Informatics

Evaluation of Platforms for Distributed Ledger Based Trade Finance

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Patrick Fichtinger, BSc
Matrikelnummer 01427619

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo

Wien, 18. August 2021

Patrick Fichtinger

Monika di Angelo

Informatics

Evaluation of Platforms for Distributed Ledger Based Trade Finance

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Business Informatics

by

Patrick Fichtinger, BSc
Registration Number 01427619

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Monika di Angelo

Vienna, 18th August, 2021

Patrick Fichtinger

Monika di Angelo

Erklärung zur Verfassung der Arbeit

Patrick Fichtinger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. August 2021

Patrick Fichtinger

Kurzfassung

Die Blockchain-Technologie ermöglicht verschiedenste Anwendung die aufgrund ihres dezentralen Charakters eine hohe Ausfallssicherheit haben, eine einheitliche, widerspruchsfreie Transaktionshistorie bieten und deren involvierte Parteien sich nicht vollständig vertrauen müssen, Durch diese Eigenschaften werden Blockchains unter anderem für dezentralisiertes Finanzwesen (DeFi) interessant. Im Außenhandel schließen Unternehmen Geschäfte mit ausländischen Unternehmen ab und verwenden zur Zahlungsabwicklung in der Regel Banken oder sonstige Finanziers. Diese Zwischenhändler sind notwendig, damit sich die Verkäufer nicht auf eine rechtzeitige und problemlose Zahlung der Käufer verlassen müssen und dadurch ihr finanzielles Risiko verringern. Doch diese traditionelle Handelsfinanzierung bringt auch einige Unannehmlichkeiten mit sich, beginnend mit hohen Kosten, bürokratischem Aufwand, erheblicher Verzögerung bei der Abwicklung und schließlich auch einem nicht zu vernachlässigenden Betrugsrisiko aufgrund veralteter Systeme und manueller Bearbeitung.

In dieser Arbeit konzentrieren wir uns auf das Finanzinstrument Akkreditiv (Letter of Credit, L/C), welches zur Absicherung von Zahlungen im internationalen Handel eingesetzt wird. Basierend auf diesem Anwendungsfall schlagen wir eine Methode zur Bewertung von Blockchains für DeFi vor. Nachdem wir einen Prototyp eines typischen L/C-Workflows diskutiert und entworfen haben, implementieren wir diesen auf drei ausgewählten Blockchain-Plattformen. Die Bewertung der drei Implementierungen und dessen Plattformen erfolgt anhand zuvor festgelegter Kriterien. Diese Kriterien umfassen allgemeine Plattform-Eigenschaften wie den Transaktiondurchsatz und die damit verbundenen Kosten sowie entwicklungsspezifische Merkmale wie die Benutzerfreundlichkeit im Entwicklungsprozess. Das Resultat ist eine Reihung der Plattformen anhand ihrer Eignung in diesem Anwendungsfall.

Abstract

Blockchain technology facilities multi-party applications that do not require the parties to trust each other, that are failure-resistant due to their decentralized nature, and that provide a consistent view on the transaction history. These properties make blockchains attractive for decentralized finance (DeFi), and in particular for trade finance, where parties do not necessarily trust each other and aim at reducing their financial risks. Traditionally, intermediaries like banks or fiduciaries provide such services – along with several inconveniences like the increased risk of fraud due to antiquated systems and processes, considerable settlement delays, and high costs.

In this work, we focus on the financial instrument Letter of Credit (L/C), which is used to secure payments in international trade. We propose a method for evaluating blockchains for DeFi based on this use case. We adapt existing catalogues of criteria for platform evaluation to fit the development and operation of DeFi applications. After discussing and designing a prototype of a typical L/C workflow, we implement it on selected blockchain platforms. The evaluation rates the feasibility and usability of the development process.

Contents

CHAPTER 1

Introduction

1.1 Motivation

According to the most recent report of the WTO [53] world merchandise exports were valued at more than USD 19 trillions in 2018. Up to 80 percent depend on trade finance, a very low risk form of financing for banks [6]. The term *trade finance* describes financial instruments for trading partners that do not fully trust each other and try to reduce the damage in case of misbehaviour of the opposite side. One possibility is to use intermediates like banks or some other kind of financiers who provide guarantees or insurances. Letter of Credit (L/C) is such a financial instrument used to secure payment in international trade. While larger companies use L/C without much hesitation, smaller businesses usually do not employ the experts needed for that kind of transactions and therefore have a higher burden of entry to import or export internationally. Beside that, banks also prefer larger and more established companies for financing than smaller ones due the differences in numbers of transactions and amounts. [6, 22]

The involved processes are still largely paper-based and a cause of errors and inefficiency due to combinations of manual checking and involvement of various persons from possibly several countries [20]. An example highlighted by Capgemini Consulting [15] is the trillion doller syndicated loan market where participants still sent more than four million faxes in 2012. Antiquated systems and processes like this also increase the risk of fraud. Another inconvenience with traditional financial contracts is the settlement delay which is on average 48 days in Europe [15].

The importance and possibilities of digitalizing trade have long been recognized by banks but previous attempts to introduce paperless trade failed to gain relevance. A typical problem is the fragmentation into various platforms of different providers and the costs involved to support all of them. For international companies and banks with a worldwide presence it is economically easier to justify than for smaller independent traders or logistic firms. This resulted in digital islands without the ability to communicate via standard interfaces to the outside world [15, 24]. McKinsey [11] estimates the greatest impact of Distributed Ledger Technology (DLT) for trade finance will be in document handling for international trades. It will help to digitize paper-based documents and contracts like letters of credit, bills of landing or invoices.

The principle of DLT combines multiple aspects of computer science, mathematics and economics like distributed networking, cryptography, game theory, graph theory and stochastic [13]. One concept of DLT is the blockchain. It introduces trust-less systems without a single point of failure that are still transparent, irreversible and maintain a single point of truth. The blockchain enables numerous use cases in many different areas [29, 56].

