Concord Documentation

VMware

Nov 05, 2019

Contents

1 Contents **3**

1.1 Getting Started . . . 3

1.2 Deployment . . . 4

1.2.1 Concord Nodes . . . 5

1.2.2 EthRPC Nodes . . . 6

1.3 Tutorials . . . 7

1.3.1 Installing your first smart contract . . . 7

1.4 Developer . . . 10

1.4.1 Launching the Dev Container . . . 10

1.4.2 Code Format . . . 10

1.5 Getting Help . . . 10

CONCORD

Project Concord is an open source scalable decentralized Blockchain. It leverages the Concord-BFT engine based on years of academic and industrial research, and implements a Blockchain which supports running Ethereum smart contracts.

This documentation is written for different types of users. IT administrators who wish to deploy Concord will want to read the installation and deployment section. Blockchain developers who want to build and add new smart contracts will want to read the tutorials, and developers who want to add new functionality, such as adding support for new APIs or state machines will want to read the developer section. For all users new to Concord, we recommend starting with the *getting started* section.

The table of contents below and on the left provides easy access to documentation for your topic of interest. You can also use the search bar in the top left corner.

CHAPTER 1

Contents

Keyword Index, Search Page

1.1 Getting Started

The preferred way to deploy and develop on Concord is through the use of Docker, which should be all you need to get a deployment of Concord running locally.

Once you have Docker installed, checkout the Concord repository:

```
git clone https://github.com/vmware/concord.git
```

After that, from the Concord directory, run the build script for the Docker images:

```
cd concord
docker/build.images.sh
```

You're now ready to launch Concord. Locally, `docker-compose` makes it very easy to setup a Concord cluster.

> **Attention:** The following instructions setup Concord to use insecure (http) communication for simplicity and should not be used in production.

To launch a simple 4-node Concord cluster, launch the `simple4.yml` file:

```
docker-compose -f docker/compose/simple4.yml up
```

You should see some messages go by, indicating the cluster is starting up. Once the cluster is up, the next setup is to interact with it. Since this cluster comes with the `ethRPC` bridge, we can communciate with it using standard Ethereum tooling.

Truffle is a popular tool for developing and debugging Ethereum smart contracts. If you're familar with Truffle (note that we only support Truffle 4.x at the moment) and have used it before, you can connect to the blockchain instance on your local machine at http://localhost:8545, which will connect to the first Concord node in your system.

Otherwise, the repository contains a docker image with Truffle pre-installed to deploy to the simple 4 node Concord instance you just created. To start the image and connect to the first Concord node, run:

```
docker exec -it compose_concord-truffle_1 bash
truffle console  --network ethrpc1
```

Now you can run a test transaction using Truffle. Type the following in the Truffle console:

```
var accounts = await web3.eth.getAccounts()
await web3.eth.sendTransaction({from: accounts[0], to: accounts[1], value: web3.utils.
↪toWei("0", "ether")})
```

You should get ouptut similar to:

```
{
blockHash: '0xb3bb6deed1446b30dcdaec50faf1b7fe40f8543bccb552743d4869cab5b50e63',
logsBloom: '0x00',
gasUsed: 10000000,
blockNumber: 6,
cumulativeGasUsed: 10000000,
contractAddress: null,
transactionIndex: 0,
from: '0x262c0d7ab5ffd4ede2199f6ea793f819e1abb019',
to: '0x5bb088f57365907b1840e45984cae028a82af934',
logs: [],
transactionHash: '0x9a6f2ae673da7454d66c74116183f680b0ea5d06e49f04bbcd312f0b362ac705',
status: true
}
```

This creates a transaction which takes 0 Ether from test account 0 and sends it to test account 1, and prints out the information of the resulting transaction. Congratulations, you've just executed your first Ethereum transaction on Concord!

From here, you can look into how to deploy Concord, read the tutorials on how to install and interact with smart contracts on Concord, or learn how to develop support for your own custom API on Concord.

1.2 Deployment

Currently, a Concord deployment consists of two types of nodes:

- `Concord` nodes, which participate in the the Byzantine consensus protocol and perform state machine replication.
- `EthRPC` nodes, which translate Ethereum API calls into requests that `Concord` nodes can understand.

A deployment needs to contain $3F + 1$ `concord` nodes, where F is the number of failures the deployment needs to tolerate. In a typical deployment, each `Concord` node will have a single, possibly co-located `EthRPC` node. Ethereum applications, such as Truffle, interact with the deployment through the `EthRPC` node.

Each node is typically deployed as a Docker container, which allows for easy deployment through tools such as `docker-compose` and `Kubernetes`.

In the next sections, we detail how to setup each type of node.

1.2.1 Concord Nodes

Concord nodes are provided by the `concord-node` container. Typically, you run the container with the command:

```
docker run concord-node:latest --expose 3501,3502,3503,3504,3505 -p 5848 -v <log-path>
↪:/concord/log \
-v <rocksdb-path>:/concord/rocksdbdata -v <localconfig-path>:/concord/config-local \
-v <publicconfig-path>:/concord/config-public:ro -v <tlscert-path>:/concord/tls_
↪certs:ro
```

The paths contain the database, log, configuration and TLS certificates for the node running in the container.

