Preliminary Notes

The majority of this laboratory was inspired by the implementation of JusticeChain [1]. Part of this laboratory was inspired in sources elaborated by KC Tam[1].

Working on the blockchain ecosystem includes two major components: smart contract (chaincode) development and operations (infrastructure). We will learn how to create the chaincode to implement the system B4S QUC.

Contrarily to previous laboratories, this laboratory does not have exercises at the end of this guide. Instead, you will take a look at the code and experiment with our system while you navigate through this guide. To do so, please clone the repository at, using the command: git clone This laboratory uses the service *carbon.sh* to create some figures.

You will learn

1. What is the process behind creating a blockchain network
2. How does a web app related to the blockchain
3. An example of a complete, full-stack decentralized application running Fabric 2.2

Let us recall the architecture of our application:

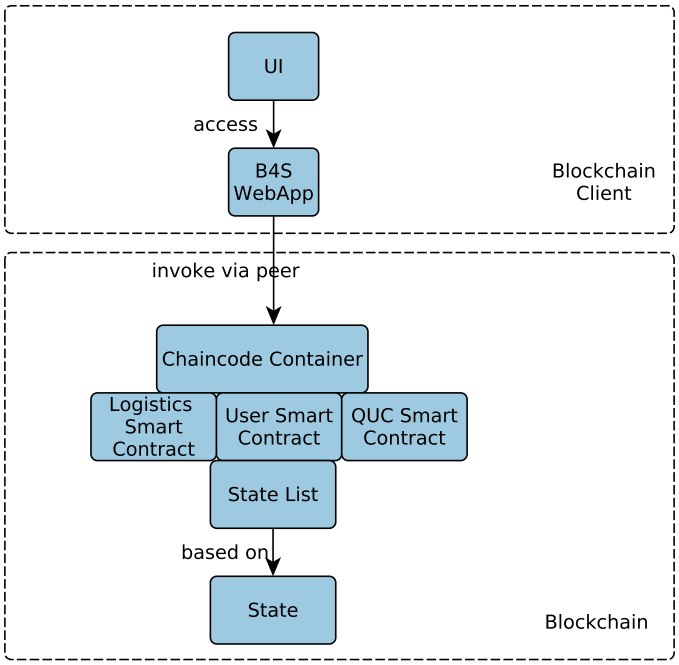

Figure 1: B4S architecture

[1] https://medium.com/@kctheservant

As discussed in the last lab, we can envision B4S as a full-stack system, in which the database is a blockchain. A user interface communicates with a blockchain client, which is a server. This server can act on behalf of an organization, issuing transactions against a peer node that redirects it to a chaincode container. This container holds several smart contracts (user, logistics, quc). Each smart contract is divided into the basic entities (User, Feedback), and useful abstractions help us develop chaincode (State, Statelist). This is illustrated in Figure 1.

1 Blockchain Network

Our network is composed by 2 organizations, each one with one peers, one CA, one couchDB instance. A channel mychannel (in our case, it is called b4s) connects all components. One orderer orders transactions of the network, as illustrated by the next Figure:

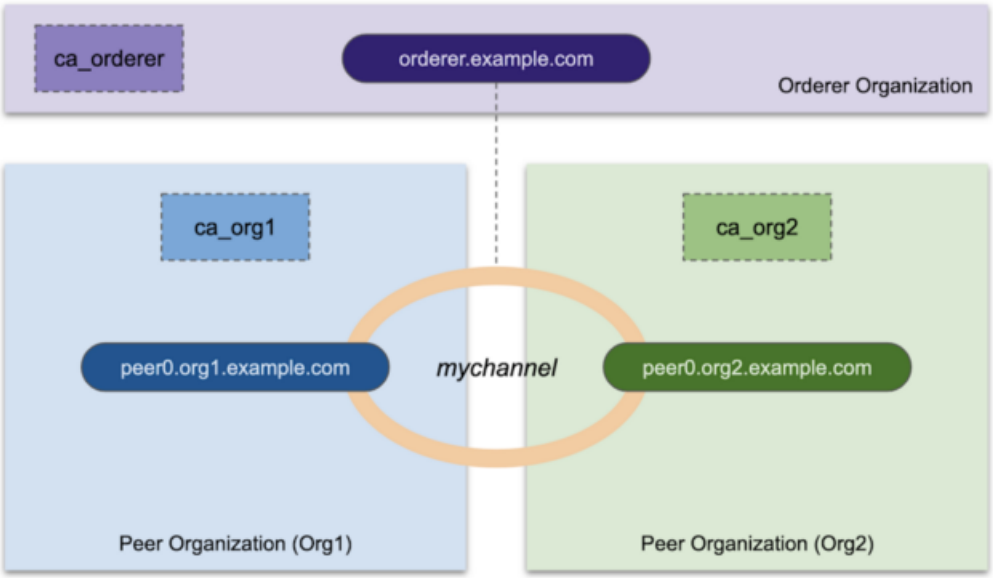

Figure 2: Network Architecture. Source: https://medium.com/@kctheservant/add-a-peer-to-an-organization-in-test-network-hyperledger-fabric-v2-2-4a08cb901c98

Let us see in more detail what happens at the `network-starter.sh`, and `network.sh` scripts.

1.1 Deploying the network

1. Bring up Certificate Authority (CA), one for each organization.
2. Generate crypto material for each organization with its own CA, including components (orderer/peer) and users (an admin and a user).
3. Generate consortium genesis block (system channel).
4. Bring up all the network components based on configuration in docker compose files.
5. For each channel
    a. generate configuration transaction and anchor peer update transactions for the channel
    b. generate channel genesis block with ordering service
    c. join all peers with that channel genesis block
    d. update anchor peer transactions for each organization on that channel
6. For each application chaincode
    a. package chaincode
    b. install chaincode package to the each peer
    c. approve chaincode definition for organizations according to lifecycle endorsement policy requirement
    d. commit chaincode definition

Figure 3: Network starter script steps. Source: https://medium.com/@kctheservant/add-a-peer-to-an-organization-in-test-network-hyperledger-fabric-v2-2-4a08cb901c98

On the B4S folder, when the `network-starter.sh` script is called, it performs some cleanup, and calls the `network.sh` script twice, located on the `test-network` folder. The first call is to instantiate the network, while the second calls installs the chaincode:

```
#Cleanup previous deployments
./network.sh down

#Instantiate network
./network.sh up createChannel -ca -s couchdb

#Deploy chaincode
./network.sh deployCC -l javascript
```

Figure 3 denote the necessary steps to start the network. Please open the `network.sh` script for inspection. As the flags given to the script are up and createChannel, the functions networkUp, and createChannel are called.

Firstly, the necessary cryptographic material has to be generated, in one of two ways: cryptogen or Fabric CA. Fabric CA is the best way to generate the necessary files, as

it is tailored to be used in production. It allows to generate new components or new users for an organization. This is achieved in the `registerEnroll.sh` script, and then the createOrgs function:

```
echo "############################################################"
echo "##### Generate certificates using Fabric CA's ############"
echo "############################################################"

IMAGE_TAG=$IMAGETAG_CA docker-compose -f $COMPOSE_FILE_CA up -d 2>&1

. organizations/fabric-ca/registerEnroll.sh

sleep 10

echo "############################################################"
echo "############ Create Org1 Identities ########################"
echo "############################################################"

createOrg1

echo "############################################################"
echo "############ Create Org2 Identities ########################"
echo "############################################################"

createOrg2

echo "############################################################"
echo "############ Create Orderer Org Identities ################"
echo "############################################################"

createOrderer

fi

echo
echo "Generate CCP files for Org1 and Org2"
./organizations/ccp-generate.sh
```

For example, to register the students union president, on behalf of org1, the registerEnroll script calls the Fabric CA client:

```
echo "================ ORG1 IDENTITIES ========================="
```

```
echo "## REGISTER studentUnionPresident"
fabric-ca-client register -d --id.name studentUnionPresident
 ↪  --id.affiliation org1 --id.attrs '"hf.Registrar.Roles=peer,client"'
 ↪  --id.attrs hf.Revoker=true --id.attrs
 ↪  'participantType=entity:ecert,entityId=1:ecert,organizationName=studentUnionTecni
 ↪  --id.secret pw --tls.certfiles
 ↪  ${PWD}/organizations/fabric-ca/org1/tls-cert.pem
echo "## ENROLL studentUnionPresident"
fabric-ca-client enroll -u https://studentUnionPresident:pw@localhost:7054
 ↪  --enrollment.attrs "participantType,entityId,organizationName"
 ↪  --caname ca-org1 -M
 ↪  ${PWD}/organizations/peerOrganizations/org1.example.com/users/studentUnionPreside
 ↪  --tls.certfiles ${PWD}/organizations/fabric-ca/org1/tls-cert.pem
```