Numerous slightly different DLT and blockchain projects emerged during the last few years. While common features like encryption, immutability and hashing are similar across most of them, a major distinction lies in the potential participants. Among the permission-less platforms, Bitcoin and Ethereum are the most prominent examples. As for the permissioned platforms, Corda and Hyperledger are interesting. Corda is a specialized distributed ledger platform created by R3 and a consortium of two hundred global financial institutions. Hyperledger is an open-source project overseen by the Linux Foundation with more than 270 organizations as official members. While Ethereum introduced a new programming language with Solidity, Corda and Hyperledger support Smart Contract (SC) written in Java. A SC is an automatically enforced program that exists and runs directly on the distributed ledger network. It revitalises the smart contract concepts introduced by Szabo [44]. Hyperledger uses the term chaincode interchangeably for what other DLT designs like Ethereum and Corda call SC. The focus of this thesis will be the usage of DLT in trade finance. [7, 14, 52, 12, 3]

1.2 Problem Definition

In summary there is a need for easier international trade finance without necessary including banks. DLT is able to introduce new options for smaller companies. The goal of this thesis is to provide prototypes of a typical workflow of trade finance (e.g. L/C) implemented in the compared distributed ledger platforms. The purpose of the following research questions is to analyse the differences and steps involved when setting up a trade via a smart contract using the introduced platforms without the need to fully trust each other.

RQ1 What are the differences when enforcing business agreements for trade finance using existing distributed ledger platforms?

RQ1.1 What are the main differences in implementing a typical workflow on the compared platforms?

RQ1.2 What are the cost differences between traditional trade financing and contracts implemented and executed on DLT?

1.3 Expected Results

The platforms in focus all differ in how modular they are, what their major usage is, who is able to participate, the programming languages, the consensus protocol and the possible throughput. Regarding the implementation of a typical use case, we determine the difference in efforts for the investigated platforms. Moreover, we provide a guide for developing SCs on the mentioned platforms and describe typical pitfalls. This will serve as a support for the selection of a platform. More specifically we will address if a permission-less or permissioned DLT fits the use case better and which limitations either have. Another interesting aspect is how hard it will be to implement the prototype with a relatively unknown programming language compared to a common one. The main focus thereby is on implementing prototypes. The results will be valuable to anyone interested in the currently most researched distributed ledger platforms for trade finance or intending to implement a similar smart contract on their own. The expected results comprise the following three parts:

- A comparison of the platforms with respect to trade finance
- A smart contract prototype of the same workflow for each platform
- A cost analysis in comparison to trade financing with banks

1.4 Related Literature

While many articles and papers analyse different distributed ledger or blockchain platforms, the comparison is mostly theoretical and none of them implement a prototype on multiple platforms. The authors typically focus on either the use cases of different platforms or the technical differences.

Cant et al. [15] did a quantitative analysis of smart contracts in addition to focus interviews with selected professionals in the banking and insurance industry. They did not focus on a specific platform and aim to highlight possible benefits of smart contracts and what needs to happen before the financial industry is able to adopt them.

Murshudli and Loguinov [35] analysed the issues that need to be addressed when digitalizing the international banking systems and focused on the possible economic benefits. They mentioned the rise of FinTech companies, how they threaten the existing banking system and how R3 (Corda) could allow to transfer the paper-based letter of credit process to the blockchain.

Belotti et al. [7] on the other hand published a guideline to choose which blockchain fits a project the best. In addition to Ismail and Materwala [29] they also take into consideration what the major usages of the described platforms are, modularity, architecture and throughput among others. Other theoretical comparisons can be found in the works of Saraf and Sabadra [42], Xu et al. [54] and Kim et al. [32].

Bogucharskov et al. [10] examine areas of blockchain application in trade finance and identify major aspects for increasing the productivity of the transaction process. They present how participants would interact with each other when using a blockchain based L/C and what kind of improvements it brings.

The authors of Chang et al. [16] focus on the dilemma of traditional international trade and design various blockchain based processes to digitalize it. One of the introduced designs is a L/C smart contract. A feasibility study is done by using use-case and activity diagrams. Furthermore a comparative analysis between the current trade process and the described models is done.

Chang et al. [17] explored the feasibility of a blockchain based L/C smart contract from a conceptual perspective. The goal is to increase the understanding of the DLT paradigm shift with a multi-case study and the role of blockchain L/C in achieving numerous targets in trade finance. The selected cases include projects on Ethereum, Hyperledger and Corda.

Blum [9] is a master thesis about a trade finance Solidity smart contract designed and analysed from a game theoretical point of view. In contrast to the other presented L/C smart contracts it removes the involved intermediaries of traditional L/C. It also covers the legal aspects in Switzerland.

In Vinayak et al. [50] the authors provide and explain the pseudo code of a European style call option smart contract on Ethereum which could be used for collateral contract services. The same authors describe in Vinayak et al. [51] how to set the network up and design a collateral service smart contract on Hyperledger Fabric.

1.5 Methodological Approach

Research in information systems uses two distinct paradigms, *behavioural science* and *design science*. Behavioural science has its roots in natural science methods. It starts with a hypothesis, the researcher tries to either prove or disprove it with collected data and in the end eventually evolves into a theory. The goal of design science on the other hand is to produce an artifact which is built and evaluated to solve a problem. Going through the process of developing and facing possible issues while doing so is a central part for gaining knowledge to improve the artifact [27]. The primary research method of the thesis is based on the guidelines and three cycle view proposed by Hevner et al. [27] and furthermore the framework for evaluating methods in a design science research project introduced by Venable et al. [49]. The main activities are:

- *Build*
  - *Systematic Literature Review* based on Kitchenham and Charters [33] to get an overview of the state of the art of blockchain development, to know typical processes for trade finance, get familiar with the terminology both in distributed ledgers and trade finance and to choose appropriate evaluation criteria for the prototypes.
  - *Implementation* of the same trade finance process on each of the introduced platforms
- *Evaluate* the implemented artifacts in terms of functionality, reliability, usability, costs and performance. The evaluation criteria will get more precise with each iteration of the cycle and are described in subsection 1.5.2.

