# EOS Testing Environment on ubuntu 16.04

## Environment configuration

### 1. Install the development toolkit:

```
 sudo apt-get update
 wget -O - https://apt.llvm.org/llnapshot.gpg.key|sudo apt-key add -
 sudo apt-get install clang-4.0 lldb-4.0 libclang-4.0-dev cmake make \
 libbz2-dev libssl-dev libgmp3-dev \
 autotools-dev build-essential \
 libbz2-dev libicu-dev python-dev \
 autoconf libtool git
```

### 2. Install Boost 1.64:

```
cd ~
wget -c 'https://sourceforge.net/projects/boost/files/boost/1.64.0/boost_1_64_0.ta
r.bz2/download' -O boost_1.64.0.tar.bz2
tar xjf boost_1.64.0.tar.bz2
cd boost_1_64_0/
echo "export BOOST_ROOT=$HOME/opt/boost_1_64_0" >> ~/.bash_profile
source ~/.bash_profile
./bootstrap.sh "--prefix=$BOOST_ROOT"
./b2 install
source ~/.bash_profile
```

### 3. Install secp256k1-zkp (Cryptonomex branch):

```
cd ~
git clone https://github.com/cryptonomex/secp256k1-zkp.git
cd secp256k1-zkp
./autogen.sh
./configure
make
sudo make install
```

### 4. To use the WASM compiler, EOS has an external dependency on binaryen:

```
  cd ~
  git clone https://github.com/WebAssembly/binaryen.git
  cd ~/binaryen
  git checkout tags/1.37.14
  cmake . && make
  echo "export BINARYEN_ROOT=~/binaryen" >> ~/.bash_profile
  source ~/.bash_profile
```

### 5. By default LLVM and clang do not include the WASM build target, so you will have to build it yourself:

```
  mkdir  ~/wasm-compiler
  cd ~/wasm-compiler
  git clone --depth 1 --single-branch --branch release_40 https://github.com/llvm-mirror/llvm.git
  cd llvm/tools
  git clone --depth 1 --single-branch --branch release_40   https://github.com/llvm-mirror/clang.git
  cd ..
  mkdir build
  cd build
  cmake -G "Unix Makefiles" -DCMAKE_INSTALL_PREFIX=.. -DLLVM_TARGETS_TO_BUILD= -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=WebAssembly -DCMAKE_BUILD_TYPE=Release ../
  make -j4 install
```

# Building EOS and running a node

## 1.Getting the code

To download all of the code, download EOS source code and a recursion or two of submodules. The easiest way to get all of this is to do a recursive clone:

```
 git clone https://github.com/eosio/eos --recursive
```

## 2.Building from source code

The `WASM_LLVM_CONFIG` environment variable is used to find our recently built WASM compiler. This is needed to compile the example contracts inside `eos/contracts` folder and their respective tests.

```
cd ~
git clone https://github.com/eosio/eos --recursive
mkdir -p ~/eos/build && cd ~/eos/build
echo "export WASM_LLVM_CONFIG=~/wasm-compiler/llvm/bin/llvm-config" >> ~/.bash_pro
file
echo "export LLVM_DIR=/usr/lib/llvm-4.0/lib/cmake/llvm" >> ~/.bash_profile
cmake -DBINARYEN_BIN=~/binaryen/bin -DOPENSSL_ROOT_DIR=/usr/local/opt/openssl -DOP
ENSSL_LIBRARIES=/usr/local/opt/openssl/lib ..
make -j4
```

## 3.Creating and launching a single-node testnet

After successfully building the project, the eosd binary should be present in the `build/programs/eosd` directory. Go ahead and run eosd -- it will probably exit with an error, but if not, close it immediately with Ctrl-C. Note that eosd created a directory named `data-dir` containing the default configuration ( `config.ini` ) and some other internals. This default data storage path can be overridden by passing `--data-dir /path/to/data` to eosd.

Edit the config.ini file, adding/updating the following settings to the defaults already in place:

```
# Load the testnet genesis state, which creates some initial block producers with
the default key
genesis-json = /root/eos/genesis.json
# Enable production on a stale chain, since a single-node test chain is pretty muc
h always stale
enable-stale-production = true
# Enable block production with the testnet producers
producer-name = inita
producer-name = initb
producer-name = initc
producer-name = initd
producer-name = inite
producer-name = initf
producer-name = initg
producer-name = inith
producer-name = initi
producer-name = initj
producer-name = initk
producer-name = initl
producer-name = initm
producer-name = initn
producer-name = inito
producer-name = initp
producer-name = initq
producer-name = initr
producer-name = inits
producer-name = initt
producer-name = initu

p2p-server-address = 39.104.66.16(your external IP)
# Load the block producer plugin, so you can produce blocks
plugin = eosio::producer_plugin
# Wallet plugin
plugin = eosio::wallet_api_plugin
# As well as API and HTTP plugins
plugin = eosio::chain_api_plugin
plugin = eosio::http_plugin
```

Now it should be possible to run eosd and see it begin producing blocks.

