CRIANDO SUA REDE ETHEREUM PRIVADA COM DOCKER

Autor:

ANDRÉ FERNANDES

CTIO da Vertigo Tecnologia
Docker Certified Associate (DCA)
Eng. de Computação pela Unicamp com pós-graduação em Gestão de Negócios.

VERTIGO TECNOLOGIA

www.vertigo.com.br

SUMÁRIO

Criando sua rede Ethereum Privada com Docker

1. INTRODUÇÃO

Olá! Legal que você baixou o nosso material. Se você já leu a parte 1 no nosso blog, clique aqui e pule para a parte dois.

Aqui iremos cobrir tópicos como o uso das imagens Docker oficiais ("ethereum/go-ethereum"), o uso local da Wallet (carteira), provisionamento de nós Ethereum em nuvens públicas e o deployment de uma aplicação simples.

Docker e Blockchain são assuntos chave aqui. Ambas estas tecnologias têm estado em foco intenso nos últimos anos. Se você ainda não sabe sobre esses dois assuntos recomendamos os dois conteúdos abaixo:

- O que é Docker
- Descubra o que é Blockchain

Docker já está bem além do "hype" e se tornou o padrão de facto para containers, assim como o uso de containers é quase onipresente no fluxo de trabalho e esteiras de entrega de desenvolvimento atualmente (exceto para quem se escondeu em uma caverna).

Blockchain, por outro lado, ainda não se popularizou como solução em diversos setores para os quais teve suas maravilhas prometidas. Além de criptomoedas (como Bitcoin), a tecnologia blockchain tem aplicabilidade genérica em vários cenários (títulos de propriedade - digital ou não, votações, gestão de identidade e privacidade, entre tantos outros). Do ponto de vista de um empreendedor, uma rede blockchain moderna pode ser vista como uma plataforma sobre a qual todo um novo conjunto de produtos pode ser construído.

Aviso: As páginas a seguir assumem alguma familiaridade com conceitos básicos de tecnologia blockchain, não se demorando em explicá-los em nenhum momento.

SOBRE ETHEREUM

A Rede Ethereum foi concebida para entregar o conceito de plataforma, extrapolando o conceito de moeda e transações do Bitcoin. Conforme a própria documentação da Ethereum:

"Ethereum é um blockchain aberto que permite a qualquer um construir e usar aplicações descentralizadas sobre esta tecnologia."

Então, vista como uma plataforma para aplicações descentralizadas, a Rede Ethereum apresenta "...níveis extremos de tolerância a falhas, garante disponibilidade (zero 'downtime') e assegura a inviolabilidade, imutabilidade e não-censura dos dados armazenados."

MAS É "A REDE" OU "UMA REDE"?

Esta pergunta significa: existe uma única rede Ethereum ("a" rede) ou uma infinidade de redes? Bem, as duas coisas estão corretas.

Existe, sim, "a" Rede Ethereum, pública e descentralizada, não controlada por ninguém em particular. Esta rede Ethereum principal (também conhecida como "mainnet") é aquela de que falamos normalmente quando dizemos "Rede Ethereum".

Existem também a rede "irmã" da mainnet chamada testnet, onde testes podem ser conduzidos sem comprometer recursos reais. Existe também um fork da rede Ethereum conhecido como Ethereum Classic, mas isto é outra história.

Enquanto em uma plataforma de nuvem tradicional, na qual você é "dono" dos nós (controla e confia) onde suas aplicações executam, em um blockchain público tem-se o oposto: ele é inteiramente constituído por participantes nos quais você não confia, mas que constroem um consenso de uma maneira confiável (eis o "milagre" do blockchain). O significado de um blockchain ser público é este: qualquer um pode ler, transacionar e participar da busca pelo consenso (basta, por exemplo, usar uma Wallet) - na descentralização é que reside o seu valor.

