P P SAVANI
UNIVERSITY

Module 5

Blockchain Fabric Development

Outline

- Participants and Components Overview
- Developer Consideration

How does Hyperledger Composer work in practice?

- For an example of a business network in action; a realtor can quickly model their business network as such:
- Assets: houses
- Participants: buyers and homeowners
- Transactions: buying or selling houses, and creating and closing listings
- Participants can have their access to transactions restricted based on their role as either a buyer, seller, or realtor. The realtor can then create an application to present buyers and sellers with a simple user interface for viewing open listings and making offers. This business network could also be integrated with existing inventory system, adding new houses as assets and removing sold properties. Relevant other parties can be registered as participants, for example a land registry might interact with a buyer to transfer ownership of the land.

Peer Channel

- Create – Create a channel and write genesis block to a file.
- Fetch – Fetch a specified block and write it to a file.
- Getinfo – Get Blockchain information of a specified channel.
- Join – Joins a new peer to the channel.
- List – List of channels a peer has joined.
- Signconfigtx - Signs the supplied configtx update file in place on the filesystem.
- Update - Signs and sends the supplied configtx update file to the channel.

Composer Package

- Composer supports four types:
- library: This is the default. It will simply copy the files to /vendor.
- project: This denotes a project rather than a library. For example application shells like the Symfony standard edition, CMSs like the SilverStripe installer or full fledged applications distributed as packages.
- metapackage: An empty package that contains requirements and will trigger their installation, but contains no files and will not write anything to the filesystem.
- composer-plugin: A package of type composer-plugin may provide an installer for other packages that have a custom type. Only use a custom type if you need custom logic during installation.

Byfn.sh

- The build your first network (BYFN) scenario provisions a sample Hyperledger Fabric network consisting of two organizations, each maintaining two peer nodes.

- It also will deploy a “Solo” ordering service by default, though other ordering service implementations are available.

Crypto generator

- Cryptogen tool to generate the cryptographic material (x509 certs and signing keys) for our various network entities. These certificates are representative of identities, and they allow for sign/verify authentication to take place as our entities communicate and transact.

Crypto generator

- Cryptogen consumes a file — crypto-config.yaml — that contains the network topology and allows us to generate a set of certificates and keys for both the Organizations and the components that belong to those Organizations. Each Organization is provisioned a unique root certificate (ca-cert) that binds specific components (peers and orderers) to that Org.

- By assigning each Organization a unique CA certificate, we are mimicking a typical network where a participating Member would use its own Certificate Authority.

- Transactions and communications within Hyperledger Fabric are signed by an entity's private key (keystore), and then verified by means of a public key (signcerts).

Crypto generator

- You will notice a count variable within this file. We use this to specify the number of peers per Organization; in this case there are two peers per Org.

- After we run the cryptogen tool, the generated certificates and keys will be saved to a folder titled crypto-config.

- Note that the crypto-config.yaml file lists five orderers as being tied to the orderer organization. While the cryptogen tool will create certificates for all five of these orderers, unless the Raft or Kafka ordering services are being used, only one of these orderers will be used in a Solo ordering service implementation and be used to create the system channel and mychannel.

Configuration Transaction Generator

- The configtxgen tool is used to create four configuration artifacts:
- orderer genesis block, channel configuration transaction, and two anchor peer transactions - one for each Peer Org.
- The orderer block is the Genesis Block for the ordering service, and the channel configuration transaction file is broadcast to the orderer at Channel creation time.
- The anchorpeer transactions, as the name might suggest, specify each Org's Anchor Peer on this channel.