This step is essential for access control within the blockchain. Note how when the client is being registered, we are recording different attributes: `id.name, id.affiliation, id.attrs, participantType, entityId, organizationName`. Those attributes will belong to the X.509 certificate issued to them, which can be later inspected by chaincode. This means that we have a reliable way of identifying participants on the chain and restricting who can access specific functions. For example, we can set up our chaincode so that only participants with the attribute participantType=university can call the chaincode function to createUniversity.

After the necessary files are created, the network is bootstrapped by using Docker. Essentially, the script calls the `docker-compose ${COMPOSE_FILES} up -d` command, where the variable COMPOSE_FILES holds the necessary docker files. The `docker-compose-test-n` and `docker-compose-couch.yaml` are used to create the peers, orderers, CA, and CouchDB images.

After that, the channel is created, via the `createChannel` script. It uses the configtxgen tool to generate two-channel artifacts: the genesis block and the first transaction. Those artifacts are configurable through the file configtx.yaml, which is located under config. After that, the channel is created via the command `peer channel create`, and both organizations join by issuing `peer channel join`.

We now have a working blockchain! But for being useful, we need to start...

1.2 Deploying Chaincode

Chaincode has a specific lifecycle, following a well-defined sequence of steps until it is usable:

1. Package chaincode: this step takes to input the source files of the chaincode (those on the chaincode directory) and creates a file (.tar.gz) with them, along with metadata

2. Install chaincode: installs the chaincode package on the peers
3. Approve chaincode definition: each organization approves the code that was installed
4. Commit chaincode definition to channel: when the lifecycle endorsement policy (in our case, all have to endorse the installed chaincode), an organization commits the new chaincode version.

This process takes place at the `deployCC.sh` script, inside the `scripts` folder. In step one, the chaincode is packed:

```
# Step 1
peer lifecycle chaincode package b4s.tar.gz

# Step 2 - for each org
peer lifecycle chaincode install b4s.tar.gz

# Step 3 - for each org
peer lifecycle chaincode approveformyorg [...]

# Step 4 -for each org
peer lifecycle chaincode commit [...]
```

Each step comprises different inputs that can be analyzed during the lab. After the respective verifications, the script tests invoking a particular smart contracts (via `chaincodeInvokeInit` and `chaincodeInvoke`).

2 Blockchain Client

We now have a blockchain up and running to interact using the command-line interface (see the previous Lab). However, this is not useful for enterprises, organizations, and communities that want to use blockchain technology. To facilitate the interaction with the blockchain, we need to build a Blockchain Client, for each organization involved. This blockchain client is a web server that interacts with the blockchain via the fabric-sdk. On top of this web server, we deploy a user interface, allowing end-users to interact with the network. Those end-users are issued credentials (cryptographic materials discussed before) via the Fabric CA, stored in a wallet. Users utilize their credentials to issue transactions on the blockchain via their organization's peer node.

Let's get started!

2.1 Setting up

First, we need to add our user identities to their respective wallets. Under the B4S folder, you have a directory called organizations. In a real-world deployment, each sub-folder (students-union and university) would belong exclusively to a single stakeholder. Moreover, the crypto materials are generated only for them. On the B4S folder, run the script `init.sh`. Analyze this script. It essentially installs the necessary dependencies, adds the credentials to the wallet, and issues a transaction on the blockchain for both organizations. For the students-union:

```
echo '====== ON BEHALF OF A STUDENTS UNION ======'
npm install
node renamePK
node addToWallet
```

For the university, a last step is conducted, the creation of a university.

```
echo '====== ON BEHALF OF UNIVERSITY ======'
npm install
node renamePK
node addToWallet
node showIdentity
node createUniversity
```

The result of the first and second scripts (node renamePK and node addToWallet) creates a new folder, `identity`, under each organization. Analyze the generated contents. The contents are .id files, which hold the certificates generated by the CA:

```
{
"credentials": {
"certificate": "-----BEGIN CERTIFICATE-----[...]\n-----END
↪  CERTIFICATE-----\n",
"privateKey": "-----BEGIN PRIVATE KEY-----[...]-----END PRIVATE
↪  KEY-----\n"
},
"mspId": "Org1MSP",
"type": "X.509",
"version": 1
}
```

The script showIdentity issues a transaction against the blockchain requesting some information.