- `<log-path>` is a path where Concord stores debug logs.
- `<rocksdb-path>` is a path that contains the rocksdb database replicated by the state machine.
- `<localconfig-path>` is a path containing a `concord.config` file for configuring this Concord instance.
- `<publicconfig-path>` is a path containing a `genesis.json` file, which describes the genesis block for the deployment, and a `log4cplus.properties` file which configures the logging level for each component.
- `<tlscert-path>` is a path to TLS certificates used to encrypt communication. The certificates are stored in `pem` format.

Ports 3501-3505 are exposed because they are specified as communication ports in the configuration files that other nodes will communicate with. Port 5848 is published to the host, so that `EthRPC` nodes can easily communicate using the hosts local IP address.

Concord requires two sets of configuration files. The local `concord.config` file configures per node settings and the public and private keys used to sign requests. The public `dockerConfigurationInput.yml` configure global settings, such as the size of the cluster and its configuration. Both of these files are yaml files and their schema is documented in the next sections. The public config also includes a `genesis.json` file which defines the genesis (or initial) state of the system. This includes information about the genesis block as well as the initial accounts, balance and storage.

The TLS certificates are stored in `pem` format and stored in subdirectories named after the principal id specified in the *concord.config* file. Each principal has a client/server pair, consisting of the public `client.cert` and `server.cert`, as well as a `pk.pem` file containing the private key. More details about the TLS certificate directory format and repository can be found in the next sections.

A convenience tool for generating configuration files and TLS certificates simplifies deployment.

Configuration File Generator

Concord nodes each have to have their own configuration files and certificates, which may be difficult to generate by hand. To simplify this task, we provide a convenience tool, `conc_genconfig` which generates configuration files as well as the TLS certificates necessary to start a Concord deployment. `conc_genconfig` takes a `yaml` file named `configurationInput.yml`.

configurationInput.yml

Note: The deployment documentation is currently a work in progress. Help contribute documentation by submitting a pull request! You can edit this page by clicking on the `Edit on GitHub` link on the top right.

concord.config

Note: The deployment documentation is currently a work in progress. Help contribute documentation by submitting a pull request! You can edit this page by clicking on the `Edit on GitHub` link on the top right.

genesis.json

The `genesis.json` file contains the genesis (initial) state of the system. It is stored in JSON format, and a sample file is shown below:

```
{
   "config": {
      "chainId": 1,
      "homesteadBlock": 0,
      "eip155Block": 0,
      "eip158Block": 0
   },
   "alloc": {
      "262c0d7ab5ffd4ede2199f6ea793f819e1abb019": {
         "balance": "12345"
      },
      "5bb088f57365907b1840e45984cae028a82af934": {
         "balance": "0xabcdef"
      },
      "0000a12b3f3d6c9b0d3f126a83ec2dd3dad15f39": {
         "balance": "0x7fffffffffffffff"
      }
   },
   "nonce": "0x000000000000000",
   "difficulty": "0x400",
   "mixhash": "0x0000000000000000000000000000000000000000000000000000000000000000",
   "parentHash": "0x0000000000000000000000000000000000000000000000000000000000000000",
   "gasLimit": "0xf4240"
}
```

TLS certificates

Note: The deployment documentation is currently a work in progress. Help contribute documentation by submitting a pull request! You can edit this page by clicking on the `Edit on GitHub` link on the top right.

1.2.2 EthRPC Nodes

Concord nodes are provided by the `concord-ethrpc` container. Typically, you run the container with the command:

```
docker run concord-ethrpc:latest java -jar concord-ethrpc.jar --ConcordAuthorities=
↪<host>:<port> \
--security.require-ssl=true --server.ssl.key-store-type=PKCS12 --server.ssl.key-
↪store=/config/keystore.p12 \
--server.ssl.key-store-password=Ethrpc!23 --server.ssl.key-alias=ethrpc
```

Attention: To simplify deployment, you may disable SSL by setting `-security.require-ssl=false`. However, this is not recommended in production environments for security reasons.

The `--ConcordAuthorities=<host>:<port>` specifies the `<host>`, a hostname or ip address and API `<port>` of the Concord node. The `--security.ssl.*` parameters specify the SSL key that the HTTPS endpoint will use. The defaults shown above are for the self-signed certificates provided in the `docker/resources/config-ethrpc*` folders. You may configure the server with your own certificate.

1.3 Tutorials

Note: The tutorials documentation is currently a work in progress. Help contribute documentation by submitting a pull request! You can edit this page by clicking on the `Edit on GitHub` link on the top right.