Isto é um pouco duro de entender no começo: ninguém controla todos os nós da rede, mas ao mesmo tempo os desenvolvedores precisam evoluir o software e seu protocolo - ou seja, o software é atualizado de tempos em tempos porque todos concordam em fazê-lo. Entende-se assim os conceitos de "soft fork" e "hard fork", mas todo o propósito é sempre preservar o valor intrínseco da descentralização enquanto se evolui o software executado em todos os nós.

Sendo Ethereum um software aberto (open-source) não há nada que impeça qualquer um de criar uma nova rede pública - apenas não faz sentido fazê-lo. Do ponto de vista técnico cada rede blockchain diverge essencialmente de outra pelo seu "bloco gênese" (ainda que com o mesmos algoritmos e protocolos) - mas o valor de uma rede está em seu uso disseminado (i.e. descentralização).

Redes blockchain privadas, por outro lado, podem ser instanciadas com qualquer propósito que venha à mente (há uma grande quantidade de cenários plausíveis). Repetindo, basta definir um "bloco gênese", disparar seus nós e colocá-los para trabalhar.

É fácil encontrar na internet discussões acaloradas sobre fazer sentido ou não o uso de redes blockchain privadas como soluções de negócio (por que usar uma solução descentralizada e ineficiente se você confia nos participantes), mas do ponto de vista dos desenvolvedores de soluções é bastante desejável ser capaz de criar e destruir por completo seus cenários - várias vezes por dia. É para este caso que este material foi escrito.

DOCKER DOCKER DOCKER

Sim, sou um fanboy. Como não ser?

A próxima parte irá mostrar como levantar uma rede Ethereum privada - localmente ou em nuvem - usando containers. Embora existam várias ofertas dos vendors de nuvens públicas para provisionamento de blockchains privados (como o Azure's Blockchain as a Service), o caminho via containers é bem mais sensato do ponto de vista do desenvolvedor. No tempo em que um conjunto de VMs neste modelo é provisionado (mesmo com automação) é possível criar e recriar um swarm de containers repetidas vezes.

2. A IMAGEM DOCKER

Iremos cobrir tópicos como o uso das imagens Docker oficiais ("ethereum/go-ethereum"), o uso local da Wallet (carteira), provisionamento de nós Ethereum em nuvens públicas e o deployment de uma aplicação simples.

Este artigo usa a imagem Docker oficial "ethereum/client-go" para instanciar vários nós de rede Ethereum localmente, de forma segura e descartável. Assumimos, portanto, que o leitor já possui um Docker Engine disponível (localmente ou em nuvem) e já tem alguma intimidade com seus comandos. A maneira mais fácil de instalar o Docker em uma estação de trabalho/notebook é o Docker Desktop (também conhecido como "Docker for Mac" ou "Docker for Windows").

Os fontes e scripts mencionados no artigo estão em repositório público do Github, em https://github.com/vertigobr/ethereum.git.

Há uma imagem oficial "ethereum/client-go" no Docker Hub que usaremos como base para o trabalho. Iremos rodar alguns scripts com funcionalidades básicas e opções de configuração para simplificar as tarefas.

Esta imagem oficial é bastante útil, completa e versátil. Ela pode ser usada, por exemplo, para executar um nó participante da mainnet Ethereum com um simples comando:

```
docker run -d --name ethereum-node \-v $HOME/.ethereum:/root \-p 8545:8545 -p
30303:30303 \ethereum/client-go:v1.8.12 --fast --cache=512
```

Para executar um nó na testnet o comando é bem semelhante:

```
docker run -d --name ethereum-node \
  -v $HOME/.ethereum:/root \
  -p 8545:8545 -p 30303:30303 \
  ethereum/client-go:v1.8.12 --testnet --fast --cache=512
```

Para checar os logs de execução do nó basta inspecionar o log do container:

```
docker logs -f ethereum-node
```

O container instancia um nó Ethereum rodando o cliente "geth" padrão. Para conectar-se (attach) com o "geth" do container de mainnet basta o comando abaixo:

```
docker exec -ti ethereum-node geth attach
```

Ou, analogamente, para conectar-se (attach) com o "geth" do container de testnet:

```
docker exec -ti ethereum-node \
  geth attach ipc:/root/.ethereum/testnet/geth.ipc
```

Mas, é claro, não faremos nada disso. Não queremos mainnet ou testnet, queremos criar nossa própria rede privada.