When running eosd you should get log messages similar to below. It means the blocks are successfully produced.

```
575001ms thread-0   chain_controller.cpp:235      _push_block          ] initm #1
@2017-09-04T04:26:15  | 0 trx, 0 pending, exectime_ms=0
1575001ms thread-0   producer_plugin.cpp:207       block_production_loo ] initm ge
nerated block #1 @ 2017-09-04T04:26:15 with 0 trxs  0 pending
1578001ms thread-0   chain_controller.cpp:235      _push_block          ] initc #2
 @2017-09-04T04:26:18  | 0 trx, 0 pending, exectime_ms=0
1578001ms thread-0   producer_plugin.cpp:207       block_production_loo ] initc ge
nerated block #2 @ 2017-09-04T04:26:18 with 0 trxs  0 pending
            ...
```

# Example "Currency" Contract Walkthrough

EOS comes with example contracts that can be uploaded and run for testing purposes. Next we demonstrate how to upload and interact with the sample contract "currency".

## 1.First, run the node

```
cd ~/eos/build/programs/eosd/
./eosd
```

## 2.Setting up a wallet and importing account key

As you've previously added `plugin = eosio::wallet_api_plugin` into config.ini, EOS wallet will be running as a part of eosd process. Every contract requires an associated account, so first, create a wallet.

```
cd ~/eos/build/programs/eosc/
./eosc wallet create # Outputs a password that you need to save to be able to lock
/unlock the wallet
```

For the purpose of this walkthrough, import the private key of the inita account, a test account included within genesis.json, so that you're able to issue API commands under authority of an existing account. The private key referenced below is found within your config.ini and is provided to you for testing purposes.

```
./eosc wallet import 5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3
```

## 3.Creating accounts for sample "currency" contract

First, generate some public/private key pairs that will be later assigned as owner*key and active*key.

```
cd ~/eos/build/programs/eosc/
./eosc create key # owner_key
./eosc create key # active_key
```

This will output two pairs of public and private keys

Private key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX Public key: EOSXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX Important: Save the values for future reference.

Run the create command where inita is the account authorizing the creation of the currency account and PUBLIC*KEY*1 and PUBLIC*KEY*2 are the values generated by the create key command

```
./eosc create account inita currency PUBLIC_KEY_1 PUBLIC_KEY_2
```

You should then get a JSON response back with a transaction ID confirming it was executed successfully.

Go ahead and check that the account was successfully created

```
./eosc get account currency
```

If all went well, you will receive output similar to the following:

```
{
  "account_name": "currency",
  "eos_balance": "0.0000 EOS",
  "staked_balance": "0.0001 EOS",
  "unstaking_balance": "0.0000 EOS",
  "last_unstaking_time": "1969-12-31T23:59:59",
  "permissions": [{
    ...
    }
  ]
}
```

Now import the active private key generated previously in the wallet:

```
./eosc wallet import XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

## 4.Upload sample "currency" contract to blockchain

Before uploading a contract, verify that there is no current contract:

```
./eosc get code currency
code hash: 0000000000000000000000000000000000000000000000000000000000000000
```

With an account for a contract created, upload a sample contract:

```
./eosc set contract currency ../../contracts/currency/currency.wast ../../contract
s/currency/currency.abi
```

As a response you should get a JSON with a transaction_id field. Your contract was successfully uploaded!

You can also verify that the code has been set with the following command:

```
./eosc get code currency
```

It will return something like:

```
code hash: 9b9db1a7940503a88535517049e64467a6e8f4e9e03af15e9968ec89dd794975
```

Next verify the currency contract has the proper initial balance:

```
./eosc get table currency currency account
{
  "rows": [{
     "key": "account",
     "balance": 1000000000
     }
  ],
  "more": false
}
```

## 5.Transfering funds with the sample "currency" contract

Anyone can send any message to any contract at any time, but the contracts may reject messages which are not given necessary permission. Messages are not sent "from" anyone, they are sent "with permission of" one or more accounts and permission levels. The following commands show a "transfer" message being sent to the "currency" contract.

The content of the message is `'{"from":"currency","to":"inita","quantity":50}'.` In this case we are asking the currency contract to transfer funds from itself to someone else. This requires the permission of the currency contract.

