liability contract address
result	*ipfs_common/Multihash*	Liability result multihash
success	std_msgs/Bool	Is liability executed successful
signature	std_msgs/UInt8[]	Sender's Ethereum signature

9.2 Messages signing

Before signing the messages is packed using abi.encodePacked solidity finction and hashed by Keccak_256.

```
demandHash = keccak256(abi.encodePacked(
     _model
   , _objective
   , _token
   , _cost
   , _lighthouse
   , _validator
   , _validator_fee
   , _deadline
   , IFactory(factory).nonceOf(_sender)
   , _sender
   ));
```

Note: `nonce` parameter is counted by factory smart contract and incremented for each created liability smart contract.

Message hash are signed using Ethereum `secp256k1` signature.

CHAPTER 10

Robonomics Liability

The package is responsible for receiving *New Liability* events (`listener` node) and playing topics from *objective* field (`executor` node). The launch file also include `ipfs_channel` node and `signer` node.

10.1 ROS Parameters

~web3_http_provider
Web3 HTTP provider address. The type is `string`, defaults to `http://127.0.0.1:8545`

~web3_ws_provider
Web3 WebSocket provider address. The type is `string`, defaults to `ws://127.0.0.1:8546`

~ipfs_http_provider
IPFS HTTP provider address. The type is `string`, defaults to `http://127.0.0.1:5001`

~factory_contract
The name of the liability factory. The type is `string`, defaults to `factory.3.robonomics.eth`

~lighthouse_contract
The name of a lighthouse you are working on. The type is `string`, defaults to `airalab.lighthouse.3.robonomics.eth`

~enable_executor
Enable or disable executor node. If it's `false`, no topics from objective would be published. The type is `boolean`, defaults to `true`

~master_check_interval
Period (in seconds) to check master for new topic publications. It's necessary for the Recorder, which records all the topics a CPS publishes. The type is `double`, defaults to `0.1`

~recording_topics
List of topics name separated by comma. It allows you to specify which topics would be recorded. The type is `string`, defaults to `""`

~ens_contract
The checksumed address of ENS registry. The type is `string`, defaults to `""`

`~keyfile`
Path to keyfile. The type is `string`, defaults to `""`. **Required parameter**

`~keyfile_password_file`
Path to a file with password for the keyfile. The type is `string`, defaults to `""`. **Required parameter**

10.2 Subscribed topics

`/liability/infochan/eth/signing/demand (robonomics_msgs/Demand)`
robonomics_msgs/Demand message to sign and send further to IPFS channel

`/liability/infochan/eth/signing/offer (robonomics_msgs/Offer)`
robonomics_msgs/Offer message to sign and send further to IPFS channel

`/liability/infochan/eth/signing/result (robonomics_msgs/Result)`
robonomics_msgs/Result message to sign and send further to IPFS channel

10.3 Published topics

`/liability/infochan/incoming/demand (robonomics_msgs/Demand)`
Contains a robonomics_msgs/Demand message which was read from IPFS channel

`/liability/infochan/incoming/offer (robonomics_msgs/Offer)`
Contains a robonomics_msgs/Offer message which was read from IPFS channel

`/liability/infochan/incoming/result (robonomics_msgs/Result)`
Contains a robonomics_msgs/Result message which was read from IPFS channel

`/liability/incoming (robonomics_liability/Liability)`
Contains all the information about the last created robonomics_liability/Liability

`/liability/ready (robonomics_liability/Liability)`
Signals when a robonomics_liability/Liability is ready for execution

`/liability/complete (robonomics_liability/Liability)`
Signals when a robonomics_liability/Liability has done its job

`/liability/finalized (std_msgs/String)`
Signals when a liability has been finalized

10.4 Services

`/liability/start (robonomics_liability/StartLiability)`
The service tells executor to play topics from the objective. It’s required to pass a liability address (robonomics_liability/StartLiability), which you can get from `/liability/ready` topic

`/liability/finish (robonomics_liability/FinishLiability)`
a CPS should call the service after performing the task. The input is robonomics_liability/FinishLiability

`/liability/restart (robonomics_liability/StartLiability)`
The service allows to restart a liability after the system shutdown. The input is robonomics_liability/StartLiability

/liability/resume (robonomics_liability/StartLiability)

The service allows to resume a liability from the last timestamp available in the persistence store. The input is robonomics_liability/StartLiability

CHAPTER 11

Robonomics Liability Messages

11.1 robonomics_liability/Liability.msg

Field	Type	Description
address	*ethereum_common/Address*	The Liability's address
model	*ipfs_common/Multihash*	CPS behavioral model Identifier
objective	*ipfs_common/Multihash*	CPS behavioral model parameters in rosbag file
result	*ipfs_common/Multihash*	Liability result hash
promisee	*ethereum_common/Address*	The promisee address
promisor	*ethereum_common/Address*	The promisor address (usually CPS)
lighthouse	*ethereum_common/Address*	The address of lighthouse your CPS works on
token	*ethereum_common/Address*	Operational token address
cost	*ethereum_common/UInt256*	CPS behavioral model implementation cost
validator	*ethereum_common/Address*	Observing network address
validatorFee	*ethereum_common/UInt256*	Observing network commission