PASSOS INICIAIS

O projeto Github com os scripts que aqui usamos pode ser baixado via Git:

```
git clone https://github.com/vertigobr/ethereum.git
```

Assumindo que o Docker já está instalado e funcionando, basta executar os passos a seguir:

- Executar um bootnode
- Executar um nó comum
- Executar um nó minerador
- Checar os pares de um nó

Isto se traduz na execução dos comandos abaixo (na ordem exata):

```
./genesis.sh    # bloco gênese, rodar apenas uma vez
./bootnode.sh
./runnode.sh
./runminer.sh
./showpeers.sh  # a aquisição de pares pode demorar um pouco
```

O primeiro passo é a definição do "genesis block" do blockchain, na forma do arquivo "genesis.json". O script "genesis.sh" faz isto facilmente, podendo ser editado para definir valores específicos em suas variáveis.

Compartilhar o bloco gênese do blockchain é fundamental para que qualquer nó possa validar completamente o blockchain. Em nosso exemplo, os scripts montam o mesmo arquivo em cada container, garantindo que os nós possam todos participar da mesma rede. Uma rede Ethereum privada é simplesmente isto tecnicamente: nós que possuem conectividade, são capazes de "se encontrar" e que compartilham o mesmo bloco gênese.

Os scripts deste projeto incluem o "bootnode.sh" que executa um elemento importante na rede: o bootnode. Um bootnode é um ponto de contato para um nó recém-executado ser capaz de encontrar seus pares, funcionando como um catálogo dinâmico de nós em execução. A rede Ethereum pública tem seus bootnodes pré-definidos, a nossa rede privada também precisa do seu.

Além disto, os scripts também montam pastas locais para cada container persistir seus caches, de forma que se os containers forem destruídos e recriados irão recomeçar seu trabalho do mesmo ponto (ex: não precisarão sincronizar todo o blockchain novamente).

Os Scripts

O projeto clonado a partir do Github possui os seguintes scripts auxiliares prontos para uso, poupando o trabalho de escrever longas linhas de comando Docker:

- **genesis.sh:** cria um novo arquivo "genesis.json" a partir do modelo em "src/genesis.json.template" e baseado em algumas variáveis que podem ser modificadas para criar redes distintas;
- **bootnode.sh:** executa um bootnode Ethereum em um container chamado "ethereum--bootnode";
- **runnode.sh:** executa um nó validador (i.e. não-minerador) em um container com um node escolhido (ex: "./runnode.sh meunode" roda um container chamado "ethereum--meunode"), sendo "node1" o argumento padrão caso seja omitido;
- **runminer.sh:** idêntico a "runnode.sh", mas executa um nó minerador (argumento padrão é "miner1");
- **showpeers.sh:** enumera todos os pares conectados ao nó do container. Exemplo: "./showpeers.sh miner1";
- **killall.sh:** destrói ("docker stop" e "docker rm") os containers executados pelos scripts, mas preserva as pastas montadas como cache;
- **wipeall.sh:** idêntico a "killall.sh", mas também remove os caches.

Todos os scripts utilizam o mesmo "genesis.json" montado localmente. Como ambos o bloco gênese e bootnode são exclusivos, não há nenhuma chance de sua rede conectar-se a outra rede Ethereum.