Analyze the code of `showIdentity.js`, under `support/Lab05/b4s/organization/university`:

```
const { Wallets, Gateway } = require('fabric-network');
const University =
↪  require('../../../../chaincode/lib/logistics/university');
[...]

/ A wallet stores a collection of identities for use
const wallet = await
↪  Wallets.newFileSystemWallet('../identity/org2/wallet');

// A gateway defines the peers used to access Fabric networks
const gateway = new Gateway();
```

First, we import the Wallet and Gateway classes from the Fabric SDK, interact with the wallet, and blockchain. We are also using the University class we created.

Now, we need to connect to Fabric via the user whose credentials were used by the CA. When we generated the connection profile for the user, it was saved on the folder Identity.

```
// Specify userName for network access
const userName = 'universityTecnicoLisboa';

// Load connection profile; will be used to locate a gateway
let connectionProfile =
↪  yaml.safeLoad(fs.readFileSync('../gateway/connection-org2.yaml',
↪  'utf8'));
```

We now connect to Fabric:

```
await gateway.connect(connectionProfile, connectionOptions);
```

... to the desired channel...

```
const network = await gateway.getNetwork('mychannel');
```

... obtain the smart contract we need to use...

```
const contract = await network.getContract('b4s','org.b4s.logistics');
```

.. and submit the createUniversity transaction!

```
const universityBytes = await
↪  contract.submitTransaction('createUniversity', 'TecnicoLisboa',
↪  'Portugal');
```

Finally, we retrieve the response from Fabric into a human-readable format:

```
let university = University.fromBuffer(universityBytes);
console.log(university)
```

Now, analyze the code of `createUniversity.js`, under `support/Lab05/b4s/organization/unive`
You will notice that it is very similar to showIdentity, only a line changes:

```
const response = await contract.submitTransaction('showIdentity');
```

That is right, only the transaction type that is called changes.

This procedure applies to all possible transactions defined by a smart contract. We first connect to the network and channel, obtain the smart contract, and submit a transaction with our credentials. But for a large organization, it would be very cumbersome to maintain a script for each employee and for each transaction type. That is why we can encapsulate this complexity into a web app.

2.2 B4S WebApp

For each organization, a web app takes the generated credentials and interacts with the network. The idea is for the end-users to interact with the blockchain simply: the web app does the same as the scripts above.

Under `b4s/organization`, there are two organizations depicted: students-union corresponds to org1, having a wallet that supports two users. University corresponds to org2. Go to the university folder, and then to b4s-client. To start the client, run npm run start.

The blockchain client is up and running! The ideal situation now would be to have a user interface to interactively issue transactions to the blockchain. For testing purposes, we will issue GET or POST requests to a specific endpoint to verify that the requests are translated to transactions. Verify the credentials loaded by the b4s-client by requesting a GET to /logistics/myIdentity:

```
curl -X GET http://127.0.0.1:3001/logistics/myIdentity
```

The response should be similar:

```
rafael@x:~/Projects/university-course/support/Lab05/b4s/organization/university/b4s_c
↪  npm run start

> logger@0.0.0 start
↪  /home/rafael/Projects/university-course/support/Lab05/b4s/organization/university
> node app

Server running at http://127.0.0.1:3001/
Connecting to Fabric
Use network channel: mychannel.
Use org.b4s chaincode
identity:  {
participantType: 'university',
auditorId: null,
organizationName: 'universityTecnicoLisboa',
loggerId: null,
adminId: null,
hfEnrollmentID: null,
getID: 'x509::/C=US/ST=North
↪  Carolina/O=Hyperledger/OU=client/OU=org2/CN=universityTecnicoLisboa::/C=US/ST=Nor
↪  Carolina/O=Hyperledger/OU=Fabric/CN=fabric-ca-server',
getMSPID: 'Org2MSP'
}
GET /logistics/myIdentity 200 2657.232 ms - 77
```