1.5.1 Build

Scientific Literature Review

To build the knowledge base of the three cycle view we use Scientific Literature Review (SLR) as introduced by Kitchenham and Charters [33] using the electronic databases IEEE Xplore, ScienceDirect, Scopus and SpringerLink. The Search Query was modified to fit the different syntaxes, as shown in Table 1.1, and returns 333 results across all databases.

```
(ethereum
  OR corda
  OR hyperledger
  OR blockchain
  OR DLT
  OR "distributed ledger"
  OR "smart contract")
AND
("trade finance"
  OR "letter of credit")
```

Listing 1.1: Search Query

Pruning

Stage 1: Removing duplicates and non-sense. Based on the initial set of records from the databases we removed duplicates and obvious non-sense. Duplicates were identified by considering the authors and title of the paper. Obvious non-sense that got removed was for example the acronym page from Gabler Banklexikon (K – Z) (2020). This removed about 7% and resulted in 310 studies left.

Table 1.1: Search queries executed against databases

Electronic Database	Search Query	Records
IEEE Xplore	(("Full Text & Metadata":ethereum OR corda OR hyperledger OR blockchain OR "smart contract") AND "Full Text & Metadata":"trade finance" OR "letter of credit")	41
ScienceDirect	(ethereum OR corda OR hyperledger OR blockchain OR DLT OR "distributed ledger" OR "smart contract") AND ("trade finance" OR "letter of credit")	36
Scopus	ALL ((ethereum OR corda OR hyperledger OR blockchain OR dlt OR "distributed ledger" OR "smart contract") AND ("trade finance" OR "letter of credit"))	82
SpringerLink	(ethereum OR corda OR hyperledger OR blockchain OR DLT OR "distributed ledger" OR "smart contract") AND ("trade finance" OR "letter of credit")	174

Stage 2: Manual selection based on title. Studies were filtered by comparing their titles with the inclusion and exclusion criteria. This removed about 79% and resulted in 65 studies left. The applied inclusion and exclusion criteria:

Inclusion criteria

- Studies that report on applications and future trends of the blockchain
- Studies that address the usage of DLT in financial services
- Studies that involve smart contracts and L/C
- Studies that compare at least two of the platforms in focus

Exclusion criteria

- Studies that are not written in German or English
- Studies that involve Islamic finance
- Studies that focus on crypto currencies, Bitcoin or Initial Coin Offerings (ICOs)
- Studies that address supply chain traceability
- Studies involving the maritime industry

Stage 3: Manual selection based on abstract. Studies were filtered by comparing their abstract with the inclusion and exclusion criteria. This removed about 57% and resulted in 28 studies left.

Stage 4: Manual selection based on content. In the final stage we read the few remaining papers with the defined criteria and the goal of the thesis in mind. In the end we selected 10 relevant articles and books that are presented in Table 1.2.

Table 1.2: Selected knowledge base

Reference	**Type**	**Title**
Aggarwal et al. [1]	Article	Blockchain for smart communities: Applications, challenges and opportunities
Bogucharskov et al. [10]	Article	Adoption of blockchain technology in trade finance process
Chang et al. [17]	Article	Blockchain-enabled trade finance innovation: A potential paradigm shift on using letter of credit
Chang et al. [16]	Article	Exploring blockchain technology in international trade: Business process re-engineering for letter of credit
Liang [34]	Article	Blockchain application and outlook in the banking industry
Vinayak et al. [51]	Article	Design and Implementation of Financial Smart Contract Services on Blockchain
Travel and Mohanty [46]	Book	R3 Corda for Architects and Developers
Xu et al. [55]	Book	Architecture for Blockchain Applications
Sunyaev [43]	Book	Internet Computing
Bhogal and Trivedi [8]	Book	International Trade Finance

Implementation

Details about the implementation of the SC on the various platforms are found in chapter 5.

1.5.2 Evaluation

The design cycle is a constant iteration of constructing and evaluating the artifact. One purpose of evaluation is to verify if an instantiation of a designed artifact achieves its stated goal. Another objective is how well an artifact fulfils the requirements compared to other artifacts with similar purposes [48]. Evaluation is also used to identify weaknesses and areas of improvement and is a key principle when developing an artifact using the iterative build-evaluate cycle by Hevner et al. [27].

While evaluation is quite specific to the artifact Hevner et al. [27] states that "artifacts can be evaluated in terms of functionality, completeness, consistency, accuracy, performance, reliability, usability, fit with the organization, and other relevant quality attributes". Checkland [18] on the other hand proposed five properties to evaluate: efficiency, effectiveness, efficacy and in particular circumstances ethicality and elegance.

Different Design Science Research (DSR) authors have identified multiple methods of evaluation. While Hevner et al. [27] describes five classes of evaluation methods, Peffers et al. [39] splits evaluation into two parts, demonstration and evaluation. Other methods have been identified by Nunamaker Jr et al. [38] and Venable [48]. The authors of Venable et al. [49] developed a comprehensive framework for designing the evaluation methods used in a particular DSR project.

The first question when choosing the DSR evaluation strategy framework based on Venable et al. [49] is to differentiate between *ex ante* and *ex post*. The guideline states *ex ante* is used to evaluate partial or full prototypes while *ex post* is used for full instantiations. As we implement multiple prototypes in this thesis *ex ante* fits best. The next step is to decide between *naturalistic* and *artificial*. This depends on whether the stakeholders are real users or not and the potential conflicts emerging from that. We choose *artificial* as there are no real stakeholders involved and therefore the risk is low. The DSR evaluation method gets selected based on the properties *ex ante* and *artificial* and results in *Criteria-Based Evaluation*.

The criteria used to find the platform that fits the use-case the best are separated into two parts. The first part rates the actual platform itself while the second part focuses on the SC development. Properties and characteristics to evaluate DLT platforms are introduced in section 2.2, more specifically in Table 2.1 and Table 2.2.

General Platform Criteria

- *Performance* How long does it take to finalise a transaction? Scalability?
- *Confidentiality* Prevention of unauthorised information access?
- *Costs* What are the costs for participating in the network?
- *Governance* Open-source? Adoption of appropriate license necessary? How are decisions about changes to the platform made?

Prototype Development Criteria

- *Usability* Comprehensive documentation of the platform available? A lot of effort to set-up the development environment?
- *Functionality* Is is possible to implement all methods as specified in chapter 4?

- *Testability* How to test the correctness of a SC? Are there any official tools?
- *Flexibility* General-purpose or domain-specific programming language? Virtual machines running the nodes?