Alguns scripts merecem maior explicação:

bootnode.sh
Este script faz o seguinte trabalho (reduzido por simplicidade):

```
docker stop ethereum-bootnode
docker rm ethereum-bootnode
docker run -d --name ethereum-bootnode \
 -v $(pwd)/.bootnode:/opt/bootnode \
 ethereum/client-go:alltools-v1.8.12 bootnode --nodekey (...)
```

Este script executa um nó "burro" e não um nó real na rede. Um bootnode é apenas um ponto inicial de conexão para que nós plenos ("full") recém-executados possam encontrar seus pares. A exemplo dos outros scripts ele monta seu cache em um volume externo (pasta ".bootnode").

runnode.sh
Este script faz o seguinte trabalho (reduzido por simplicidade):

```
NODE_NAME=$1
CONTAINER_NAME="ethereum-$NODE_NAME"
docker stop $CONTAINER_NAME
docker rm $CONTAINER_NAME
BOOTNODE_URL=$(./getbootnodeurl.sh)
docker run -d --name $CONTAINER_NAME \
  -v $DATA_ROOT:/root/.ethereum \
  -v $DATA_HASH:/root/.ethash \
  -v $(pwd)/genesis.json:/opt/genesis.json \
  ethereum/client-go:v1.8.12 --cache=512 --bootnodes=(...)
```

O script "getbootnode.sh", chamado internamente, serve para obter dinamicamente uma URL válida para chegar ao bootnode executado anteriormente. Os volumes montados para os caches também são únicos para cada container.

showpeers.sh
Este script faz o seguinte trabalho (reduzido por simplicidade):

```
CONTAINER_NAME="ethereum-$NODE"
docker exec -ti "$CONTAINER_NAME" geth --exec 'admin.peers' attach
```

Este comando se conecta ao "geth" (cliente Ethereum) de um container e lista os pares a ele conectados.

VAMOS LÁ!

Repetindo, estes serão os passos que seguiremos:

1. Criar a rede Docker e executar o bootnode;
2. Executar um nó validador (não-minerador);
3. Executar um outro nó validador;
4. Verificar se ambos encontraram seus pares (i.e. um ao outro);
5. Executar um nó minerador;
6. Verificar se todos encontraram seus outros dois pares.

Estes passos são triviais com o uso dos scripts fornecidos. Com tempo é possível, obviamente, estudar seu funcionamento, mas este artigo não irá se demorar nisto.

Passo 1: Criar rede e executar o bootnode

O script faz isso sozinho:

```
./bootnode.sh
```

O log deste container pode ser lido com o comando a seguir:

```
docker logs ethereum-bootnode
```

O resultado será um trecho como o abaixo:

```
INFO [12-06|17:31:44] UDP listener up
self=enode://d92e0ce7786191.....https://www.linkedin.com/redir/invalid=-link-page?url-
cb240726078439e3%40%5B%3A%3A%5D%3A30301
```

Esta saída de log é usada em "getbootnodeurl.sh" para obter dinamicamente uma URL válida para o bootnode.

```
./getbootnodeurl.sh
enode://3fa3af4.....c565fcc@172.18.0.2:30301
```

Este é um truque de "baixo nível" que torna o uso dos demais scripts ainda mais simples.

Passo 2: Executar um nó validador

Bastante simples:

```
./runnode.sh node1
```

O log deste container é visto com o comando a seguir:

```
docker logs ethereum-node1
```

O resultado será semelhante ao abaixo:

```
INFO [12-06|17:38:34] Starting peer-to-peer node
instance=Geth/v1.8.12-stable/linux-amd64/go1.10.3
(...)
INFO [12-06|17:38:34] Starting P2P networking
INFO [12-06|17:38:34] UDP listener up
self=enode://454bc00ecd1.....https://www.linkedin.com/redir/invalid-link-page?url=e5d-
d0cc4741a239%40%5B%3A%3A%5D%3A30303
INFO [12-06|17:38:36] RLPx listener up
self=enode://454bc00e461.....https://www.linkedin.com/redir/invalid-link-page?url=e5d-
d0cc4741a239%40%5B%3A%3A%5D%3A30303
INFO [12-06|17:38:36] IPC endpoint opened: /root/.ethereum/geth.ipc
(...)
```

Este é um nó solitário, que de tempos em tempos comunica-se novamente com o bootnode em busca de novos pares. Como não há nenhuma mineração, o blockchain é essencialmente inútil.