11.2 ipfs_common/Multihash.msg

Field	Type	Description
multihash	std_msgs/String	A wrapper for model and objective fields

11.3 robonomics_liability/StartLiability.srv

Request

Field	Type	Description
address	std_msgs/String	The address of Liability you are willing to execute

Response

Field	Type	Description
success	std_msgs/Bool	Weather or not the Liability was started
msg	std_msgs/String	Status of launch

11.4 robonomics_liability/FinishLiability.srv

Request

Field	Type	Description
address	std_msgs/String	The address of Liability to finish
success	std_msgs/Bool	The status of execution

Response

The response is empty

CHAPTER 12

Ethereum Common

The packages contains two launch files: `erc20.launch` and `signer.launch`. The last one is included in Robonomics Liability.

Below is the description for `erc20` node which contains utils for convenient work with Ethereum accounts and XRT token.

12.1 ROS Parameters

`~web3_http_provider`
Web3 HTTP provider address. The type is `string`, defaults to `http://127.0.0.1:8545`

`~erc20_token`
ERC20 token to work with. Type is `string`, defaults to `xrt.3.robonomics.eth`

`~factory_contract`
The name of the liability factory. The type is `string`, defaults to `factory.3.robonomics.eth`

`~ens_contract`
The checksumed address of ENS registry. The type is `string`, defaults to `""`

`~keyfile`
Path to keyfile. The type is `string`, defaults to `""`. **Required parameter**

`~keyfile_password_file`
Path to a file with password for the keyfile. The type is `string`, defaults to `""`. **Required parameter**

12.2 Published topics

`/eth/event/transfer (ethereum_common/TransferEvent)`
The event *ethereum_common/TransferEvent* is emitted after the transfer of tokens was made

`/eth/event/approval (ethereum_common/ApprovalEvent)`
The event *ethereum_common/ApprovalEvent* is emitted after the approval of tokens was made

12.3 Services

`/eth/accounts (ethereum_common/Accounts)`
List of available Ethereum accounts. See *ethereum_common/Accounts.srv*

`/eth/account_eth_balance (ethereum_common/AccountBalance)`
Returns the balance of the given address in Wei. See *ethereum_common/AccountBalance.srv*

`/eth/eth_balance (ethereum_common/Balance)`
Returns the balance of the default address. See *ethereum_common/Balance.srv*

`/eth/current_block (ethereum_common/BlockNumber)`
Returns current block number. See *ethereum_common/BlockNumber.srv*

`/eth/transfer (ethereum_common/Transfer)`
Transfers tokens from the default account to a given one. See *ethereum_common/Transfer.srv*

`/eth/transfer_from (ethereum_common/TransferFrom)`
Transfers tokens from a given account to another one. See *ethereum_common/TransferFrom.srv*

`/eth/approve (ethereum_common/Approve)`
Approves tokens from the default account to a given one. See *ethereum_common/Approve.srv*

`/eth/account_xrt_balance (ethereum_common/AccountBalance)`
Returns the XRT balance of a given account. See *ethereum_common/AccountBalance.srv*

`/eth/xrt_balance (ethereum_common/Balance)`
Return the XRT balance of the default account. See *ethereum_common/Balance.srv*

`/eth/account_xrt_allowance (ethereum_common/AccountToAddressAllowance)`
Returns how much one account is allowed to spend from another account. See *ethereum_common/AccountToAddressAllowance.srv*

`/eth/xrt_allowance (ethereum_common/Allowance)`
Returns how much the Factory is allowed to spend from the default account. See *ethereum_common/Allowance.srv*

CHAPTER 13

Ethereum Common Messages

13.1 ethereum_common/Address.msg

Field	Type	Description
address	std_msgs/String	Address in Ethereum blockchain

13.2 ethereum_common/UInt256.msg

Field	Type	Description
uint256	std_msgs/String	A wrapper for big integers

13.3 ethereum_common/TransferEvent.msg

Field	Type	Description
args_from	*ethereum_common/Address*	Sender address
args_to	*ethereum_common/Address*	Receiver address
args_value	*ethereum_common/UInt256*	Amount of tokens

13.4 ethereum_common/ApprovalEvent.msg

Field	Type	Description
args_owner	*ethereum_common/Address*	Owner address
args_spender	*ethereum_common/Address*	Spender address
args_value	*ethereum_common/UInt256*	Amount of tokens

13.5 ethereum_common/AccountBalance.srv

Request

Field	Type	Description
account	*ethereum_common/Address*	Ethereum address

Response

Field	Type	Description
balance	*ethereum_common/UInt256*	Balance in Wei

13.6 ethereum_common/AccountToAddressAllowance.srv

Request

Field	Type	Description
account	*ethereum_common/Address*	Ethereum address
to	*ethereum_common/Address*	Ethereum address