```
./eosc push message currency transfer '{"from":"currency","to":"inita","quantity":
50}' --scope currency,inita --permission currency@active
```

Below is a generalization that shows the currency account is only referenced once, to specify which contract to deliver the transfer message to.

```
./eosc push message currency transfer '{"from":"${usera}","to":"${userb}","quantit
y":50}' --scope ${usera},${userb} --permission ${usera}@active
```

We specify the `--scope` ... argument to give the currency contract read/write permission to those users so it can modify their balances. In a future release scope will be determined automatically.

As confirmation of a successfully submitted transaction, you will receive JSON output that includes a transaction_id field.

### 6.Reading sample "currency" contract balance

So now check the state of both of the accounts involved in the previous transaction.

```
./eosc get table inita currency account
{
  "rows": [{
      "key": "account",
      "balance": 50
       }
    ],
  "more": false
}
./eosc get table currency currency account
{
  "rows": [{
      "key": "account",
      "balance": 999999950
    }
  ],
  "more": false
}
```

As expected, the receiving account inita now has a balance of 50 tokens, and the sending account now has 50 less tokens than its initial supply.

## Running a local node connected to the public testnet

To run a local node connected to the public testnet operated by block.one, a script is provided.

```
cd ~/eos/build/scripts
./start_npnode.sh
```

This command will use the data folder provided for the instance called testnet_np.

You should see the following response:

Launched eosd. See testnet_np/stderr.txt for eosd output. Synching requires at least 8 minutes, depending on network conditions. To confirm eosd operation and synchronization:

```
tail -F testnet_np/stderr.txt
```

To exit tail, use Ctrl-C. During synchronization, you will see log messages similar to:

```
3439731ms            chain_plugin.cpp:272          accept_block         ] Syncing
Blockchain --- Got block: #200000 time: 2017-12-09T07:56:32 producer: initu
3454532ms            chain_plugin.cpp:272          accept_block         ] Syncing
Blockchain --- Got block: #210000 time: 2017-12-09T13:29:52 producer: initc
```

Synchronization is complete when you see log messages similar to:

```
42467ms            net_plugin.cpp:1245           start_sync           ] Catching u
p with chain, our last req is 351734, theirs is 351962 peer ip-10-160-11-116:9876
42792ms            chain_controller.cpp:208      _push_block          ] initt #351
947 @2017-12-12T22:59:44  | 0 trx, 0 pending, exectime_ms=0
42793ms            chain_controller.cpp:208      _push_block          ] inito #351
948 @2017-12-12T22:59:46  | 0 trx, 0 pending, exectime_ms=0
42793ms            chain_controller.cpp:208      _push_block          ] initd #351
949 @2017-12-12T22:59:48  | 0 trx, 0 pending, exectime_ms=0
```

This eosd instance listens on 127.0.0.1:8888 for http requests, on all interfaces at port 9877 for P2P requests, and includes the wallet plugins.

when you see log messages similar to:

```
514792ms net_plugin.cpp:2515 plugin_startup ] starting listener, max clients is 25
514802ms net_plugin.cpp:567 connection ] created connection to p2p-testnet1.eos.io
:9876
514802ms net_plugin.cpp:1284 connect ] host: p2p-testnet1.eos.io port: 9876
515718ms net_plugin.cpp:1628 handle_message ] received a go away message, reason =
 duplicate
515718ms net_plugin.cpp:1417 operator() ] Error reading message from p2p-testnet1.
eos.io:9876: Bad file descriptor
```

The current workaround is to edit your `config.ini` file and use something unique for `"p2p-server-address"` The suggestion is to use your **external IP** address. …

# Running multi-node local testnet

To run a local testnet you can use a launcher application provided in the `~/eos/build/programs/launcher` folder.

For testing purposes you will run two local production nodes talking to each other.

```
cd ~/eos/build
cp ../genesis.json ./
./programs/launcher/launcher -p2 --skip-signature
```

This command will generate two data folders for each instance of the node: tn*data*0 and tn*data*1.

You should see the following response:

```
spawning child, programs/eosd/eosd --skip-transaction-signatures --data-dir tn_dat
a_0
spawning child, programs/eosd/eosd --skip-transaction-signatures --data-dir tn_dat
a_1
```

To confirm the nodes are running, run the following eosc commands:

```
~/eos/build/programs/eosc
./eosc -p 8888 get info
./eosc -p 8889 get info
```

For each command, you should get a JSON response with blockchain information.