Configuration Transaction Generator

- Configtxgen consumes a file - configtx.yaml - that contains the definitions for the sample network. There are three members - one Orderer Org (OrdererOrg) and two Peer Orgs (Org1 & Org2) each managing and maintaining two peer nodes. This file also specifies a consortium - SampleConsortium - consisting of our two Peer Orgs.
- TwoOrgsOrdererGenesis: generates the genesis block for a Solo ordering service.
- SampleMultiNodeEtcdRaft: generates the genesis block for a Raft ordering service. Only used if you issue the -o flag and specify etcdraft.
- SampleDevModeKafka: generates the genesis block for a Kafka ordering service. Only used if you issue the -o flag and specify kafka.
- TwoOrgsChannel: generates the genesis block for our channel, mychannel

Build an application with Hyperledger Fabric

Follow the link for detail process with consideration of asset transfer example.

https://hyperledger-fabric.readthedocs.io/en/release-2.2/write_first_app.html

About Asset Transfer

1. Sample application: which makes calls to the blockchain network, invoking transactions implemented in the chaincode (smart contract). The application is located in the following fabric-samples directory:

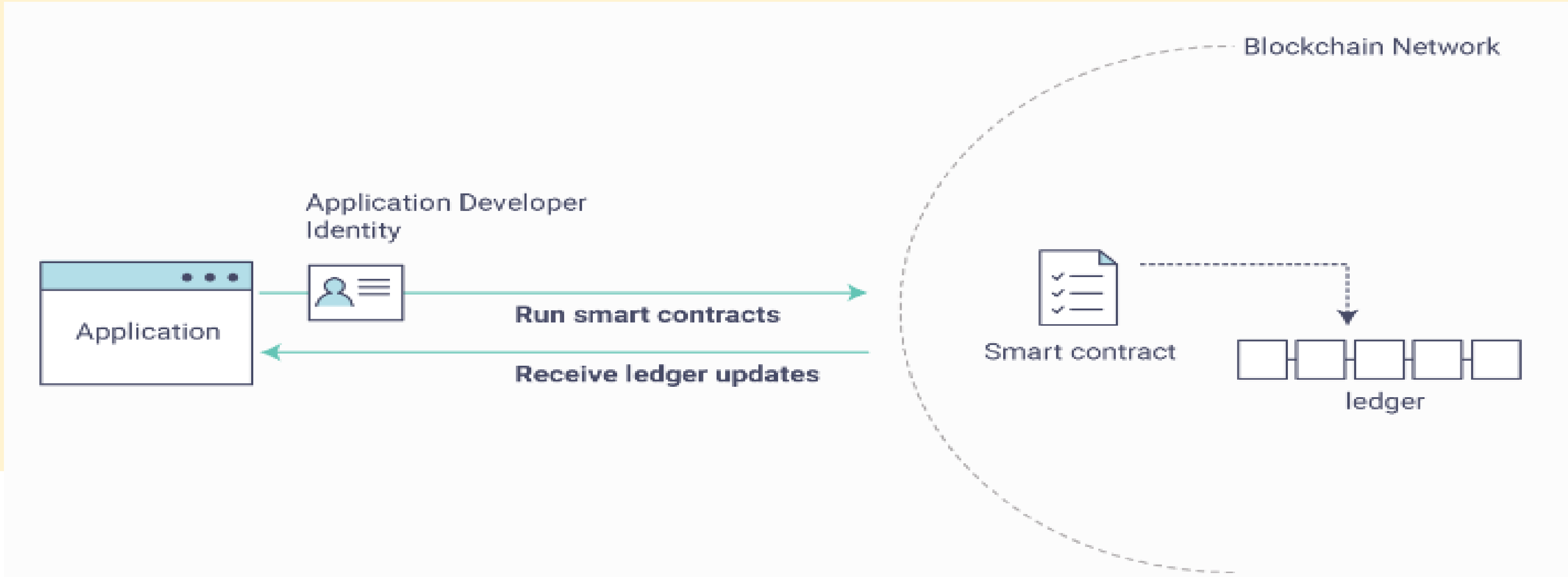
asset-transfer-basic/application-javascript

2. Smart contract itself, implementing the transactions that involve interactions with the ledger. The smart contract (chaincode) is located in the following fabric-samples directory:

asset-transfer-basic/chaincode-(javascript, java, go, typescript)

Principle Steps

1. Setting up a development environment. Our application needs a network to interact with, so we'll deploy a basic network for our smart contracts and application.