Response

Field	Type	Description
amount	*ethereum_common/UInt256*	Balance in Wn

13.7 ethereum_common/Accounts.srv

Request

Request is empty

Response

Field	Type	Description
accounts	*ethereum_common/Address[]*	List of available accounts

13.8 ethereum_common/Allowance.srv

Request

Request is empty

Response

Field	Type	Description
amount	*ethereum_common/UInt256*	Amount of XRT the Factory is allowed to spend

13.9 ethereum_common/Approve.srv

Request

Field	Type	Description
spender	*ethereum_common/Address*	Who is allowed to spend
value	*ethereum_common/UInt256*	How much tokens are allowed

Response

Field	Type	Description
txhash	std_msgs/Uint8[32]	Transaction hash

13.10 ethereum_common/Balance.srv

Request

Request is empty

Response

Field	Type	Description
balance	*ethereum_common/UInt256*	The balance of default account

13.11 ethereum_common/BlockNumber.srv

Request

Request is empty

Response

Field	Type	Description
number	std_msgs/Uint64	Current block number

13.12 ethereum_common/Transfer.srv

Request

Field	Type	Description
to	*ethereum_common/Address*	Ethereum address
value	*ethereum_common/UInt256*	The amount of tokens

Response

Field	Type	Description
txhash	std_msgs/Uint8[32]	Transaction hash

13.13 ethereum_common/TransferFrom.srv

Request

Field	Type	Description
owner	*ethereum_common/Address*	Owner's address
to	*ethereum_common/Address*	Another account
value	*ethereum_common/UInt256*	The amount of tokens

Response

Field	Type	Description
txhash	std_msgs/Uint8[32]	Transaction hash

CHAPTER 14

Connect the Simplest CPS

In this section we will build the simplest real cyber-physical system!

We will buy a “wink” from Arduino, e.g. make Arduino blink with its onboard led. The lesson is tested on Arduino Uno, but any other board with a led will do the job.

Note: The source code of this lesson is here.

14.1 Arduino

The firmware for the board is located in arduino_blink/misc/arduino/arduino.ino. Use Arduino IDE to load the code to your Arduino board.

In the code we subscribe for the `/blink_led` topic and set a callback. The type of the topic is `Empty`, so the board waits until someone publishes to the topic and performs the LED blinking.

```
#include <ros.h>
#include <std_msgs/Empty.h>

ros::NodeHandle  nh;

void blink(int led, int mil) {

  digitalWrite(led, HIGH);
  delay(mil);
  digitalWrite(led, LOW);
  delay(mil);

}

void messageCb( const std_msgs::Empty& toggle_msg){
  blink(LED_BUILTIN, 500);
```

(continues on next page)

(continued from previous page)

```
  blink(LED_BUILTIN, 500);
  blink(LED_BUILTIN, 500);
}

ros::Subscriber<std_msgs::Empty> sub("blink_led", &messageCb );

void setup()
{
  pinMode(LED_BUILTIN, OUTPUT);
  nh.initNode();
  nh.subscribe(sub);
}

void loop()
{
  nh.spinOnce();
  delay(1);
}
```

14.2 AIRA client

Note: You can download the latest release from here

Set up the COM port forwarding as described in this lesson. You should forward your /dev/ttyUSB0 or /dev/ttyACM0 port (depending on the system) to COM1. In the client /dev/ttyS0 will represent the board. After this launch the virtual machine.

14.3 ROS

When new liability is created it goes to /liability/ready topic. We have to remember the address and call /liability/start service to get the data from objective.

```
def newliability(l):
  self.liability = l.address
  rospy.loginfo("Got new liability {}".format(self.liability))

  prefix = "/liability/eth_" + self.liability
  rospy.Subscriber(prefix + '/blink', Empty, self.blink)

  rospy.wait_for_service("/liability/start")
  rospy.ServiceProxy('/liability/start',␣
↪StartLiability)(StartLiabilityRequest(address=self.liability))
rospy.Subscriber("/liability/ready", Liability, newliability)
```

A message in the /blink topic come from the objective field. Have a look at Basic usage page.

14.4 AIRA

Connect to AIRA client via SSH as described here. All tutorials are pre-installed. To launch the ros package run the following command:

```
$ rosrun arduino_blink blink.py
```

Also we need to add a rosbag file to IPFS:

```
$ ipfs add rosbag/blink.bag
```

Note: Before the next step you should approve XRT tokens on the Factory.

On your host system build and launch an Dapp for the lesson:

```
$ git clone https://github.com/airalab/robonomics_tutorials/
$ cd robonomics_tutorials/arduino_blink_dapp
$ npm i && npm run dev
```

Open the link and press Demand then Offer buttons. Wait until a new liability is created and you should see the board blinking. Congratulations on the first agent!