É possível alterar a verbosidade do log editando runnode.sh para qualquer valor entre 0 (silencioso) e 6 (detalhado). O padrão é 4 (mensagens principais).

Passo 3: Executar outro nó validador

Igualmente simples rodar um novo container validador:

```
./runnode.sh node2
```

É fácil checar os logs de ambos os containers para verificar que, via comunicação com o bootnode, ambos os nós se encontram e passam a se comunicar diretamente:

```
DEBUG[07-12|01:10:45.194] Adding p2p peer
name=Geth/v1.8.12-stable/...addr=https://www.linkedin.com/redir/invalid=-link-page?url-
172%2e18%2e0%2e3:30303 peers=1
DEBUG[07-12|01:10:45.196] Ethereum peer connected
id=684d4388205cb179
 conn=dyndial name=Geth/v1.8.12-stable/linux-amd64/go1.10.3
```

Passo 4: Verificar se nós acharam seus pares

Bem mais fácil que vasculhar logs é usar o script "showpeers.sh" criado para enumerar os pares de um nó:

```
./showpeers.sh node1
```

Este comando retorna um array JSON com a lista de pares deste nó (no caso, o “node2”):

```
[{
  caps: ["eth/63"],
  id: "3e48d35c5ccdb.....fcc5fa685eeb72e86",
  name: "Geth/v1.8.12-stable/linux-amd64/go1.10.3",
  network: {
    inbound: true,
    localAddress: "https://www.linkedin.com/redir/invalid=-link-page?url-
172%2e18%2e0%2e3%3A30303",
    remoteAddress: "https://www.linkedin.com/redir/invalid=-link-page?url-
172%2e18%2e0%2e4%3A42052",
    static: false,
    trusted: false
  },
  protocols: {
    eth: {
      difficulty: 131072,
      head: "0x2084e6ce93b0.....c49e1a00753052b987",
      version: 63
    }
  }
}]
```

Um comando semelhante mostrará os pares de “node2” (ou seja, o “node1”):

```
./showpeers.sh node2
```

Parabéns! Você já tem uma rede Ethereum privada de múltiplos nós (embora ainda inútil sem a mineração).

Passo 5: Executar um nó minerador

Temos também um script para auxiliar a execução de um nó minerador:

```
./runminer.sh miner1
```

Na primeira execução um tempo longo irá transcorrer antes da mineração começar. Esta progressão pode ser verificada nos logs deste nó:

```
docker logs -f ethereum-miner1
...
INFO [12-06|18:08:53] Generating DAG in progress
epoch=0 percentage=0 elapsed=3.893s
INFO [12-06|18:08:57] Generating DAG in progress
epoch=0 percentage=1 elapsed=7.695s
...
```

Este mecanismo (geração do grafo DAG) tem a intenção de ser um componente oneroso em recursos de forma a combater a concentração de nós em farms especializadas (ASIC--resistant). A mineração só começará quando a geração do DAG for concluída, portanto é bom ter um pouco de paciência.