2. Explore a sample smart contract. We’ll inspect the sample assetTransfer (javascript) smart contract to learn about the transactions within it, and how they are used by an application to query and update the ledger.

3. Interact with the smart contract with a sample application. Our application will use the assetTransfer smart contract to create, query, and update assets on the ledger. We’ll get into the code of the app and the transactions they create, including initializing the ledger with assets, querying an asset, querying a range of assets, creating a new asset, and transferring an asset to a new owner.

Prerequisite

- **If you are using macOS**, complete the following steps:

Install Homebrew.

- Check the Node SDK prerequisites to find out what level of Node to install.

Run brew install node to download the latest version of node or choose a specific version, for example: brew install node@10 according to what is supported in the prerequisites.
Run npm install.

- **If you are on Windows**, you can install the windows-build-tools with npm which installs all required compilers and tooling by running the following command:

npm install --global windows-build-tools

- **If you are on Linux**, you need to install Python v2.7, make, and a C/C++ compiler toolchain such as GCC. You can run the following command to install the other tools:
- sudo apt install build-essential

Setup Blockchain Network

- Navigate to the test-network subdirectory within your local clone of the fabric-samples repository.

```
cd fabric-samples/test-network
```

- If you already have a test network running, bring it down to ensure the environment is clean.

```
./network.sh down
```

- Launch the Fabric test network using the network.sh shell script.

```
./network.sh up createChannel -c mychannel -ca
```

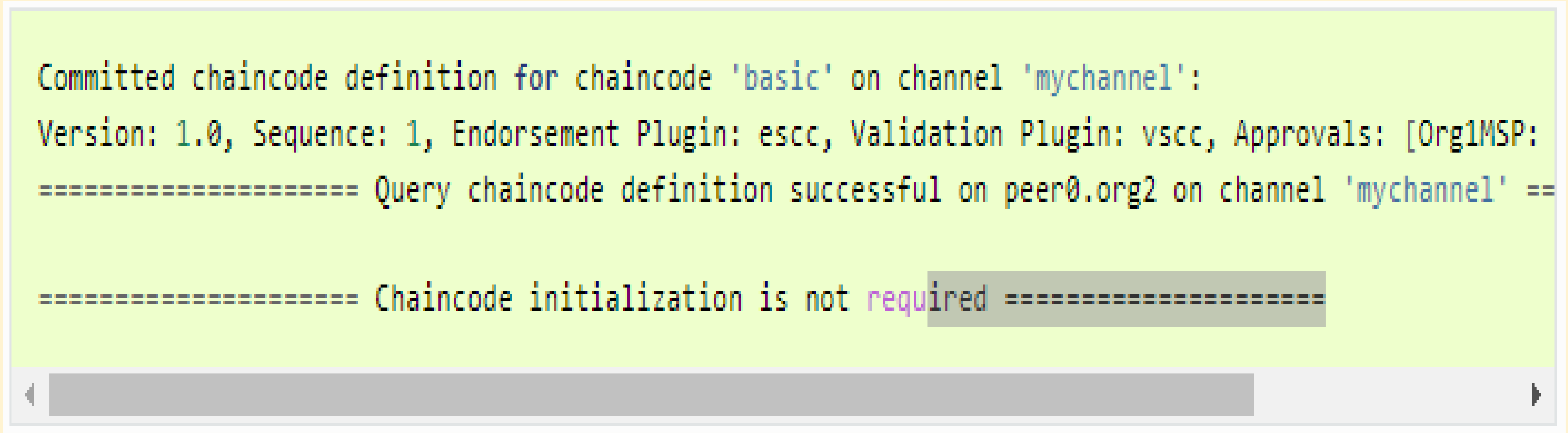
This command will deploy the Fabric test network with two peers, an ordering service, and three certificate authorities (Orderer, Org1, Org2). Instead of using the cryptogen tool, we bring up the test network using Certificate Authorities, hence the -ca flag. Additionally, the org admin user registration is bootstrapped when the Certificate Authority is started.

