Testing and deployment of the Private Tangle

Foreword: The purpose of this document is to detail all the steps necessary to deploy a Private Tangle on AWS (and most likely, any cloud provider) and reproduce the performance tests provided under the Results directory.

1) AWS Instances

Several solutions exist to deploy a private Tangle on AWS, notably provided by the IOTA foundation (the “One-click Tangle”). However, to better reproduce the tests, we give the full procedure to create AWS instances running the same software as during our own tests.

During our tests, we were running at most 10 nodes simultaneously. To deploy these 10 nodes, it is relevant **to first prepare a functional instance**, then to generate **an AMI** to deploy all the other instances faster. To deploy the first instance:

- Go to the panel EC2 → Instances → Launch an instance
- Choose the free-tier eligible Ubuntu 22.04 TLS SSD Volume type
- Create a new security group
- Keep the default 8GiB storage gp2 storage

Once the instance is deployed, you need to modify the **security group** to allow **the UCS to connect** to the IOTA node. To do that:

- Go to EC2 → Security Groups → *Your security group* → Edit inbound rules
- Click Add rule (bottom left corner), Type: All traffic, Source: My IP
- Click Save rules (bottom right corner)

You should now be able **to connect to the first instance**. To do so, we will use the dedicated instance connect, but SSH is also a convenient way:

- Go to EC2 → Instances → Connect to instance
- Choose ubuntu as the user name and click Connect (bottom right corner)

You need to install software on the AWS instance to be able to run a IOTA node:

- You will need **Go** installed on the instance, using the installation procedure in Figure 1. **The version used for the tests is the version 1.18.5**;
- Install **Hornet** to run the IOTA node. To install a specific version, it is necessary to use the *build from source* method (cf. Figure 2). **Make sure you are using the 1.2.1 release**, as we did our tests on this version.

At this point, you should be able to start the nodes using the hornet/private/run_xxx.sh scripts. However, the scripts are configured to run the nodes locally i.e. all the nodes on the instance. Therefore, you need to configure the scripts manually, in order to run only one node on each AWS instance.

2) IOTA nodes

2)a) Configuration file

For better stability, it is better to have at least three neighbors (to avoid being disconnected from the network) and at most eight (to avoid communication overheads). To do so, copy any run_xxx.sh script file and modify the following lines:

--p2p.bindMultiAddresses=”/ip4/$*ipaddress*/tcp/$*port*” where $*ipaddress* is the address of the AWS instance on which the node is running, and $*port* is the port used for the peering. To better differentiate the nodes in the configuration file, I suggest using the port number 600X where X is the number of the node (from 1 to 10 in our tests)

--profiling.bindAddress=”$*ipaddress/$port”* with the same IP and port as above
--p2p.peers=”/ip4/$*peerIPaddress*/tcp/$*peerPort*/p2p/$*peerId*

This must be done for each peer node, using the IP address of their respective AWS instance and the port exposed in their configuration file. Note that this step must be done manually as **public** IP addresses change with each start. The procedure to get the $*peerId* is described in the following.

2)b) Node identity
Each node has a identity and a set of keys (public/private) from which the peer Id is derived. Each node needs to generate this set of keys running the command : ***hornet tools p2identity-gen***
which will return the following information:
1)p2p private key
2)p2p public key (hex)
3)p2p public key (base58)
4)The **PeerId**

It is highly recommended to keep the track of all peerId generated in a dedicated file, as they are needed for peering. Besides, the public key (3) in base58 must be stored in the p2pstore/identity.key file as:

```
-----BEGIN PRIVATE KEY-----
MC4CAQAwBQYDK2VwBCIEIB9G+tT1OKAx1Ph/SQ9rykMZ39AwdjaldZoiteiHS9YI
-----END PRIVATE KEY-----
```

The peerId will be compared against this file when peering, and peering will not work if this step is not completed.

At the end of this step, nodes on AWS instances should be able to connect to one another.