Concluída a geração do DAG teremos logs reportando mineração bem sucedida:

```
...
INFO [07-12|13:47:39.477] Successfully sealed new block
number=3 hash=69781b…4697c9
DEBUG[07-12|13:47:39.518] Trie cache stats after commit
misses=0 unloads=0
INFO [07-12|13:47:39.529] ⛏ mined potential block
number=3 hash=69781b…4697c9
INFO [07-12|13:47:39.532] Commit new mining work
number=4 txs=0 uncles=0 elapsed=2.937ms
...
```

Os nós validadores receberão cópias de cada bloco minerado, como esperado:

```
docker logs ethereum-node1
...
DEBUG[07-12|13:52:58.097] Queued propagated block
peer=4691ae7cb9f9ff84 number=4 hash=129832…1236df queued=1
DEBUG[07-12|13:52:58.097] Importing propagated block
peer=4691ae7cb9f9ff84 number=4 hash=129832…1236df
DEBUG[07-12|13:52:58.534] Trie cache stats after commit
misses=0 unloads=0
DEBUG[07-12|13:52:58.542] Inserted new block
number=4 hash=129832…1236df uncles=0 txs=0 gas=0 elapsed=16.446ms
...
```

Passo 6: Verificar se todos encontraram seus pares

Se os blocos estão sendo propagados é certo que os pares se encontraram. Mesmo assim, com três nós rodando cada um dos nós deve ter dois pares:

```
./showpeers.sh miner1
...
```

DE NOVO!

Para desligar e destruir os containers há um script preparado:

```
./killall.sh
```

Reexecutar os scripts anteriores para levantar novos containers com os mesmos argumentos irá “ressuscitar” a sua rede (uma vez que os volumes montados estão intactos):

```
./bootnode.sh
./runnode.sh node1
./runnode.sh node2
./runminer.sh miner1
```

Os blocos minerados anteriormente ainda estão ali, portanto destruir os containers não irá desfazer qualquer trabalho ou publicação anterior.

Se, contudo, o desejo for realmente começar “do zero”, cabe usar outro script para destruir tanto containers quanto volumes (cache):

```
./wipeall.sh
```

Boa sorte!

3. A CARTEIRA

Podemos, no momento, considerar a Carteira (Wallet) como um mero front-end pronto para o uso para os nossos experimentos. Contudo, além de ser uma interface para criar e transacionar com contas de ETH em uma rede Ethereum, a Carteira é "...a porta para aplicações descentralizadas no blockchain".

Quando executada em sua configuração padrão a Carteira é também um nó pleno (full node) da rede Ethereum pública (mainnet ou testnet) - o que não serve em nosso caso. Com algumas modificações podemos fazer com que ela se conecte a nossa rede privada.

Instalando a Carteira

A Carteira é um software aberto, disponível no Github em https://github.com/ethereum/mist/releases ou diretamente no site da Ethereum Foundation. Há instaladores para diversas plataformas (OSX, Windows e Linux), além de uma distribuição genérica em ZIP.

Importante: no presente momento a Ethereum Wallet 0.11 não funciona bem com estes exemplos. Recomendo instalar a versão 0.10.

Nós não iremos executar a Carteira como um nó da rede pública, então precisamos executá-la a partir da linha de comando para poder fornecer argumentos adicionais. A localização do arquivo executável da Carteira é particular de cada plataforma, mas os argumentos possíveis podem ser listados executando a Carteira com o parâmetro "--help".

No Windows o executável estará no local escolhido durante a instalação:

```
D:\Ethereum-Wallet\Ethereum-Wallet.exe --help
```

No Linux, analogamente, estará na pasta escolhida na instalação:

```
/opt/Ethereum-Wallet/Ethereum-Wallet --help
```

No OSX o local é um pouco mais obscuro, dentro de "Applications":

```
/Applications/Ethereum\ Wallet.app/Contents/MacOS/Ethereum\ Wallet --help
```

CONECTANDO A CARTEIRA AO MINERADOR

Iremos executar o nó minerador com uma configuração específica que permitirá a conexão da Carteira.

Habilitando a interface JSON-RPC

O comportamento padrão da Carteira é executar um nó local conectado a uma das redes

públicas (mainnet ou testnet). A conexão da Carteira com este nó é via socket IPC cujo caminho é conhecido (ou named pipe em Windows). Este comportamento padrão é inconveniente para nosso cenário, no qual o nó reside dentro de um container (em máquinas Windows e OSX o container não compartilha o kernel do S.O. da máquina). Precisamos executar a Carteira sem que esta execute um nó e forçando a conexão com um nó minerador que já rodamos antes.