A platform will get one to three points for each criteria mentioned. A more detailed description about the reasoning and points awarded to the platforms is outlined in chapter 6. In the end we will sum up the points and rank the platforms from the highest to the lowest number.

1.6 Structure of the Work

The rest of this thesis is organized as follows:

Chapter 2 introduces basic terms involving international trade and DLT. L/C is the trade finance process in focus and a comprehensive description will highlight the steps of the participants involved and help to understand the problems of the traditional approach. It also clarifies the difference between distributed ledger technology and blockchain.

In **Chapter 3** we will have a look at promising platforms like Ethereum, Hyperledger Fabric and Corda. This chapter will focus on the technical differences and builds a foundation of knowledge for the smart contracts implemented later.

Chapter 4 presents a platform independent design of the L/C smart contract we will implement with all its stakeholders and components involved.

Chapter 5 describes the steps involved to implement the introduced smart contract on the various platforms and changes to the default design if required.

A comparison between the implemented prototypes and cost analysis is provided in **Chapter 6**.

Finally, **Chapter 7** summarizes the findings and closes with an outlook on future work.

CHAPTER 2

Fundamentals

In section 1.1 we briefly discussed the terms trade finance, DLT and SC. The goal of this chapter is to describe the involved processes and technologies in more detail.

2.1 Trade Finance

International trade plays a crucial role in the economy of many countries. When moving goods importers and exporters often share the same set of problems originating in the different legislations, practises and customs of the involved countries. Each side has their own concerns. While exporters want to be certain that they are paid after shipping their merchandise, importers want to make sure to receive exactly what has been ordered. Reasons for importing or exporting range profit to not enough supply or demand in the home market but often are more complex due the following risks:

- Geographical: due geographical reasons the buyer and seller are less likely to know each other
- Legal: often the customs and legal system is different between the involved actors
- Language: different languages require translations of the involved documents and could be a cause of misunderstandings
- Non-Payment: domestic sales have a lower risk of non-payment than international trade
- Money-bound: because of longer shipping times compared to local trades the money invested is typically restricting the cash-flow of the involved companies. While suppliers usually want payment before shipping the goods, importers prefer to be able to inspect the received order.

- Currency exchange: trading with partners in countries with volatile currency results in a risk for both sides
- Manufacturing: the buyer modifies or cancels the order after the manufacturer has already produced customized goods.

To fill the resulting gap of uncertainty commercial banks provide numerous products and therefore play an important role in foreign trade. Banks usually have branches in multiple countries or are at least partnered with a local bank. Because of a combination of the legal knowledge of the involved countries, practical experience in international trade and the general trustworthiness, banks are often chosen as intermediaries. There are various payment methods like Cash in Advance (CIA), Open Account (OA), L/C and many more. [6, 8, 22, 53]

2.1.1 Payment Methods

Cash in Advance

This payment method requires a payment made by the buyer before the goods are received or often before a shipment is even made. CIA removes the risk of non-payment for the seller and shifts the trade risks fully to the buyer. It also eliminates possible liquidity problems for the seller to manufacture or buy goods them-self. CIA is often used for customised goods as the seller would be in a disadvantaged strategic position after starting to invest on a buyer specific item. The buyer could try to renegotiate the price because they know the seller will not be able to sell the good with a similar price to other parties. [8, 25]

Open Account

When both involved parties trust each other, usually based on a common trade history, OA payment is often used. With OA the seller ships the goods and forwards documents of title (like Bill of Lading (B/L)) to the buyer before the payment is made. The payment is settled in the future, sometimes combining payment of regular shipments within a given interval to pay goods received during that period. This kind of payment method puts a lot of risk on the seller's side as they neither have any control over the goods nor have to trust the importer to pay. [8, 25]

Letter of Credit

L/C (also known as documentary credit) is a guarantee of payment issued by the buyer's bank after it got requested by the buyer and all requirements are met. Usually the seller also gets their bank involved in order to estimate the worthiness of the purchasers bank guarantee and to minimize possible exchange problems or political risks. The typical trade participants of a L/C process are sellers (exporters), buyers (importer), shippers (logistics carrier) and banks. The three major flows are money, documents, goods. Figure 2.1 illustrates the following steps involved in the process:

1. a sale contract between the seller and buyer is established
2. the buyer applies at their bank to issue an L/C to the seller's bank
3. buyer's bank issues L/C to seller's bank
4. seller's bank notifies seller about L/C
5. seller checks the received L/C for correctness of described goods
6. seller arranges shipment to the buyer
7. the carrier provides the seller shipping documents like the B/L
8. B/L is considered a document to claim ownership of the goods and gets sent to seller's bank
9. the buyer's bank pays the seller through their bank in exchange for the B/L
10. the buyer pays their bank in exchange for the B/L to be able to claim the goods
11. the carrier ships the goods to the buyer
12. the carrier checks if the buyer has the correct B/L

Several types of L/C exist. A clean L/C for example does not require any document, like the terms and conditions to fulfil, other than a written demand for payment by the seller. For financial institutes it is not safe to agree to such a type of L/C as neither the goods nor the documents of title for the goods (like B/L come into their possession. In international trade L/C often is used as a documentary proof of trust between the involved parties where each participant has to provide a number of logistic related documents. Another type is the irrevocable L/C. The issuing bank is not able to alter or cancel its terms without the consent of all participants, including the beneficiary. Payment is usually made when the agreed terms and conditions are met. Difficulties in communication and coordination caused by the number of involved participants in various types of cross border business activities with unfamiliar counter parties often result in issues such as tedious document processing, higher issuance cost and forgery. [8, 16, 17, 25]

Letter of Credit. *A letter of credit is a payment mechanism used in international trade that provides the seller a guarantee from the buyer's bank.*