2)3) Starting the private Tangle

As an example, we will consider the three nodes configuration, with two AWS instances. Each node is configured according to the above-mentioned instructions, so that all the nodes are peered together. The node running on the local machine (your own PC) should run a copy a **modified version of the run_coo.sh script** as it will be running the Coordinator node, instead of a generic run_xxx.sh script, but the instructions remains the same. **Start the run_coo_bootstrap.sh each time you restart the Tangle for testing**, it will erase the current state of the ledger and will enable the synchronisation with new nodes. You also need to modify the

If you run ./run_coo.sh and run_xxx.sh on the two AWS instances, the three nodes should be peering with one another at this step. To check this, you can open a web browser, go on localhost:8081 (check the –dashboard.bindAddress if it does not work, and change the port after localhost in that case). Connect using the default admin/admin credentials and go to “Peer” panel. Both AWS nodes should appear on this panel and be green after a few minutes (synced and healthy).

Now that the private Tangle is deployed and working, we need to do some modifications to enable the interaction with the UCS.

2)4) Distributing IOTAs

In our tests, we measured the transaction time, which requires iotas available on an address. To do that, a **genesis transaction** is required, where all the funds of the network are sent on an address. This is done by running the script */hornet/private_tangle/create_snapshot_private_tangle.sh*.
You can specify the address that will receive all the funds **under the ED25519** form. In our case, we specified a buyer address for the tests for convenience. The line to modify (on every node) is :

go run ../main.go tool snap-gen --networkID private_tangle1 **--mintAddress 60200bad8137a704216e84f8f9acfe65b972d9f4155becb4815282b03cef99fe** --outputPath snapshots/private_tangle1/full_snapshot.bin

Now, you should have a fully working private Tangle, ready to be used for testing. In the next part, we will focus on the usage control system, and how to conduct the tests.

3) Usage control and tests

The code of the usage control and the tests are the main components of the code on the repository. The code can be
compile using **mvn compile** at the root (i.e. where the .pom file is located) and run using the command line:

mvn exec:java -Dexec.mainClass="iotawucon.App" -e

However, there are a few changes to do in the code so that the code can work on your device.

3)1) Adjusting variables

A configuration file at the root of the project (configuration.properties) is used to define some variables. In particular, the **Ip address of the remote node** for the tests. You need to specify the IP address of a node of the private Tangle (on AWS) to do the remote node testing. The path to write **the results of the different tests** is also specified in the configuration file.
Two settings are available for the tests, either *REMOTE* or *LOCAL*. The remote setting uses a distant AWS node (or a public devnet node, using the Chrysalis URL). The local setting uses the node of the same device running the UCS and the Hornet node simultaneously. The local setting corresponds to the integrated setting in the article. The setting can be picked directly in the main function.

3)2) Rust bindings

You need to re-generate the code for interacting with the IOTA node. For Java, **it is derived from Rust bindings**. The full procedure is (at the moment of writing) detailed in the following link:

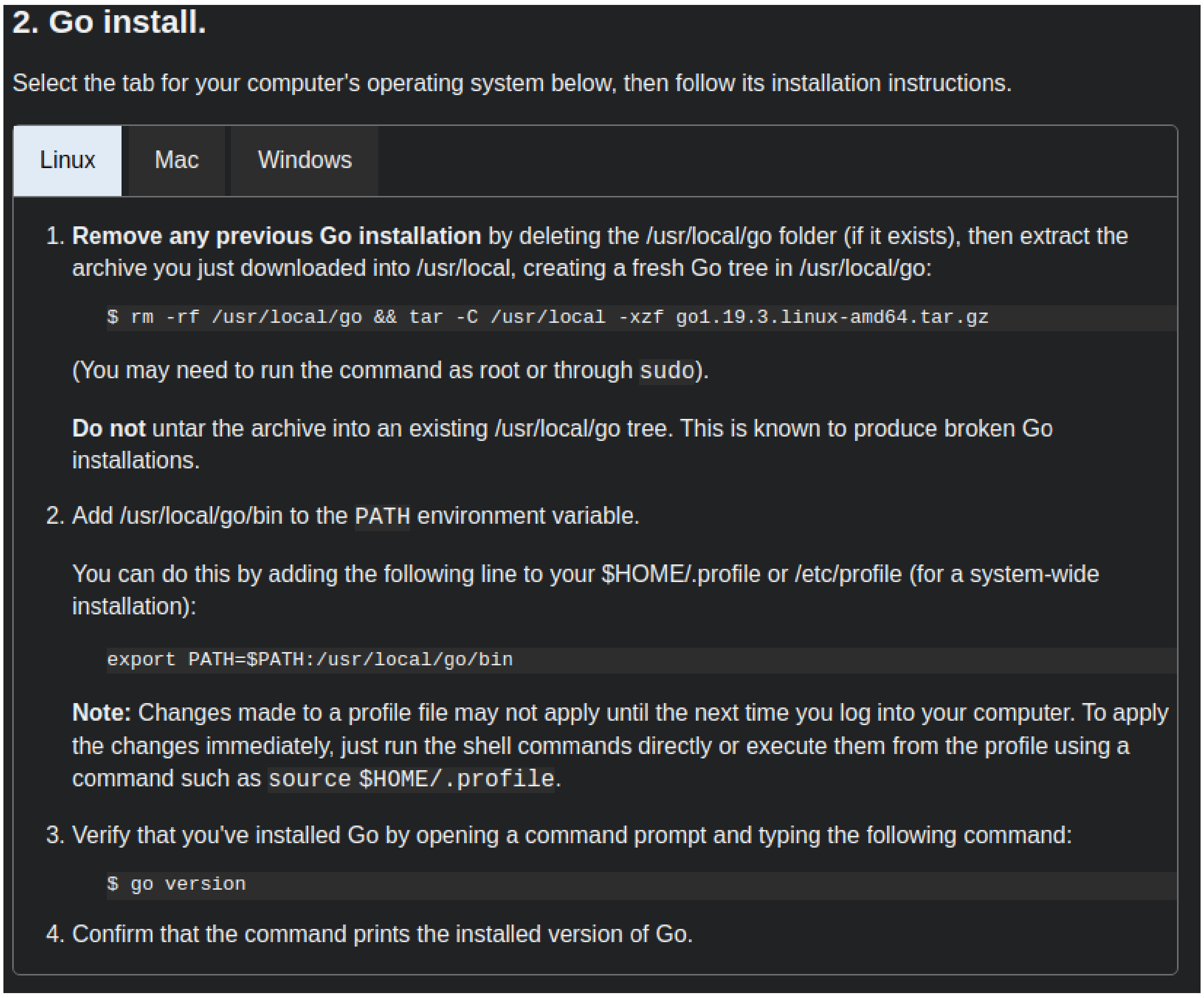
https://github.com/iotaledger/iota.rs/tree/production/bindings/java

If the procedure is successful, you should be able to generate a .so file that you need to import using the `System.loadLibrary("iota_client"); call.`

The generation of the .so file can be a tricky part, but once the .so library is generated and added, you should be able to start the tests.

3)2) Starting the tests

The main class is already configured to start the “**remote transaction time**” testing. Just run the code after the above configuration and you should get the results inside the URLRESOURCE/results/transactions_times_remote.csv, for the transaction time etc.
Change REMOTE to LOCAL to do the integrated testing, and uncomment remoteTransactionsOverhead to measure the time needed for **fetchTransaction** (as named in the article)



2. Go install.

Select the tab for your computer's operating system below, then follow its installation instructions.

Linux | Mac | Windows

1. **Remove any previous Go installation** by deleting the /usr/local/go folder (if it exists), then extract the archive you just downloaded into /usr/local, creating a fresh Go tree in /usr/local/go:

   ```
   $ rm -rf /usr/local/go && tar -C /usr/local -xzf go1.19.3.linux-amd64.tar.gz
   ```

   (You may need to run the command as root or through `sudo`).

   Do not untar the archive into an existing /usr/local/go tree. This is known to produce broken Go installations.

2. Add /usr/local/go/bin to the `PATH` environment variable.

   You can do this by adding the following line to your $HOME/.profile or /etc/profile (for a system-wide installation):

