BLOCKCHAIN TESTBED GUIDE/USER MANUAL

The following document will go over several areas of the blockchain testbed, what needs to be configured and what commands must be used in order to perform testing. Currently several areas of the testbed are still missing such as network delay, fully automated testing and multi-host capable setups. This guide is written with this in mind and will, to the extent possible not be impacted by this.

DOCKER

Docker is the virtualization/scaling tool used thourghout the testbed to achieve a larger setup/architecture than what physical nodes would allow us. Docker is faily simple and consists of two files. The Docker file and the Docker-compose file.

Docker File

The docker file specifies what is needed inside each docker container. It crates an image which each container will be based upon. For this setup the docker file uses a standard Ubuntu image and specifies several instances of things to install upon this image. Once this is done each container can then simply launch this docker image and have a running Ubuntu with the specified packages. It is here important to note that only the things pre-specified in the docker file can be used on the container. Any change to the docker file requires re-building the docker image. (If for instance ping is required, it must be added to the docker file and the image must be re-build in order to have access to ping on the containers). To build the image the following command is used:

Docker build –t <nameOfImage> .

This requires that the command is cast inside the docker folder. Other useful commands:

- Docker images (shows all existing images)
- Docker ps (shows running containers)

Docker-compose file

The docker compose file specifies the number and the configuration of the docker containers. It is in this file the size of the network must be specified. If for instance a network with 3 nodes is desired this file must contain configurations for 3 nodes as well as the network bridge between them. An example of such a node is:

- miner1:
  - image: test
  - entrypoint: bash
  - tty: true
  - ports:
    - - 9000
    - - 30303
    - - 30301
    - - 8545
  - networks:
    - app_net:

        - ipv4_address: 172.18.0.100
    - volumes:
        - /home/administrator/testbed/docker/composer/filecontainer:/workspace
    - command: -c "cd ../workspace &&
      geth --datadir miner1 --networkid 1234 init genesisTesting.json 2>> miner1.txt &&
      geth --datadir miner1 --networkid 1234 --exec 'personal.newAccount("\"1234"\")' console 2>> miner1.txt &&
      geth --datadir miner1 --networkid 1234 --verbosity 3 --exec 'personal.unlockAccount(eth.accounts[0], "\"1234"\", 10000)' --mine --rpc --rpcapi "admin, eth, personal, miner, txpool, net, web3" 2>> miner1.txt &&
      bin/bash"

As well as the network bridge used for them to communicate through:

- networks:
    - app_net:
        - driver: bridge
        - driver_opts:
            - com.docker.network.enable_ipv6: "true"
        - ipam:
            - driver: default
            - config:
                - - subnet: 172.18.0.0/24
                - gateway: 172.18.0.1
                - - subnet: 2001:3984:3989::/64
                - gateway: 2001:3984:3989::/1

The command label specifies which commands/tasks the container must run when it starts. This allows us to distinguish between miner nodes and non-miner nodes as only the miner nodes are required to start mining. The image label in the top must match the name given the use image build above in order to have access to the desired configuration. In order to launch these nodes on a single-host machine the following command is used (it is assumed the command is cast inside the docker/composer folder):

Docker-compose up –d

This will launch all containers specified in the file as well as the network bridge. The –d flag simply performs the startup in the background. You can now do a docker ps command to see if/what containers are online. The output will be something like this:
In a fully automated setup this would be all which is required in order to create logfiles for the nodes. However, as there are several things not yet present in the system this is not enough yet and it is necessary to access some of the containers to perform tasks manually.
Once you have attached to a docker container you can utilize truffle to launch the contracts. Truffle has been installed on the containers through the docker file. It is necessary to enter a docker container which holds some ether (use miner1 as it will always exist) this can be achieved by doing:

docker exec-i -t composer_miner1_1 /bin/bash

This will open a terminal on this container allowing you to interact with it. The path to truffle is ../workspace/truffle
When there some miner configuration of the truffle files is required to fit the test paramters which are randomly generated at each test run. In order to make it as simple as possible I have created a python script which sets the necessary things. In order to launch contracts simply do:

python truffleFileCreater.py

rm -r build

truffle migrate

It is important to remove the build folder as else the new contracts will not be compiled. Truffle migrate compiles, builds and commits the contracts to the miners txpool through an rpc tunnel set in the truffle.js file.

This process is required any time a change is made to a contract. The contracts can be found in testbed/docker/composer/filecontainer/truffle/contracts. To launch new contracts they are required to be added to one of the migration files or to create a seperate migration file for that contract. This is quite simple:

```
var subscription = artifacts.require("./subscription.sol");
module.exports = function(deployer) {
  deployer.deploy(subscription);
};
```

This is the only required content of a migration file to launch a contract. If the contract use libraries or dependancies these must be added first as they are linked back towards the contract:

```
var StringUtilsLib = artifacts.require("./StringUtils.sol");
var stringsLib = artifacts.require("./strings.sol");
var Offer = artifacts.require("./Offer.sol");

module.exports = function(deployer) {
  deployer.deploy(StringUtilsLib);
  deployer.deploy(stringsLib);
  deployer.link(StringUtilsLib, Offer)
  deployer.link(stringsLib, Offer)
  deployer.deploy(Offer);
};
```

It is important to note that the gas limit of the blocks must be high enough to contain the entire contract in a single block. Currently this is around 6.5 million gas. If the gas limit in the block fall below this (due to mining blocks with no transactions) the contracts cannot be submitted. This requires either starting over (launching a new chain) or creating transactions and waiting for the gas limit it increase.