---

[1] Icons made by Pixel perfect and Good Ware from www.flaticon.com

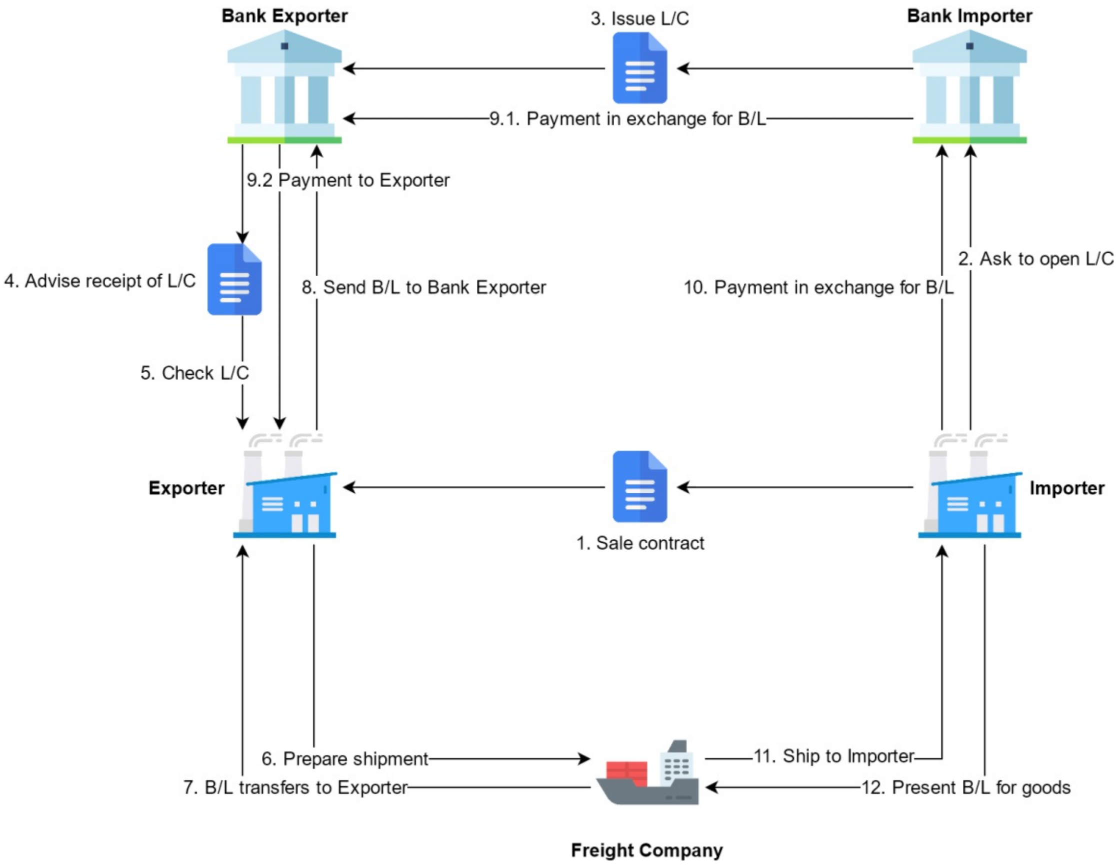


Figure 2.1: L/C process (based on Chang et al. [16]) [1]

2.2 Distributed Ledger Technology

This section introduces the technical background of DLT and presents innovations since the introduction of the blockchain concept. In recent years DLT gained major attention in the media and academic field caused by the creation of the Bitcoin blockchain in the year 2009. It is one of the most promising innovations in the IT and has to potential to change the economy, society and industry. The principle of DLT combines multiple aspects of computer science, mathematics and economics like distributed networking, cryptography, game theory, graph theory and stochastic. Together these components enable temper-proof transactions and safeguard the data from manipulation and theft. [13, 43]

Managing and especially storing data is an essential part of many applications. In most of the cases relational databases are used for this purpose which defined tables and relationships between those. Transactions handle changes to the data as a set of CRUD (create, read, update, delete) operations and are executed in isolation to enable a rollback in case of failure. In general there are three types of databases: *centralised*, *decentralised* and *distributed*. In *centralised databases* data is stored at a single place and while easier to maintain the drawbacks are performance and availability. Availability describes the probability of the database to function as expected at a random time. The performance bottleneck becomes apparent when there is a high load of requests and a single machine is not able to handle it well enough. *Distributed databases* on the other hand do not have a central storage. The data is stored on multiple connected devices, often in different locations. The structure is hierarchical with some nodes fulfilling a coordinator role. Another explanation is a hierarchical organisation of centralised databases. In *decentralised databases* this organisational bottleneck is removed. Replications of the data is stored across numerous independent machines and if one database fails the other devices in the network are able to handle the request and provide a similar result. Each decentralised database is also a distributed database. We also call the involved machines nodes. In a decentralised database the nodes form a mesh of connected devices as no hierarchical structure exists. In this context distributed refers to the distribution of data across multiple devices while decentralised refers to the distribution of control of the data. [43, 55]

Decentralised Database. *A decentralised database is a type of database where data is replicated across multiple storage devices (nodes) with equal rights.*

Although physically separated, a CRUD operation on a distributed database should always return the same result. To achieve a consistency of the stored data the nodes are logically centralised while the architecture is distributed. The nodes are separated and therefore some form of communication to be able to synchronise the data must exist. Algorithms and protocols managing the synchronisation with possible unreliable nodes in mind are called consensus mechanisms. [43, 55]

Consensus Mechanism. *A consensus mechanism is designed to achieve agreement on the respective state of replications of stored data between a distributed database's nodes under consideration of network failures. [43]*

A special type of a distributed database is the *distributed ledger*. In contrast to distributed databases the only allowed operation is to add new data, therefore deleting or updating should not be possible. The used consensus mechanisms are designed to be able to handle the third Byzantine failure. The term Byzantine failure takes it name from the "Byzantine Generals Problem" and describes a situation in which actors must agree on a joint strategy to avoid catastrophic failure of the system but some of the actors are unreliable. The first type of Byzantine failure is a unreachable or crashed node. If a node sends ambiguous responses and the monitoring system is not able to determine the nodes status the second type is present. While the first two types are often technical problems, the third one describes nodes with malicious intentions such as trying to store incorrect data. [43, 55]

Distributed Ledger. *A distributed ledger is a type of distributed database that assumes the presence of nodes with malicious intentions. A distributed ledger comprises a ledger's multiple replications in which data can only be appended or read. [43]*

Because of the application of game theory [2] to consensus finding in distributed databases DLT allows unknown or untrusted nodes to run the distributed ledger. One of the major innovations of DLT is the reliable synchronisation of a distributed ledger while the set of nodes is dynamically changing and respecting the Byzantine failures. [43, 55]

Distributed Ledger Technology. *DLT enables the realization and operation of distributed ledgers, which allow benign nodes, through a shared consensus mechanism, to agree on an (almost) immutable record of transactions despite Byzantine failures and eventually achieving consistency. [43]*

One concept of DLT is the blockchain. Introduced and implemented in 2009 with the goal of accessible digital money transfer without the involvement of banks, Bitcoin is often considered DLT generation [3] 1.0.