CHAPTER 15

Passing Dynamic Parameters

In previous example we discussed how to create a simple CPS with Arduino. Our first CPS is able to do only one task: to blink a led. We suggest you to get through the first lesson. Now let's expand the example and teach our board to blink blue or red led depending on objective parameter.

Note: The source code of this lesson is here.

15.1 Arduino

The only difference in Arduino source code is instead of subscribing to one topic now we subscribe to `/blink_red` and `/blink_blue` topics

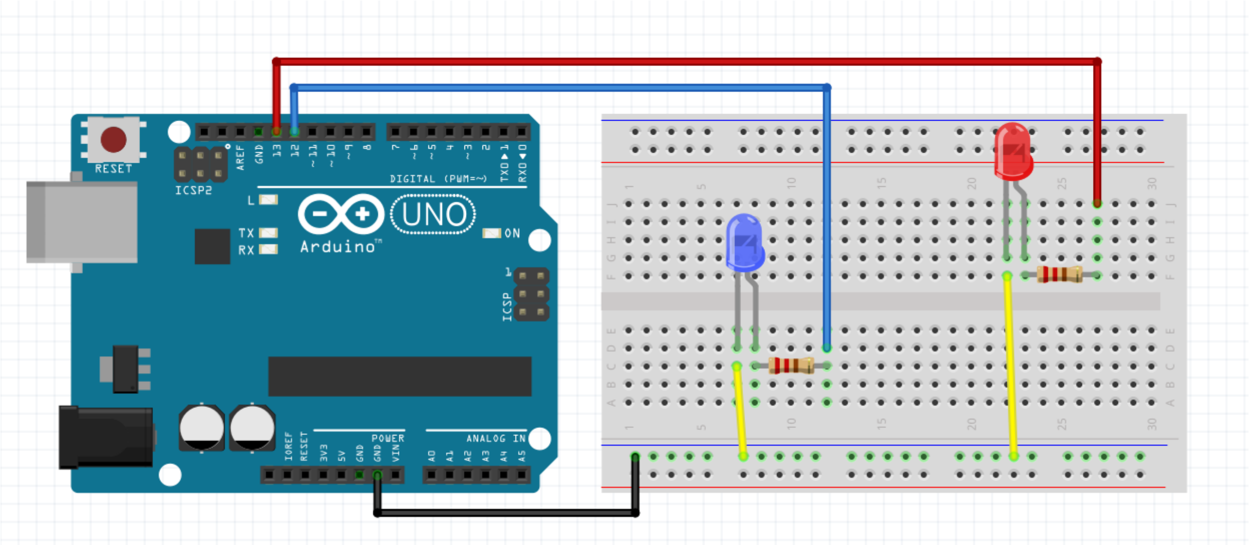
```
#include <ros.h>
#include <std_msgs/Empty.h>

ros::NodeHandle nh;

void blink(int led, int mil) {

  digitalWrite(led, HIGH);
  delay(mil);
  digitalWrite(led, LOW);
  delay(mil);

}

void blinkRedCb(const std_msgs::Empty& msg) {
  blink(13, 500);
  blink(13, 500);
  blink(13, 500);
}
```

(continues on next page)

(continued from previous page)

```
void blinkBlueCb(const std_msgs::Empty& msg) {
  blink(12, 500);
  blink(12, 500);
  blink(12, 500);
}

ros::Subscriber<std_msgs::Empty> subRed("blink_red", &blinkRedCb);
ros::Subscriber<std_msgs::Empty> subBlue("blink_blue", &blinkBlueCb);

void setup()
{
  pinMode(13, OUTPUT);
  pinMode(12, OUTPUT);

  nh.initNode();
  nh.subscribe(subRed);
  nh.subscribe(subBlue);
}

void loop()
{
  nh.spinOnce();
  delay(1);
}
```

Here is the diagram of all connections:



15.2 ROS

We can pass arguments via objective which points to rosbag file. Have a look at `blink.py` script. The main difference is `blink()` method:

```
...
```

(continues on next page)

(continued from previous page)

```
def blink(self, data):
    if data.data == "blue":
        rospy.loginfo("Blinking blue...")
        self.blink_blue.publish(Empty())

    if data.data == "red":
        rospy.loginfo("Blinking red...")
        self.blink_red.publish(Empty())

    rospy.wait_for_service('/liability/finish')
    fin = rospy.ServiceProxy('/liability/finish', FinishLiability)
    fin(FinishLiabilityRequest(address=self.liability, success=True))
    rospy.loginfo("Finished")

...
```

Now `/blink` topic has a `String` type. You can find prepared rosbags in `rosbag` folder.

15.3 AIRA

Connect to AIRA client via SSH as described here. Do not forget to add `COM1` port in settings. Run the following command:

```
$ rosrun arduino_with_args blink.py
```

Also we need to add rosbag files to IPFS:

```
$ ipfs add rosbag/blink_blue.bag
$ ipfs add rosbag/blink_red.bag
```

Note: Before the next step you should approve XRT tokens on the Factory.