SCRIPTS

To achieve this there are several different functions inside the createTransactions.js script.

CreateTransactions.js

This script was originally designed to crate offerings faster than a human could do it. It contains the following:

- createTransactions(int numberOfTransactions)

This function creates valid offerings towards the offering contract. The total number of offerings created is set by the numberOfTransactions. These transactions vary somewhat in category and pricing model. However further changes can easily be made in the source code.

- moreGas(numberOfTransactions)

This function creates transaction with the sole purpose of increasing the gas limit of the blocks. These transactions have no valid destination and will not effect the state of the offering contract.

- deleteTransactions(startID, endID)

This functions delete offerings in the smart contract from a specified starting to ending ID. It is important to note that only offerings owned by the sender/functionCaller will be deleted.

- modifyTransactions(startID, endID)

This function was used for testing the modifications of offerings. This function changes the endpoint set in the offering in the given interval. It is important to note that only offerings owned by the sender/functionCaller will be modified.

GetListOffers.js

The getListOffers.js script contains all the implemented search algorithms in the contracts. If a node wants to search through the offerings it loads this script which gives it access to AMR and RFMR search methods.

The search function is called offeringQueryList() and takes the following parameters:
offeringQueryList(category, price, inputs, outputs, ageLimit, method, loops)

- Category (string): Specifying the category of which the offering is a member of
- price (integer): the maximum price the consumer is willing to pay
- inputs (string array): Specifies the inputs the offerings must be able to support
- outputs (string array): Specifies the outputs the offerings must be able to support
- ageLimit (integer): How many blocks must have been mined after the offering. This increases the chance that the offerings wont disappear in forking scenarios.
- Method (string: “AMR”/”RFMR”) specifies which type of search is to be used
- loops (integer): is used for testing fairness of searching methods and can be omitted on normal runs.

IMPORTANT: for all scripts the following is always required when new contracts are launched:

- Copy the contract address given by truffle and insert it into the scripts returnContractAddress() function. Else the script will not know where the contract is.
- If changes has been made to the contract since last launch the ABI of the contract found in its build file in truffle must be placed in the returnContractInterface() functions as it would otherwise not know the structure of the contract.

Docker-compose file creation

to simplify the creation of different sized setups a java program has been created. It is located in testbed/docker/composer and is called composer. This java program allows to easily create docker files with many nodes. It takes the following 3 parameters:

Writer writer = new Writer(miners, nodes, logDetailLevel)

- miners (integer): the amount of miners in the network
- nodes (integer): the amount of nodes in the network who are not miners (total sum of nodes will be miners + nodes)
- logDetailLevel (integer: 1-6): this specifies the level of detail in the log files 1 being the least a 6 being the most. At 3 the most common and useful information regarding blocksharing is available. I recommend using 4, but if you are interested in the peer-sharing, handshaking and so on you should go higher.

Notes & discoveries:

It is very difficult to create a fully automated setup as many of the parameters required are first generated at run-time (contract addresses, node ID numbers for peering, account addresses). This includes launching contracts which for now must be done manually by connecting to one of the containers. Automatic peer discovery is also difficult as the bootnode key is not known beforehand, it must be set manually for now, which becomes a huge task if the setup is large. This should therefore definitely be investigated more.

Some of the implementations made in the java file, to automatically create the docker file, limits the amount of nodes to 100 which is a limitation, it can be changed but the difficulty is that the nodes are required to be on the same subnet and to avoid them getting the same ips some hardcoding and hard caps have been created. This can be changed but I dont think a single setup system, no-matter how powerful will be able to support a 100 node setup if it, at the same time, also has multiple miners.

The single node setup is the easiest to make as there is essentially a working version now. However, the big issue is as mentioned the necessity to still perform manual tasks to get it up and running and more importantly the issue of network delay. We have not found a way to simulate network delay to a level which was desired. We have investigated ns-3 as a possible solution to this but we have no experience with this and there have been little progress. (Peter knows more about this than me, so he should be the point man on this). Likewise the topology is again very difficult to control as it again requires setting paths manually which in it-self is time consuming. But is also required the knowledge of parameters which are first generated when the nodes are created i.e. at runtime. This makes if very difficult to created customized topologies in the network.

The multi-host setup has been even more challenging as it has been difficult to get the containers (nodes) on different hosts to communicate. We have looked at different solutions such as individual docker instances on the hosts, docker swarm and so on. We seem to have had little success in this area however I think Peter mentioned that he could ping the containers in the swarm setup. It is also necessary to think about the time-sync between host and how this impacts the log-files created. In the single host setup they use the same time-source and they are therefore directly comparable. This is not necessarily the case and this can cause problems in interpretation of the data.

Currently the bootnodes must be implemented again, this should probably be the first task. I have/will add the delay method of using linux’s netem tool for delays. The delays are specified by the following:

tc qdisc add dev eth0 root netem delay 100ms 20ms distribution normal

There are a few different distributions and a lot of other functionalities in netem which can become useful later on. Check out https://wiki.linuxfoundation.org/networking/netem for more information regarding netem.