   ```
   export PATH=$PATH:/usr/local/go/bin
   ```

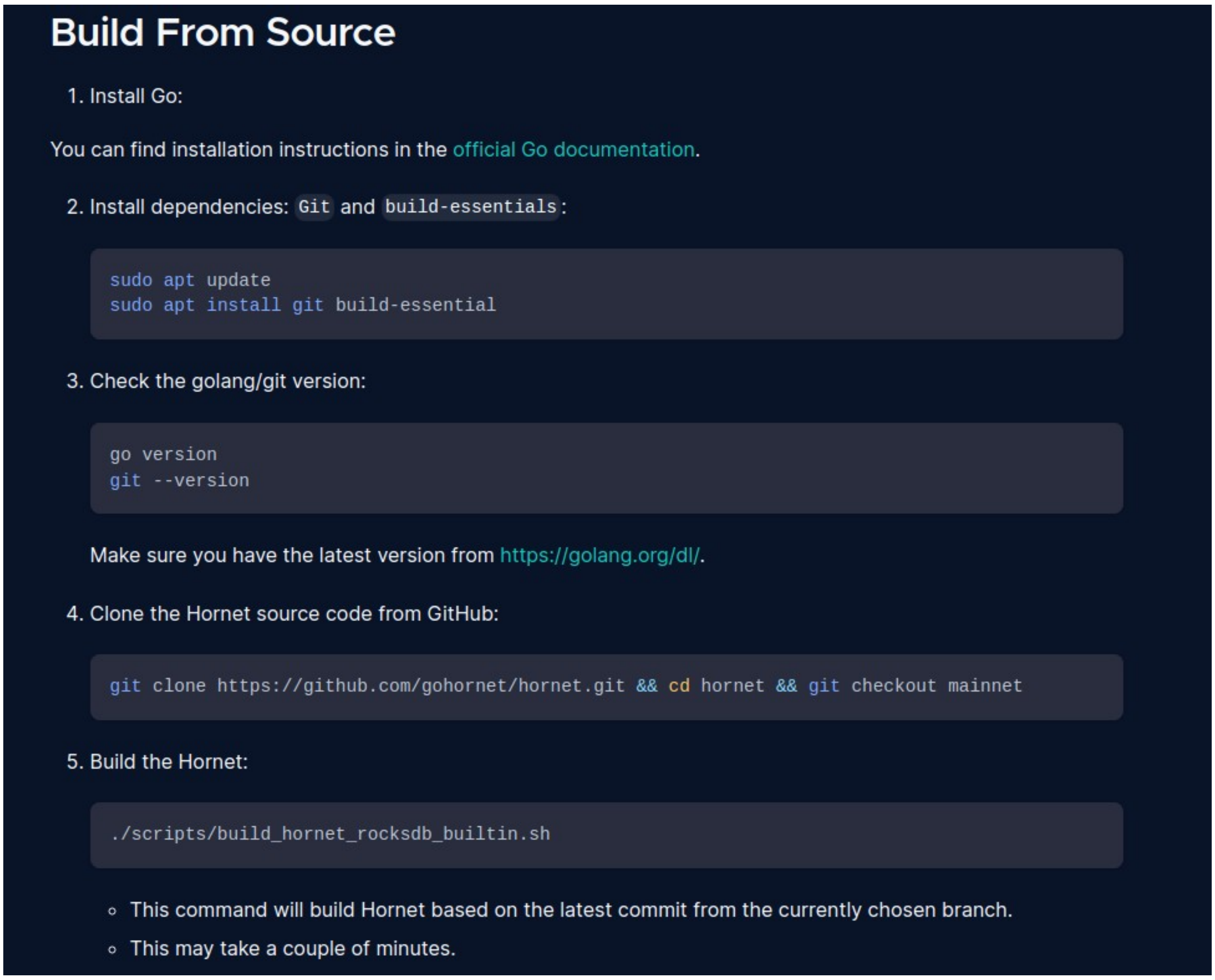
   Note: Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as `source $HOME/.profile`.

3. Verify that you've installed Go by opening a command prompt and typing the following command:

   ```
   $ go version
   ```

4. Confirm that the command prints the installed version of Go.

Figure 1: Go install



Build From Source

1. Install Go:

You can find installation instructions in the official Go documentation.

2. Install dependencies: `Git` and `build-essentials`:

```
sudo apt update
sudo apt install git build-essential
```

3. Check the golang/git version:

```
go version
git --version
```

Make sure you have the latest version from https://golang.org/dl/.

4. Clone the Hornet source code from GitHub:

```
git clone https://github.com/gohornet/hornet.git && cd hornet && git checkout mainnet
```

5. Build the Hornet:

```
./scripts/build_hornet_rocksdb_builtin.sh
```

- This command will build Hornet based on the latest commit from the currently chosen branch.
- This may take a couple of minutes.

Figure 2: Hornet install