- let’s deploy the chaincode by calling the ./network.sh script with the chaincode name and language options.

```
./network.sh deployCC -ccn basic -ccp ../asset-transfer-basic/chaincode-javascript/ -ccl javascript
```

- Behind the scenes, this script uses the chaincode lifecycle to package, install, query installed chaincode, approve chaincode for both Org1 and Org2, and finally commit the chaincode.

Output After deploying Chaincode



```
Committed chaincode definition for chaincode 'basic' on channel 'mychannel':
Version: 1.0, Sequence: 1, Endorsement Plugin: escc, Validation Plugin: vscc, Approvals: [Org1MSP:
====================== Query chaincode definition successful on peer0.org2 on channel 'mychannel' ==

====================== Chaincode initialization is not required ======================
```

Sample Aplication

- let’s prepare the sample Asset Transfer Javascript application that will be used to interact with the deployed chaincode.

JavaScript application

- Note that the sample application is also available in Go and Java at the links below:

Go application
Java application

- Open a new terminal, and navigate to the application-javascript folder.

```
cd asset-transfer-basic/application-javascript
```

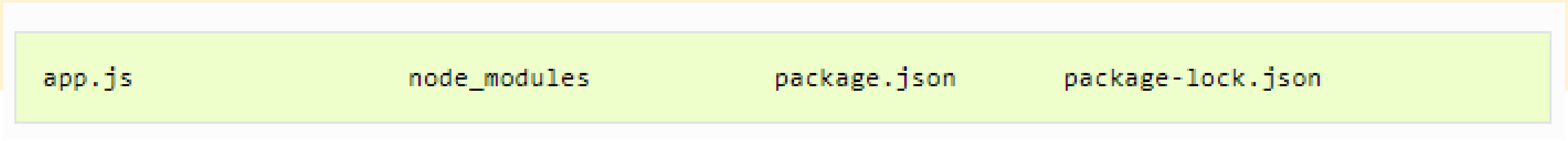
- This directory contains sample programs that were developed using the Fabric SDK for Node.js. Run the following command to install the application dependencies. It may take up to a minute to complete:

```
npm install
```

- This process is installing the key application dependencies defined in the application's **package.json**. The most important of which is the fabric-network **Node.js** module; it enables an application to use identities, wallets, and gateways to connect to channels, submit transactions, and wait for notifications.

- Once npm install completes, everything is in place to run the application. Let's take a look at the sample JavaScript application files we will be using in this tutorial. Run the following command to list the files in this directory:

```
ls
```



```
app.js              node_modules              package.json        package-lock.json
```

- The first part of the following section involves communication with the Certificate Authority.
- You may find it useful to stream the CA logs when running the upcoming programs by opening a new terminal shell and running **docker logs -f ca_org1.**
- When we started the Fabric test network back in the first step, an admin user — literally called admin — was created as the registrar for the Certificate Authority (CA).
- Our first step is to generate the private key, public key, and X.509 certificate for admin by having the application call the enrollAdmin .
- This process uses a Certificate Signing Request (CSR) — the private and public key are first generated locally and the public key is then sent to the CA which returns an encoded certificate for use by the application.
- These credentials are then stored in the wallet, allowing us to act as an administrator for the CA