The last step is to build Dapp and launch. Take a look at the previous lesson. To make Arduino blink the blue led send blue demand and blue offer messages. For the red one send corresponding messages.

That’s it! Now you are able to pass dynamic parameters to your cyber-physical system agent!

CHAPTER 16

Connect an Air Pollution Sensor

In this lesson you are going to learn how to connect your sensor to the network and make it publish data. You will see how it is easy to become a member of a global sensor network!

Note: Source code is located here

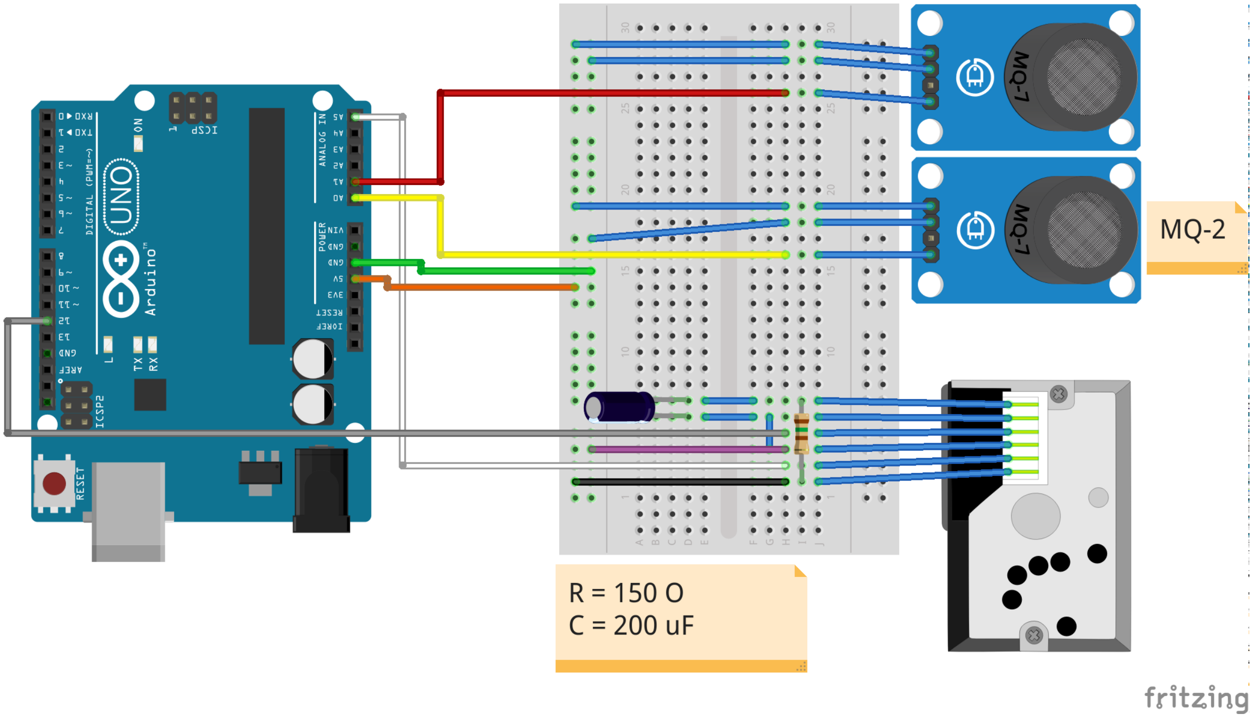
In this section we are not going to create a liability contract. Instead we will teach Arduino with sensors to publish the data by a request. All measurements will be published as a Result message

16.1 Arduino

Let's begin with an Arduino circuit. You need the following components:

- Arduino Uno
- Optical Dust Sensor Sharp GP2Y1010AU0F
- Gas Sensor MQ-2
- Gas Sensor MQ-7
- Resistor 150 Ohm
- Capacitor 220 uF
- Wires