Importante compreender que o Docker Desktop (também conhecido como Docker for Mac/Windows) roda o Docker Engine dentro de uma VM leve e "invisível" ao usuário (virtualização Hyper-V ou xhyve), enquanto a Carteira roda nativamente no S.O. local. Neste caso o socket IPC é inútil.

A solução é habilitar a interface JSON-RPC em um dos nós (protocolo HTTP na porta 8545). Esta porta representaria uma exposição indesejada em blockchains de produção, mas atende perfeitamente a nossa necessidade local em uma rede privada que é essencialmente efêmera.

Felizmente nossos scripts já facilitam a exposição desta porta pelo uso de uma variável de ambiente que seleciona em qual porta a interface estará aberta:

```
./bootnode.sh    # se bootnode não estiver em execução ainda
RPC_PORT=8545 ./runminer.sh miner1
```

Lembre-se que a geração do DAG sempre demora bastante.

A mágica deste script é simples: o nó é executado expondo a porta da interface JSON-RPC, a qual é também exposta pelo container para o host, na porta escolhida em RPC_PORT.

O *binding* da porta do container é feito pelos argumentos da linha de comando Docker:

```
-p $RPC_PORT:8545
```

Já os parâmetros do nó que expõem a interface são os abaixo:

```
--rpc --rpcaddr=0.0.0.0 --rpcapi="db,eth,net,web3,personal" \
  --rpccorsdomain "*"
```

Para testar se a interface está exposta e funcional uma simples linha de comando pode ser usada:

```
$ curl -X POST -H "Content-Type: application/json" \
   --data '{"jsonrpc":"2.0","method":"web3_clientVersion","params":[],"id":67}' \
   localhost:8545
{"jsonrpc":"2.0","id":67,"result":"Geth/v1.8.12-stable/linux-amd64/go1.10.3"}
```

Nota: na ausência do "curl" em seu prompt uma ferramenta como o Postman pode ser usada para fabricar requests HTTP.

Note que o Docker Desktop faz o NAT das portas expostas nos containers para "localhost" em sua estação de trabalho Windows ou OSX. Em instalações diferentes (Docker Toolbox, "docker-machine" ou VMs Vagrant customizadas) o mais provável é que um IP "host-only" deva substituir "localhost".

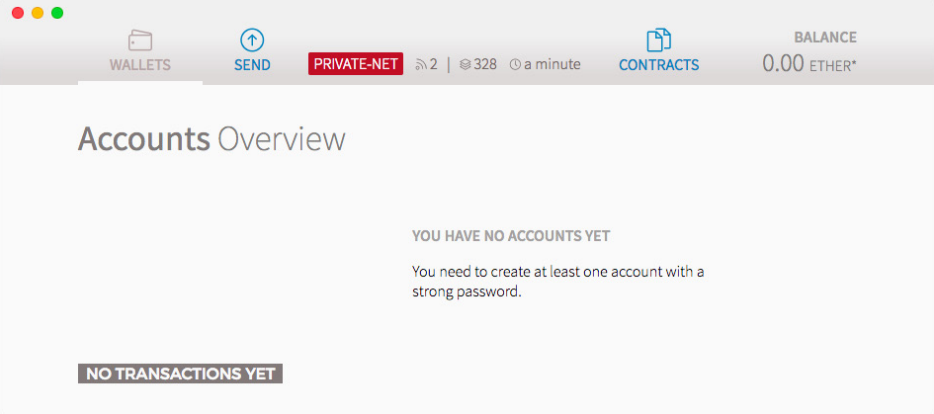
Rodando a Carteira

Com a porta JSON-RPC exposta podemos executar a Carteira com os argumentos corretos ("--rpc http://localhost:8545") na linha de comando.