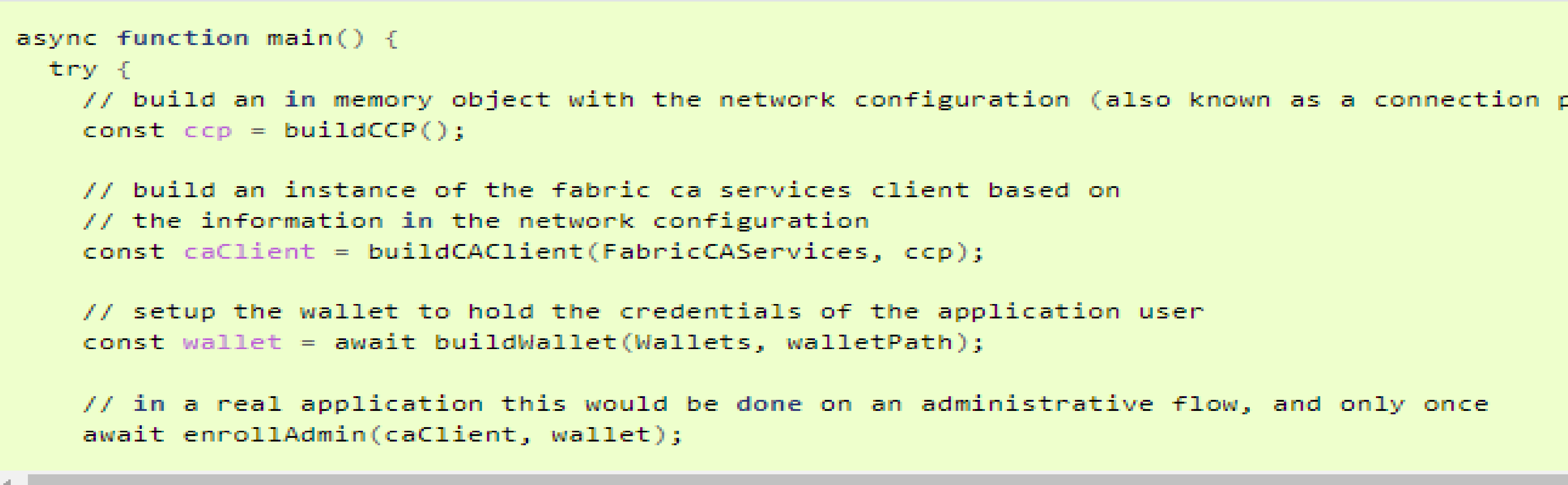
- Run the application and then step through each of the interactions with the smart contract functions.
- From the asset-transfer-basic/application-javascript directory, run the following command:
- **node app.js**

The application enrolls the admin user

- It is important to note that enrolling the admin and registering the app user are interactions that take place between the application and the Certificate Authority, not between the application and the chaincode.

- If you examine the chaincode in **asset-transfer-basic/chaincode-javascript/lib** you will find that the chaincode does not contain any functionality that supports enrolling the admin or registering the user.

- In the application code below, you will see that after getting reference to the common connection profile path, making sure the connection profile exists, and specifying where to create the wallet, enrollAdmin() is executed and the admin credentials are generated from the Certificate Authority.



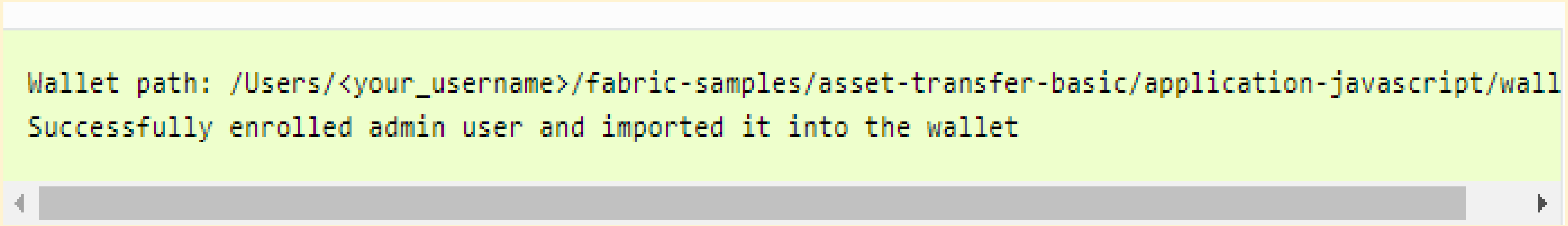
```
async function main() {
  try {
    // build an in memory object with the network configuration (also known as a connection p
    const ccp = buildCCP();

    // build an instance of the fabric ca services client based on
    // the information in the network configuration
    const caClient = buildCAClient(FabricCAServices, ccp);

    // setup the wallet to hold the credentials of the application user
    const wallet = await buildWallet(Wallets, walletPath);

    // in a real application this would be done on an administrative flow, and only once
    await enrollAdmin(caClient, wallet);
```