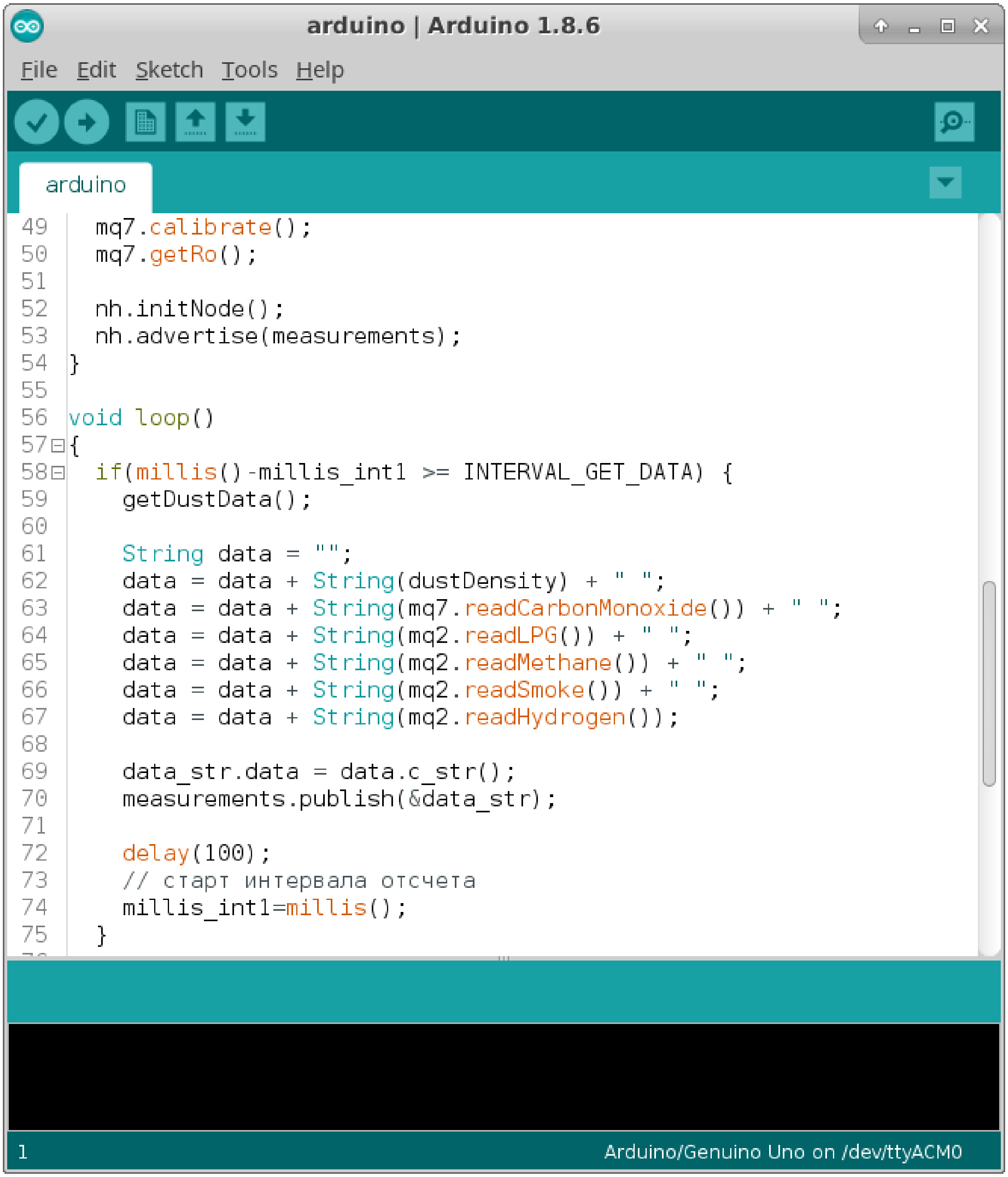
Connect all parts as described below:



MQ-2

R = 150 O
C = 200 uF

A firmware for Arduino Uno is in `sensor_city/scetches` folder. In order to upload it to the board use Arduino IDE.