1.3.1 Installing your first smart contract

In this first tutorial, we'll install a simple Hello World contract into Concord and interact with it. This tutorial assumes you've followed the steps from the getting started, Start by running the simple four node Concord instance, `simple4.yml` from the getting started file:

```
docker-compose -f docker/compose/simple4.yml up
```

Next, start the truffle image:

```
docker exec -it compose_concord-truffle_1 bash
```

We'll now create a simple hello world contract. We'll use `vim` to create the contract, which needs to be inserted in the contracts directory:

```
vi contracts/HelloWorld.sol
```

Insert the following code and save the file:

```
pragma solidity ^0.5.8;
contract HelloWorld {
    address public creator;
    string public message;

    constructor() public {
        creator = msg.sender;
        message = 'Hello, world';
    }
}
```

Now, we'll need to create a javascript file to instruct truffle to deploy the contract. This file needs to be inserted in the migrations directory:

```
vi migrations/2_deploy_contracts.js
```

Insert the following code and save the file:

```
var HelloWorld = artifacts.require("./HelloWorld.sol");

module.exports = function(deployer) {
deployer.deploy(HelloWorld);
};
```

Now we can deploy the contract by running:

```
truffle migrate --network ethrpc1
```

If the contract is sucessfully deployed, you should see output similar to:

```
Compiling your contracts...
===========================
> Compiling ./contracts/HelloWorld.sol
> Compiling ./contracts/Migrations.sol
> Artifacts written to /truffle/build/contracts
> Compiled successfully using:
   - solc: 0.5.8+commit.23d335f2.Emscripten.clang



Starting migrations...
======================
> Network name:    'ethrpc1'
> Network id:      5000
> Block gas limit: 0xf4240


1_initial_migration.js
======================

   Deploying 'Migrations'
   ----------------------
   > transaction hash:    ␣
↪0x7a62d178231bcee0167fd82330e96781bcf15e08c186a143de68bf0ce0a163c5
   > Blocks: 0            Seconds: 0
   > contract address:    0xc2b3150D03A3320b6De3F3a3dD0fDA086C384eB5
   > block number:        1
   > block timestamp:     1571682384
   > account:             0x262C0D7AB5FfD4Ede2199f6EA793F819e1abB019
   > balance:             0.000000000000012345
   > gas used:            5449
   > gas price:           0 gwei
   > value sent:          0 ETH
   > total cost:          0 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:                   0 ETH


2_deploy_contracts.js
=====================
```

(continues on next page)

(continued from previous page)

```
   Deploying 'HelloWorld'
   ----------------------
   > transaction hash:    ␣
↪0x73bc4a9e91b0da818cec84259c0faee8e46d26bdffa14d72ad51447c4e2870d6
   > Blocks: 0            Seconds: 0
   > contract address:    0x7373de9d9da5185316a8D493C0B04923326754b2
   > block number:        3
   > block timestamp:     1571682385
   > account:             0x262C0D7AB5FfD4Ede2199f6EA793F819e1abB019
   > balance:             0.000000000000012345
   > gas used:            11019
   > gas price:           0 gwei
   > value sent:          0 ETH
   > total cost:          0 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:                   0 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0 ETH
```

Next, we'll want to interact with the contract. We can do that through the truffle console:

```
truffle console --network ethrpc1
```

The truffle console accepts javascript as input. We can get access to the HelloWorld contract through the `HelloWorld` variable. We want the deployed version of the contract, which we can retrieve using the asynchronous `deployed()` method. In javascript, we can use await to wait for an asynchronous function (which returns a `Promise`) to complete:

```
var app = await HelloWorld.deployed()
```

Now you can access the contract through the `app` variable. If you type `app.` and press tab, tab completion should give you the list of functions you can call:

```
truffle(ethrpc1)> app.
app.__defineGetter__      app.__defineSetter__      app.__lookupGetter__      app.__
↪lookupSetter__      app.__proto__
app.hasOwnProperty        app.isPrototypeOf         app.propertyIsEnumerable  app.
↪toLocaleString        app.toString
app.valueOf

app.abi                   app.address               app.allEvents             app.
↪constructor           app.contract
app.creator               app.message               app.send                  app.
↪sendTransaction
app.transactionHash
```

Try calling the message function to view the message we entered in the app. Remember, the function is asynchronous, so use the `await` keyword to resolve the `Promise`:

```
await app.message.call()
> 'Hello, world'
```

Congratulations, you just installed your first smart contract on Concord and made a simple call to one of its functions.

1.4 Developer

Note: The developer documentation is currently a work in progress. Help contribute documentation by submitting a pull request! You can edit this page by clicking on the `Edit on GitHub` link on the top right.

Concord is developed inside of Docker containers to simplify dependency management, and the easiest way to get a development environment setup for Concord is by using Visual Studio Code, which includes Dev Container features, making it easy to deploy and debug from within a container.

Concord has a `devcontainer.json` file included in the repository which will correctly setup the container and build environment.

1.4.1 Launching the Dev Container

To get a dev container environment for Concord, open your Concord checkout in Visual Studio Code. Open the command palette (`View > Command Palette...`) and type `Remote Containers: Reopen in container`.

The dev container environment is setup with CMake tooling. When prompted to select a CMake kit, select `clang-7.0`.

1.4.2 Code Format

We enforce code formatting through `clang-format`. The build will fail if there are code formatting errors. To fix formatting errors automatically, run the `format` target, which will attempt to automatically fix any code formatting issues in the repository.

1.5 Getting Help

Currently, the best place to get help is via the Github issue tracker.