- This command stores the CA administrator's credentials in the wallet directory. You can find administrator's certificate and private key in the wallet/admin.id file.



```
Wallet path: /Users/<your_username>/fabric-samples/asset-transfer-basic/application-javascript/wall
Successfully enrolled admin user and imported it into the wallet
```

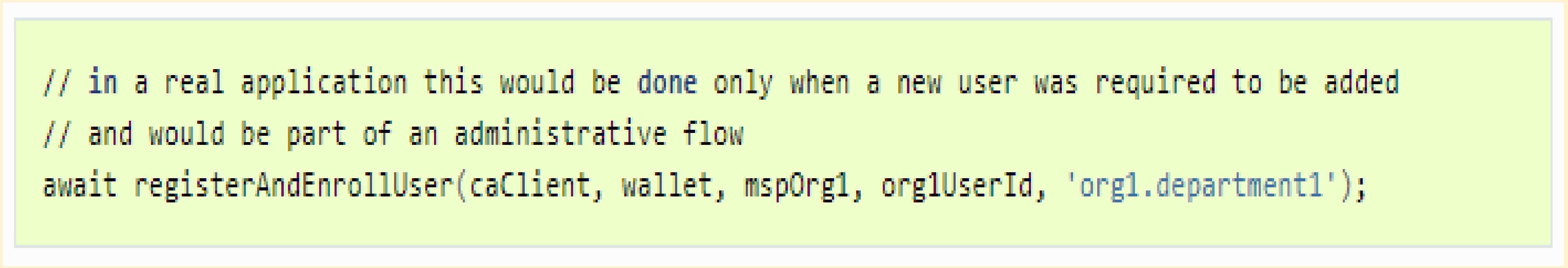
- If you decide to start over by taking down the network and bringing it back up again, you will have to delete the wallet folder and its identities prior to re-running the javascript application or you will get an error.

- This happens because the Certificate Authority and its database are taken down when the test-network is taken down but the original wallet still remains in the application-javascript directory so it must be deleted. When you re-run the sample javascript application, a new wallet and credentials will be generated.

- Since the Fabric CA interactions are common across the samples, enrollAdmin() and the other CA related functions are included in the **fabric-samples/test-application/javascript/CAUtil**.js common utility.

The application registers and enrolls an application user

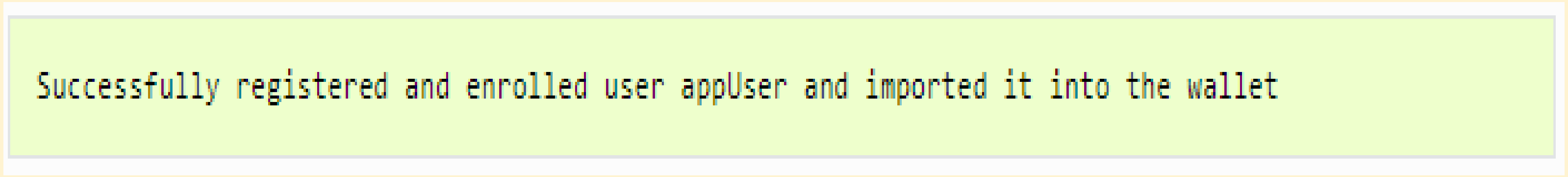
- we have the administrator’s credentials in a wallet, the application uses the admin user to register and enroll an app user which will be used to interact with the blockchain network.



```
// in a real application this would be done only when a new user was required to be added
// and would be part of an administrative flow
await registerAndEnrollUser(caClient, wallet, mspOrg1, org1UserId, 'org1.department1');
```