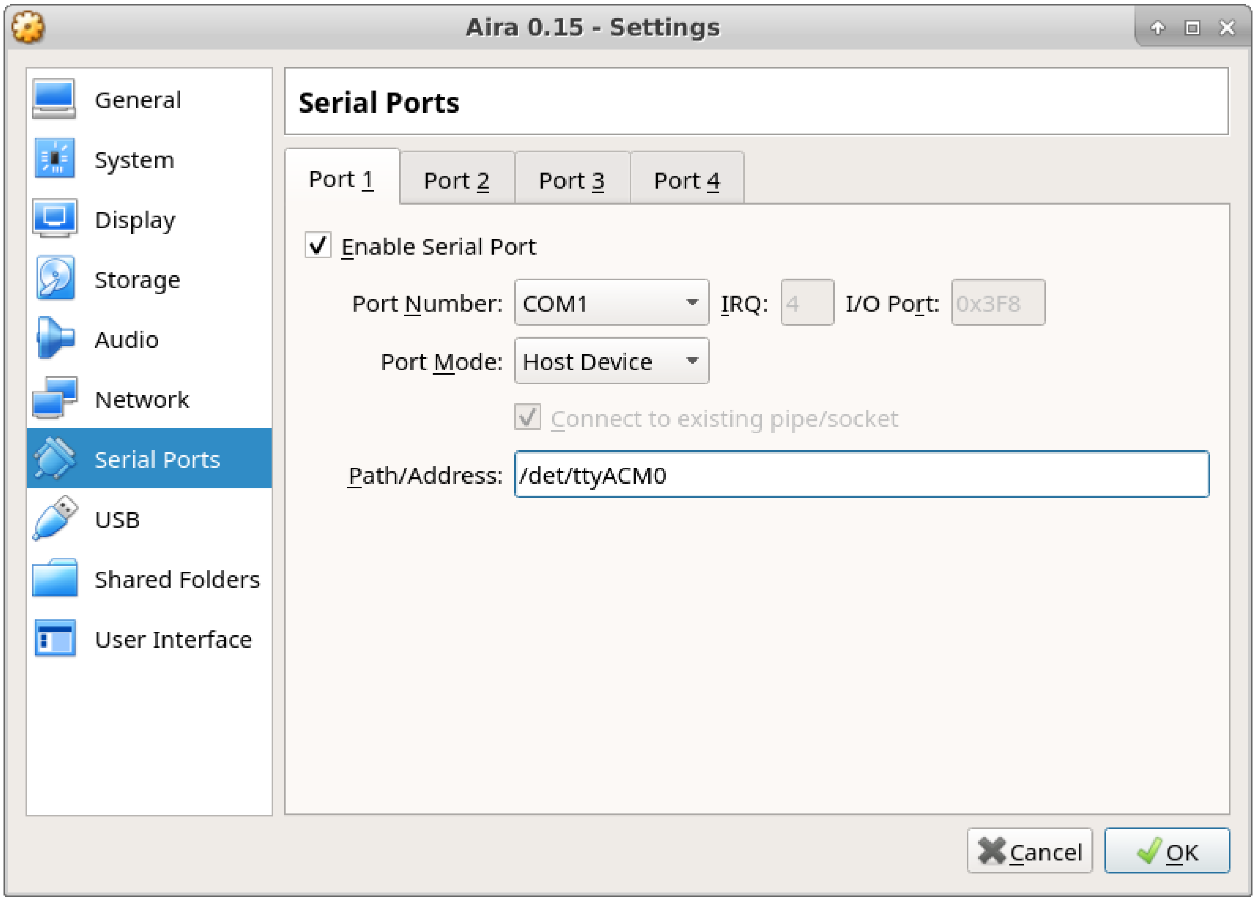


```
  mq7.calibrate();
  mq7.getRo();

  nh.initNode();
  nh.advertise(measurements);
}

void loop()
{
  if(millis()-millis_int1 >= INTERVAL_GET_DATA) {
    getDustData();

    String data = "";
    data = data + String(dustDensity) + " ";
    data = data + String(mq7.readCarbonMonoxide()) + " ";
    data = data + String(mq2.readLPG()) + " ";
    data = data + String(mq2.readMethane()) + " ";
    data = data + String(mq2.readSmoke()) + " ";
    data = data + String(mq2.readHydrogen());

    data_str.data = data.c_str();
    measurements.publish(&data_str);

    delay(100);
    // старт интервала отсчета
    millis_int1=millis();
  }
```

16.2 Aira

Note: The following steps are performed in Aira client. You can download the latest image from this page. It's convenient to connect via SSH

After you have imported the image to VirtualBox, connect Arduino via USB to your PC and enable serial port forwarding. You should check *Enable Serial Port* and assign `/dev/ttyACM0` in *Path/Address*. Inside the virtual machine `/dev/ttyS0` refers to your external Arduino.



☑ Enable Serial Port

Port Number: COM1 IRQ: 4 I/O Port: 0x3F8

Port Mode: Host Device

☑ Connect to existing pipe/socket

Path/Address: /det/ttyACM0

Finally launch the image and run these command:

```
$ roslaunch sensor_city publish_data.launch
```

Hint: Check out the source code to learn how it works under the hood!

Now Aira patiently waits for a signal to publish the measurements. Go to Dapp and click on *Broadcast signal*. You should see the data!

CHAPTER 17

Introduction

Robonomics-js is a simple Javascript library for working with Robonomics network

17.1 Installation

```
npm install robonomics-js --save
```

or

```
yarn add robonomics-js
```

CDN

```
<script src="https://cdn.jsdelivr.net/npm/robonomics-js/dist/robonomics.min.js"></
↪script>
```

17.1.1 Dependencies

- Web3
- Ipfs

17.2 Initialization

```
import Robonomics, { MessageProviderIpfsApi } from 'robonomics-js'
import IPFS from 'ipfs-api'

const robonomics = new Robonomics({
    provider: new MessageProviderIpfsApi(new IPFS('http://localhost:5001'))
```

(continues on next page)

(continued from previous page)

```
})

robonomics.ready().then(() => {
    console.log('robonomics js ready')
    console.log('xrt', robonomics.xrt.address)
    console.log('factory', robonomics.factory.address)
    console.log('lighthouse default', robonomics.lighthouse.address)
})
```

17.2.1 Available arguments

- `web3` - isn't necessary if Metamask is available
- `account` - isn't necessary if Metamask is available
- `privateKey` - optional
- `provider` - IPFS HTTP API
- `version` - the latest by default
- `ens` - ENS address, 0x314159265dD8dbb310642f98f50C066173C1259b by default
- `lighthouse` - a lighthouse name in ENS, airalab.lighthouse.1.robonomics.eth by default