Para o OSX:

```
/Applications/Ethereum\ Wallet.app/Contents/MacOS/Ethereum\ Wallet \
  --rpc http://localhost:8545
```

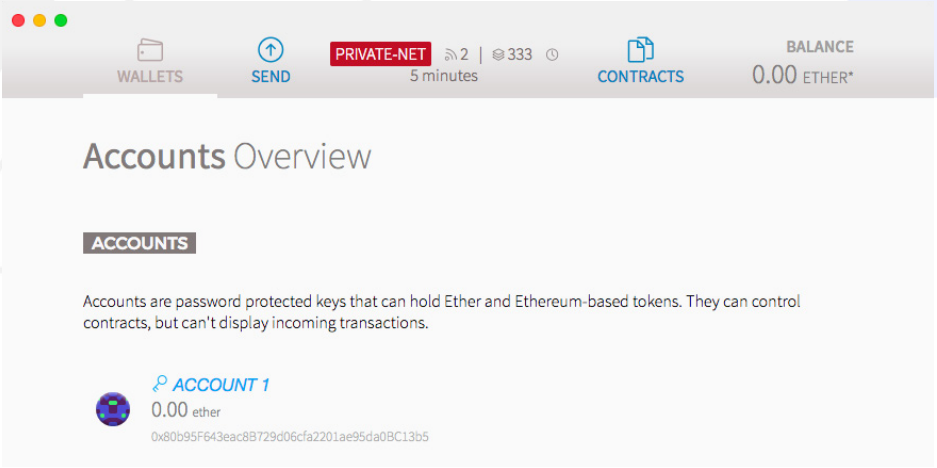
Isto trará a interface da Carteira, a qual já destaca que se trata de uma rede privada:



Note o destaque "PRIVATE-NET" e a contagem de blocos minerados que continua subindo enquanto houver nós mineradores.

Criando uma conta

Solicitar uma nova conta no menu da Carteira é trivial, sendo necessário fornecer uma senha longa. Após a criação da conta a interface da Carteira ficará como abaixo:



O id da conta deve ser copiado (a aplicação facilita a cópia) para que possamos reiniciar o nó minerador para que os créditos da mineração passem a alimentar esta conta.

Minerando na nova conta

Para reiniciar o minerador alimentando esta conta o script considera uma outra variável de ambiente (ETHERBASE). O minerador pode então ser reiniciado com o comando abaixo:

```
ETHERBASE=0x80b95F6.....0BC13b5 RPC_PORT=8545 ./runminer.sh miner1
```

Esta variável indica a conta que recebe os créditos de mineração do nó.

Reinicie então a Carteira e note que esta conta aumenta seu saldo de ETH a cada novo bloco minerado pelo nó.

DIVIRTA-SE

Várias coisas podem ser testadas para se observar o comportamento de uma rede privada de vários nós:

Rodar vários mineradores com distintos ETHERBASEs

Você pode criar novas contas e rodar mineradores que estarão disputando por cada bloco (usar um ETHERBASE diferente em cada um).

Transferir ETH entre contas

Usar a funcionalidade “Send” da Carteira para enviar ETH de uma conta para a outra.

Criar contas em nós diferentes

Usando um valor diferente para RPC_PORT é possível expor a interface JSON-RPC da mais de um nó e rodar mais de uma Carteira ao mesmo tempo, transferindo ETH entre Carteiras criadas em nós diferentes. **Divirta-se!**

CRIANDO SUA REDE ETHEREUM PRIVADA COM DOCKER

Autor:

ANDRÉ FERNANDES

CTIO da Vertigo Tecnologia

Docker Certified Associate (DCA)

Eng. de Computação pela Unicamp com pós-graduação em Gestão de Negócios.

QUER SABER MAIS?

VERTIGO TECNOLOGIA

www.vertigo.com.br

/vertigo.tecnologia

/vertigobr

/company/vertigo-tecnologia/