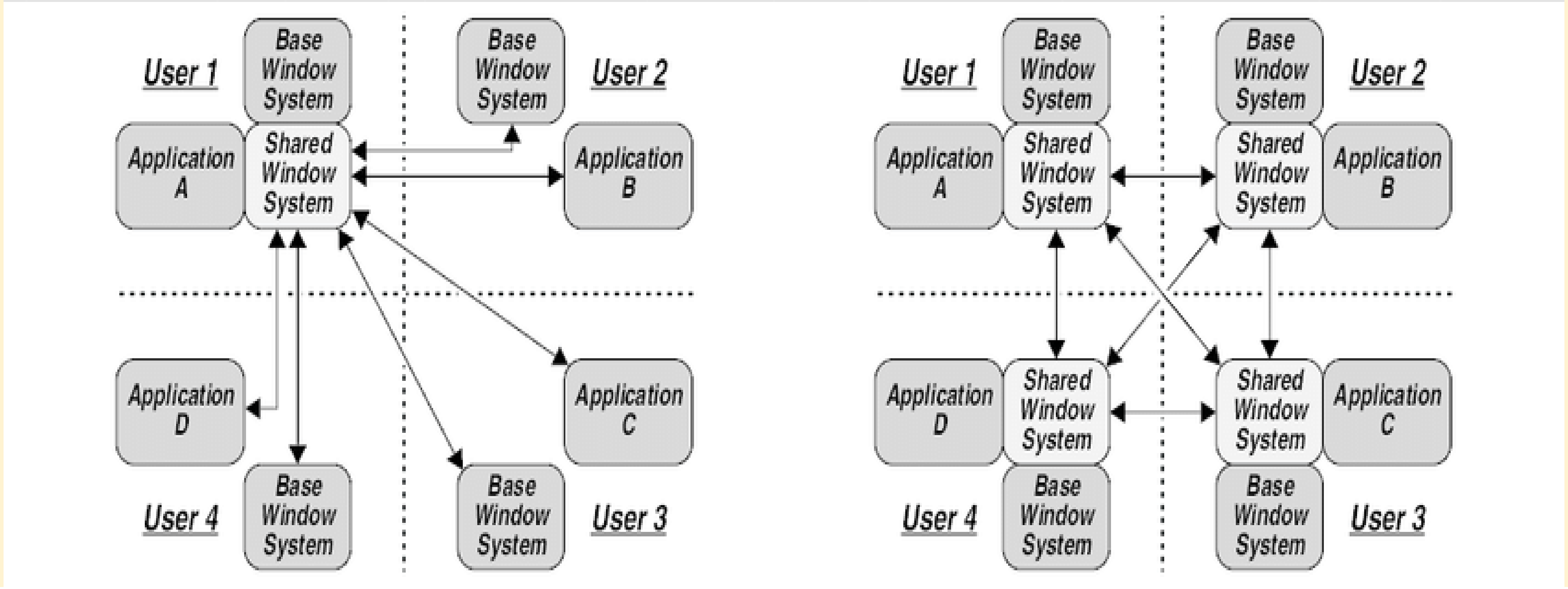
- Similar to the admin enrollment, this function uses a CSR to register and enroll appUser and store its credentials alongside those of admin in the wallet. We now have identities for two separate users — admin and appUser — that can be used by our application.

- Scrolling further down in your terminal output, you should see confirmation of the app user registration similar to this:



```
Successfully registered and enrolled user appUser and imported it into the wallet
```

Software Architecture



Advantages of Distributed System

- Higher computing power
- Cost reduction
- Higher reliability
- Ability to grow naturally

Higher Computing Power

- The computing power of a distributed system is the result of combining the computing power of all connected computers. Hence, distributed systems typically have more computing power than each individual computer.