Blockchain. *A blockchain is a distributed ledger that is structured into a linked list of blocks. Each block contains an ordered set of transactions. Typical solutions use cryptographic hashes to secure the link from a block to its predecessor. [55]*

Figure 2.2 shows the concept of blocks in a blockchain. The cryptographic method of hashing guarantees that a preceding block is unchanged. If the block *n* gets changed, for example someone tries to manipulate the data, the hashing algorithm returns another value for the changed block and thus the link between the blocks *n* and *n+1* breaks as n+1 refers to a block with the old hash value.

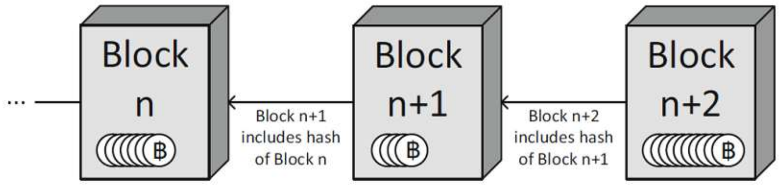


Figure 2.2: Blockchain data structure (adapted from Xu et al. [55])

Blockchain System. *A blockchain system consists of:*

(i) a blockchain network of machines, also called nodes;

---

[2] Myerson [36] describes Game Theory as "(...) study of mathematical models of conflict and cooperation of intelligent rational decisionmakers".

[3] In literature the term blockchain generation is frequently used although the blockchain is only a concept of DLT and a more broad definition would be DLT generation.

(ii) a blockchain data structure, for the ledger that is replicated across the blockchain network. Nodes that hold a full replica of this ledger are referred to as full nodes;

(iii) a network protocol that defines rights, responsibilities, and means of communication, verification, validation, and consensus across the nodes in the network. This includes ensuring authorization and authentication of new transactions, mechanisms for appending new blocks, incentive mechanisms (if needed), and similar aspects.[55]

The incentive mechanism used in most public blockchains is proof-of-work. The so called miners create new blocks by solving cryptographic puzzles. In case of Bitcoin by finding a value for a field in the block header, the nonce. To keep the average time between blocks around ten minutes the threshold is adjusted over time. As the transaction of a mined block has no input this is also the way new BTC tokens are minted and is the form of payment for miners. After a new block is successfully mined it gets broadcasted over the whole network and each full node holds a replica of the most current state of the ledger. [37]

Public Blockchain. *A public blockchain is a blockchain system that has the following characteristics:*

(i) it has an open network where nodes can join and leave as they please without requiring permission from anyone;

(ii) all full nodes in the network can verify each new piece of data added to the data structure, including blocks, transactions, and effects of transactions; and

(iii) its protocol includes an incentive mechanism that aims to ensure the correct operation of the blockchain system including that valid transactions are processed and included in the ledger and that invalid transactions are rejected.[55]

People soon realised that crypto-currencies are not the only field of application of blockchains and developed distributed ledgers with the capability of storing additional data. As a consequence Smart Contracts got introduced as it was now possible to store applications inside transactions. The simple scripting language of Bitcoin is not Turing complete [4] and therefore not classified as SC capable. The Ethereum blockchain addressed this weakness of Bitcoin and introduced the possibility of more powerful applications with the introduction of Turing complete SCs on a distributed ledger. Ethereum is a DLT generation 2.0 blockchain as it is not limited to the usage as a crypto currency but also allows the distributed ledger to be utilised in other ways, for example to store data. While Bitcoin and Ethereum offer pseudo anonymity [5] and are permission-less

---

[4] Turing completeness describes a systems ability to simulate any other Turing machine. That means the system is able to decide other data manipulating rule sets based on the current state. The usage of loops in programming is enabled by Turing completeness.

[5] Pseudo anonymity describes when persons are at first not directly identifiabl for example because of the usage of number plates (or in the blockchain context wallet addresses) but in the end no real anonymity is given.

blockchains, some use cases require more confidentiality, increased flexibility or a higher throughput. [43, 54]

One of the main differences between DLT designs is the question who is able to participate. On *public* blockchains like Bitcoin and Ethereum the underlying network allows unknown nodes to join and to contribute to the distributed ledger. This usually results in a large number of nodes maintaining the network and causes a high level of availability as each node stores a replication of the ledger. On the other hand, in *private* DLT designs nodes are identifiable and such networks typically require some form of verification to be able to join the network. Such distributed ledgers are useful when multiple companies cooperate and the involved data should not be accessible by everyone. Caused by the difference in number of nodes involved between the two blockchain designs the consensus mechanisms vary. Whereas the consensus algorithms in public designs must be highly scalable, their counterparts in private designs are often designed for a smaller number of nodes and focus on other aspects. In addition to the distinction between public and private distributed ledgers, the consensus finding and transaction validation can also be assigned to a subset of nodes. DLT designs where only a subset of nodes are involved in the consensus finding process are called *permissioned*. The advantage of using only a smaller known group of nodes to validate transactions is a much higher throughput, up to multiple thousands per second. *Permissionless* DLT designs do not require the identity of the node to be known because every node has the same permissions. As nodes constantly join and leave the network the consensus finding is much less finite and usually more probabilistic than in permissioned networks. The large public DLT designs Bitcoin and Ethereum are permissionless. HyperLedger Fabric introduces this functionalities and is considered DLT generation 3.0. [14, 43, 3, 54, 55]

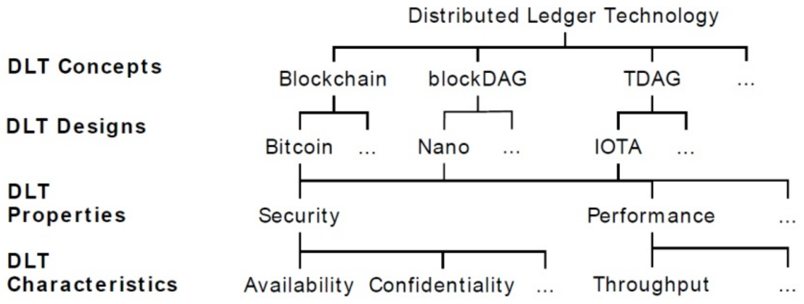


Figure 2.3: Schematic overview of the DLT terminology (adapted from Kannengießer et al. [31])

Figure 2.3 provides a visualisation of various DLT terms. While Ethereum and Bitcoin both emply the DLT concept blockchain their implementation differs because the there is a trade-off of the characteristics applied. Characteristics have interdependencies and can either be complementary (high level of transparency helps with audit-ability) or contradictory (high availability needs multiple replications which decreases consistency).