You can check that the transaction was successful by analyzing the peer logs and the logs of the chaincode container. Execute `docker logs peer0.org1.example.com`, and then

```
docker logs  dev-peer0.org1.example.com-b4s_1-[SPECIFIC-ID]
```

replacing SPECIFIC-ID by the ID of your container (you can check this by issuing docker logs, and searching for dev-peer...). The peer0.org1 container shows:

```
2020-09-10 10:59:48.490 UTC [kvledger] CommitLegacy -> INFO 0cd
↪  [mychannel] Committed block [15] with 1 transaction(s) in 34ms
↪  (state_validation=0ms block_and_pvtdata_commit=3ms state_commit=29ms)
↪  commitHash=[8810dfd0a66bdb305e0932ca3ed80af47472ccdabfc8db02cef23d449fef3f0d]
```

The chaincode container shows:

```
Show identity called by:  x509::/C=US/ST=North
↪  Carolina/O=Hyperledger/OU=client/OU=org2/CN=universityTecnicoLisboa::/C=US/ST=Nor
↪  Carolina/O=Hyperledger/OU=Fabric/CN=fabric-ca-server
```

```
2020-09-10T10:51:56.134Z info [c-api:lib/handler.js]
↪ [mychannel-c0ad8972] Calling chaincode Invoke() succeeded. Sending
↪ COMPLETED message back to peer
```

Which corresponds to the console.log done at the chaincode (see showIdentity from the logistics-contract).

This request is semantically equivalent to the script showIdentity. In fact, the client just loads this script and manages the credentials for you.

Now, we want to perform more complex transactions. For instance, we can create a university by requesting a POST to /logistics/createUniversity, containing the necessary arguments.

```
curl -X POST -F 'universityId=123' -F 'regionId=Portugal'
↪ http://127.0.0.1:3001/logistics/createUniversity
```

The webapp console returns:

```
===============
University {
class: 'org.b4s.university',
key: '"123":"Portugal"',
currentState: null,
universityId: '123',
regionId: 'Portugal'
}
Transaction complete.
POST /logistics/createUniversity 200 2605.641 ms - 122
```

Let's also create two students, by issuing:

```
curl -X POST -F 'studentId=012345' -F 'universityId=123' -F
↪ 'courses=course1'  http://127.0.0.1:3001/users/createStudent
```

That's it! Notice that there is also a b4s-client for students-union under their directory. You can repeat this experiment but notice that the students' client only contains the myIdentity endpoint. We are now leveraging a user interface from the University institution to render the results from the getAllStudents endpoint, which retrieves a full list of enrolled students. Go to /organizations/university/user-interface. To install dependencies, run npm install. After that, bootstrap the user interface by running npm run serve. (Note, if this step does not work, install the Vue Client[2]). It is out of

[2]https://vuejs.org/v2/guide/installation.html

this course's scope to learn about Vue.js, but you can check the code at University-frontend.vue, under src/Views to inspect how it works. Essentially, it makes a request to http://localhost:3001/users/getAllStudents, which has credentials loaded and parses the request.

Open your browser at localhost:8080, and try out to retrieve the students' list.

That's it! These are the basics of Hyperledger Fabric.

3 Clean up

On the B4S folder, run: `./network-clean`. This calls `network.sh down`, that tears down the whole infrastructure, as removes all stopped containers.

For each organization, you may run the script remove_wallet.sh, under the `utils` folder. This deletes the created credentials.

4 Exercises

The `network.sh` script can be reproduced step by step[3][4]. Reproduce the process described on the B4S network.

Suppose that we want to add a new peer node to the network. Referring to Figure 3, which steps are to be taken?

In the deployCC.sh script, what is the difference between chaincodeInvoke and chaincodeInvokeInit?

What are some dangers of using the web app as a centralized party to access the blockhain?

What are the differences of this project to a production environment?

References

[1] R. Belchior, A. Vasconcelos, and M. Correia. Towards Secure, Decentralized, and Automatic Audits with Blockchain. In *European Conference on Information Systems*, 2020.

[3] https://medium.com/@kctheservant/add-a-peer-to-an-organization-in-test-network-hyperledger-fabric-v2-2-4a08cb901c98

[4] https://hyperledger-fabric.readthedocs.io/en/release-2.2/tutorial/commercial_paper.html