- This has been proven true even when comparing distributed systems comprised of computers of relatively low computing power with isolated super computers.

Cost Reduction

- The price of mainstream computers, memory, disk space, and networking equipment has fallen dramatically during the past 20 years.

- Since distributed systems consist of many computers, the initial costs of distributed systems are higher than the initial costs of individual computers.

- However, the costs of creating, maintaining, and operating a super computer are still much higher than the costs of creating, maintaining, and operating a distributed system.

- This is particularly true since replacing individual computers of a distributed system can be done with no significant overall system impact.

Higher Reliability

- The increased reliability of a distributed system is based on the fact that the whole network of computers can continue operating even when individual machines crash.

- A distributed system does not have a single point of failure. If one element fails, the remaining elements can take over. Hence, a single super computer typically has a lower reliability than a distributed system.

Ability to Grow Naturally

- The computing power of a distributed system is the result of the aggregated computing power of its constituents. One can increase the computing power of the whole system by connecting additional computers with the system. As a result, the computing power of the whole system can be increased incrementally on a fine-grained scale.
- This supports the way in which the demand for computing power increases in many organizations. The incremental growth of distributed systems is in contrast to the growth of the computing power of individual computers.
- Individual computers provide identical power until they are replaced by a more powerful computer. This results in a discontinuous growth of computing power, which is only rarely appreciated by the consumers of computing services.

The Disadvantages of Distributed Systems

- Coordination overhead
- Communication overhead
- Dependency on networks
- Higher program complexity
- Security issues

Coordination Overhead

- Distributed systems do not have central entities that coordinate their members. Hence, the coordination must be done by the members of the system themselves.

- Coordinating work among coworkers in a distributed system is challenging and costs effort and computing power that cannot be spent on the genuine computing task, hence, the term coordination overhead.

Communication Overhead

- Coordination requires communication. Hence, the computers that form a distributed system have to communicate with one another.

- This requires the existence of a communication protocol and the sending, receiving, and processing of messages, which in turn costs effort and computing power that cannot be spend on the genuine computing task, hence, the term communication overhead.

Dependencies on Networks

- Any kind of communication requires a medium. The medium is responsible for transferring information between the entities communicating with one another.

- Computers in distributed systems communicate by means of messages passed through a network.

- Networks have their own challenges and adversities, which in turn impact the communication and coordination among computers that form a distributed system.

- However, without any network, there will be no distributed system, no communication, and therefore no coordination among the nodes, thus the dependency on networks.

Higher Program Complexity

- Solving a computation problem involves writing programs and software. Due to the disadvantages mentioned previously, any software in a distributed system has to solve additional problems such as coordination, communication, and utilizing of networks. This increases the complexity of the software.

Security Issues

- Communication over a network means sending and sharing data that are critical for the genuine computing task.

- However, sending information through a network implies security concerns as untrustworthy entities may misuse the network in order to access and exploit information.

- Hence, any distributed system has to address security concerns. The less restricted the access to the network over which the distributed nodes communicate is, the higher the security concerns are for the distributed system.

Distributed Peer-to-Peer Systems

- Peer-to-peer networks are a special kind of distributed systems. They consist of individual computers (also called nodes), which make their computational resources (e.g., processing power, storage capacity, data or network bandwidth) directly available to all other members of the network without having any central point of coordination.

- The nodes in the network are equal concerning their rights and roles in the system. Furthermore, all of them are both suppliers and consumers of resources.

Mixing Centralized and Distributed Systems

- Centralized and distributed systems are architectural antipodes. Technical antipodes have always inspired engineers to create hybrid systems that inherit the strength of their parents.

- Centralized and distributed systems are no exception to this. There are two archetypical ways of combining these antipodes, and they need to be understood since they will become important when learning about blockchain applications in the real world.

- They are centrality within a distributed system and the distributed system inside the center.