Kannengießer et al. [31] identified six DLT properties as described in Table 2.1 and numerous characteristics grouped by properties in Table 2.2.

Transaction. *A transaction updates the state recorded on a blockchain. [55]*

The state information of transactions of crypto-currency blockchains is usually about the transfer of tokens between accounts. On blockchains such as Ethereum a transaction possibly contains code, variables or the results of function calls. Once a transaction reaches a mining node it is verified an possibly included in the newly mined block.

Even though the blockchain brings new possibilities, it is not practical for every usage scenario. Many use cases do not require the decentralized and immutable aspects. For example computation heavy programs are not the target market for blockchain applications.

Table 2.1: DLT properties (adapted from Kannengießer et al. [31])

Property	**Description**
Security	Preservation of confidentiality, integrity, and availability of information.
Performance	The accomplishment of a given task measured against standards of accuracy, completeness, costs, and speed.
Usability	The extent to which a DLT design can be used by specified users to achieve specified goals with respect to effective-ness, efficiency, and satisfaction in a context of use.
Development Flexibility	The possibilities offered by a DLT design for maintenance and further development.
Level of Anonymity	The degree to which individuals are not identifiable within a set of subjects.
Institutionalization	The emerging embedding of concepts and artifacts (here DLT) in social structures.

2.2.1 Smart Contracts

Instead of simply storing data, some DLT designs are able to store code and execute it. We call those programs smart contracts. Buterin [14] describes the term DLT as "*more complex applications involving having digital assets being directly controlled by a piece of code implementing arbitrary rules*". The idea to represent contracts in soft and hardware with pre-defined programmed conditions was introduced by Szabo [44] about 20 years earlier but did not find much traction and just gained relevance with the recent blockchain developments. Szabo [44] suggested to translate clauses of traditional contracts into code and use hard- or software that is capable of self-enforcing them. The

Table 2.2: Extract of DLT characteristics (adapted from Kannengießer et al. [31])

Property	**Characteristics**
Security	*Availability.* Availability is the probability that a system can be accessed when needed. *Confidentiality.* Prevention of unauthorised information access and release. *Consistency.* Strong consistency means that all nodes store the same data in their ledger at the same time. *Integrity.* Integrity requires that information is protected against unauthorized modification or deletion as well as irrevo-cable, accidental, and undesired changes by authorized users.
Performance	*Block Creation Interval.* The time between the creation of consecutive blocks (only in DLT designs using blocks). *Scalability.* The capability of a DLT design to handle an increasing amount of workload or its potential to be enlarged to accommodate that growth. *Throughput.* The number of transactions validated and appended to the ledger in a given time interval. *Transaction Validation Speed.* Duration required for verifying transaction validity.
Usability	*Costs.* Costs related to the implementation and usage of a DLT design, including software development and operational costs. *Ease of Node Adoption.* The ease of preparing a new or failed device to be added to the DLT design in the role of a validating node or a consuming terminal device. *Ease of Use.* The ability to easily access and work with the DLT design.

goal was to minimize the number of involved intermediaries and to remove the need of trust. Despite using the term smart contract the applications are neither smart (as using some form of artificial intelligence) nor legally enforceable contracts. A smart contract acts like a self-operating computer program that automatically executes when specific conditions are met.

Once deployed on the blockchain, smart contract code is immutable and is executed exactly as programmed. Applications are called Decentralized Applications (DApps) when its central logic is deployed as smart contracts. Smart contracts are used to build all kind of DApps ranging from creating digital assets like crypto-curriencies to creating uncensorable web applications to build decentralized autonomous organisations. Details about the differences in implementing and using SCs on Ethereum, Corda and Hyperledger Fabric will be presented in chapter 3 and chapter 5.

Smart Contract. *A smart contract is an application that manipulates digital assets based on pre-defined conditions implemented in code and stored on a DLT.*

CHAPTER 3

Platforms

3.1 Ethereum

The smart contract platform with the highest market capitalization as of today, in terms of capital employed, is Ethereum and was developed by Buterin [14] and Wood [52] in 2014. Ethereum is a public, Proof-of-Work (PoW)-based [1] permissionless blockchain-based, distributed platform and offers a built-in and Turing complete smart contract functionality, allowing everyone to write decentralized applications. As already mentioned one difference between a first generation DLT like Bitcoin and second generation DLTs like Ethereum is the support for programmable transactions. Smart contracts are considered first-class elements in Ethereum and the code is executed in a decentralized virtual machine, known as Ethereum Virtual Machine (EVM). DApps on Ethereum are written in the high-level programming language Solidity, which is an object-oriented language with predefined instructions and later compiled into a low-level stack-based bytecode language. In the end, a smart contract is a series of sequentially executed instructions by the EVM. [14, 52, 55]

Buterin [14] states five principles the design of Ethereum follows:

1. *Simplicity*: the Ethereum protocol should be as simple as possible, even if that results in some inefficiencies
2. *Universality*: instead of having any features the platform provides a Turing-complete language and the programmer is able to build whatever he needs
3. *Modularity*: the protocol should be as modular as possible to be able to change and upgrade some parts without requiring modifications on others.

---

[1] PoW is a consensus mechanism in which the miners compete with each other to solve a mathematical problem.

4. *Agility*: details of the protocol may change in the future, for example if improvements in scalability or security are found.
5. *Non-discrimination and non-censorship*: the protocol should not restrict of actively prevent specific use cases.