CHAPTER 18

How to

18.1 How to create a demand?

Listen to a demand with a specific model:

```
const model = 'QmWXk8D1Fh5XFJvBodcWbwgyw9htjc6FJg8qi1YYEoPnrg'
robonomics.getAsk(model, (msg) => {
    console.log(msg)
})
const ask = {
    objective: 'QmSt69qQqGka1qwRRHbdmAWk4nCbsV1mqJwd8cWbEyhf1M',
    token: robonomics.xrt.address,
    cost: 1,
    deadline: 9999999
}
```

Fields:

- `objective` - IPFS hash to a rosbag file with a task
- `token` - token address
- `cost` - cost
- `validator` - validator address
- `validatorFee` - validator fee
- `deadline` - block number

It's necessary to make an approve:

```
robonomics.xrt.send('approve', [robonomics.factory.address, ask.cost], { from:␣
↪robonomics.account }).then((tx) => console.log(tx))
```

In case of other token:

```
import { Token } from 'robonomics-js'
const token = new Token(robonomics.web3, '0x123132132132132132132132132321')
token.send('approve', [robonomics.factory.address, ask.cost], { from: robonomics.
↪account })
  .then((tx) => console.log(tx))
```

And send a demand message:

```
robonomics.postAsk(market, ask)
    .then((liability) => {
        console.log('liability', liability.address)
        liability.watchResult((result) => {
            console.log('liability result', result)
        })
        return liability.getInfo()
    })
    .then((info) => {
        console.log('liability info', info)
    })
```

18.2 How to get an offer?

Obtain all the messages by a given model:

```
const model = 'QmWXk8D1Fh5XFJvBodcWbwgyw9htjc6FJg8qi1YYEoPnrg'
robonomics.getBid(model, (msg) => {
    console.log(msg)
})
```

Fields:

- `objective` - IPFS hash to a rosbag file with a task
- `token` - token address
- `cost` - cost
- `lighthouseFee` - lighthouse fee
- `deadline` - block number

18.3 How to listen to a result?

Obtain all the messages by a given model:

```
robonomics.getResult((msg) => {
    console.log(msg)
})
```

Note: It's not a verified result. A verified result could be obtained from a liability contract.

18.4 How to create a lighthouse?

```
const minimalFreeze = 1000 // Wn
const timeout = 25 // blocks
const name = 'mylighthouse' //
robonomics.factory.send('createLighthouse', [minimalFreeze, timeout, name], { from:␣
↪robonomics.account })
    .then((tx) => console.log(tx))

robonomics.factory.watchLighthouse((lighthouse) => {
    console.log(lighthouse.name)
})
```

18.5 How to become a provider?

```
const name = 'mylighthouse' //
const stake = 1000 // Wn

robonomics.setLighthouse(name)

robonomics.xrt.send('approve', [robonomics.lighthouse.address, stake], { from:␣
↪robonomics.account })
    .then((tx) => console.log(tx))

robonomics.lighthouse.send('refill', [stake], { from: robonomics.account })
  .then((tx) => console.log(tx))
```

18.6 How to change a lighthouse?

```
robonomics.setLighthouse(name)
```

18.7 How to check the balance?

```
robonomics.xrt.call('balanceOf', [robonomics.account])
.then((balance) => console.log('balance', balance))
```

18.8 How to check the allowance?

```
robonomics.xrt.call('allowance', [robonomics.account, robonomics.factory.address])
    .then((allowance) => console.log('allowance', allowance))
```

CHAPTER 19

Creating Dapp

Almost every project needs a user interface to interact with. A user should not type in a *Demand* message. In Airalab repository there's a convenient template for a Dapp. In this section you are going to learn how to get a new Dapp for your CPS.

Note: The source code is here

To get a template you don't even have to clone the repo. Instead do these steps:

```
$ npm install -g vue-cli
$ vue init airalab/vue-dapp-robonomics-template my-project
$ cd my-project
$ npm install
$ npm run dev
```

After the last step a webserver has started on http://localhost:8000/. But before you open this link in a browser you should configure the Dapp.

Note: MetaMask is required for the Dapp

Here is a configuration file below. You have to specify a `LIGHTHOUSE` you work on, your CPS `MODEL` and `OBJECTIVE`. Also the Dapp uses IPFS message broker. You can either set up your own broker or use existing one, for example *https://wss.pool.aira.life*.

```
export const NETWORK = 1
export const LIGHTHOUSE = 'airalab.lighthouse.3.robonomics.eth'
export const MODEL = 'QmdFh1HPVe7H4LrDio899mxA7NindgxqiNUM9BNnBD7ryS'
export const OBJECTIVE = 'QmbSW1E73DKUvGDrgx8GirEVfHJLvj8RBijtH9iEZ7UecU'
export const IPFS_PUBSUB = 'http://127.0.0.1:9999'
export const ENS = ''
export const VERSION = 1
```

After editing the file, launch the Dapp

```
$ npm run dev
```

Check the source code out to get familiar with the structure of the template.

Good luck!