3.1.1 Ethereum Protocol

To address the long delays of a Bitcoin transactions, Ethereum is designed to have relatively short time intervals between blocks, on average around 15s. As a result the possibility that multiple competing blocks are created concurrently is much higher. This becomes a problem because most of the public blockchains use the Nakamoto consensus, where processing nodes treat the longest chain of blocks as the authoritative chain, the main chain. Blocks that were successfully created by a miner and already propagated and verified by some nodes but eventually dismissed because another longer chain becomes the main chain are called stale blocks. [55]

A way to settle this problem is the usage of a modified Greedy Heaviest Observed Subtree (GHOST) protocol. With GHOST miners reference stale blocks (so called ommer blocks) to add weight to their chain. In contrast to other protocols the decision which chain becomes the main chain is not only based on the length but also on the weight. Referenced ommer blocks contribute to that weight. In addition of allowing shorter inter-block times and higher throughput by recognising concurrent work the network keeps the miners financial interest high as the miners of ommer blocks also receive a (reduced) block reward. [14]

In Ethereum the consensus finding and validation of transactions is combined as follows:

1. Every node builds a block containing valid transactions. Validation of transactions is done by pre-executing them.
2. The node tries to solve the PoW puzzle.
3. If the puzzle got solved the node publicizes the block to the network.
4. Receiving nodes validate the solution of the puzzle and all transactions contained in the block.

In the end every node in the network repeats the executions done in step 1 sequentially.

3.1.2 Smart Contract

To deploy a SC on the Ethereum blockchain a contract-creation transaction is used. The payload of the transaction includes the code. After the contract is successfully created it is identified by a contract address on the blockchain. Every smart contract contains:

- executable code
- internal storage to store its state
- Ether, the token of Ethereum and therefore a balance

Smart contracts must be externally invoked and to interact with a deployed contract users have to call the defined functions by sending contract-invoking transactions to the contracts address. It is also possible to invoke functions of other smart contracts. An invoke transaction contains:

1. the interface of the invoked function
2. the parameters in the data payload
3. some amount of Ether to pay for the execution

3.1.3 Gas

Each node in the network has to execute every operation within a contract and consumes the computational resources of the miner. To compensate the miners and to limit the use of resources the concept of *Gas* is used. Gas is a proportional fee. The gas has to be paid by the Ethereum account sending the transaction. Every transaction has a fixed gas cost and additional variable costs dependent on the data and the number bytecode instructions executed of called functions. Gas cost is paid with Ether and the user sets how much he is willing to pay when creating a transaction. A transaction also has a gas limit parameter to be able to set an upper bound on how much gas can be consumed by the transaction and acts as a safe-guard to prevent draining the whole balance due programming errors or malicious intent. [14]

3.2 Hyperledger Fabric

Another well known open-source blockchain project is Hyperledger. It is an umbrella project and since 2015 hosted by the Linux Foundation. The members included are well known technology platform providers (Intel, Cisco, Red Hat, ...), finance firms (J.P. Morgan, SWIFT, Sberbank, ...) academic institutions (Cambridge, UCLA) and other various well known corporations (IBM, SAP, Accenture, ...). The Hyperledger project is home to numerous frameworks with the core topic focusing on blockchain. The tools range from programs visualising data on the blockchain to developing DApps. One of the areas explored is Hyperledger Fabric, a business blockchain framework with the goal of developing modular blockchain-based applications. [3]

Based on the definitions in section 2.2 Hyperledger Fabric is classified as a private and permissioned blockchain. In Androulaki et al. [3] the authors present some limitations of other permissioned blockchains. In particular:

- the consensus mechanism is hard-coded although it is well established knowledge that there is no "one-size-fits-all" consensus protocol
- a fixed, non-standard or domain-specific language is used to write smart contracts
- all transactions must be executed by all peers in sequential order which limits performance
- every node executing every smart contract is problematic for confidential data

Fabric supports the execution of DApps written in standard general-purpose programming languages (Go, Java, JavaScript, TypeScript) consistently across the globe and is therefore also described as the first distributed operating system for permissioned blockchains. Smart contracts are hosted in Docker container to isolate them from each other and called *chaincode*. Maintaining and participating in the network is exclusive to members enrolled via a trusted Membership Service Provider (MSP). In contrast to Ethereum all nodes of a Fabric network have known identities. Hyperledger Fabric and Ethereum use the virtual computer model which models the database as a in-memory state of a global computer [26]. The architecture is split into the following components to keep the modularity as high as possible [3]:

- *Ordering service*: broadcasts state updates and establishes consensus on the order of transactions
- *MSP*: links the peers with cryptographic identities and is used to keep the blockchain private
- *Peer-to-Peer gossip service*: distributes the blocks output to all peers
- *Smart contract*: is executed within a container environment and does not have direct access to the ledger state.
- *Ledger*: maintained by each peer locally as a append-only blockchain and as a snapshot of the most recent state in a key-value store.

Previous blockchains (e.g. Ethereum) implemented the *order-execute* architecture, which means the network first orders the transactions using a consensus protocol and then executes [2] them in the same order on all nodes sequentially. While this architecture is conceptually simple it has various drawbacks: [3]

---

[2] the transaction execute step is often also called transaction validation

- *Sequential Execution*: limits the throughput and as the throughput is inversely proportional to the execution latency this becomes a performance bottleneck. That kind of architecture is also prone to denial-of-service (DoS) attacks by introducing smart contracts that slow down the whole blockchain. Ethereum solved this problem by introducing subsection 3.1.3.
- *Non-deterministic* [3] *code*: One of the most fundamental basics in a blockchain is that all peers hold the same state. If code execution is non-deterministic the distributed ledger "forks". This problem is usually addressed by introducing domain-specific programming languages that are limited to deterministic expressions.
- *Confidentially*: In classic permission-less, public blockchains every node has access to the whole smart contract code, transaction data and ledger state. In some use-cases there is a need to restrict this. Some possible solutions are cryptographic techniques like zero-knowledge proofs but they add additional overhead to the transaction. Another solution is to execute the code only on a small set of trusted nodes and propagate the same state to all peers.

To solve the described problems, Fabric introduces a three-phase *order-execute-validate* architecture. A crucial part in this architecture is the *endorsement* step. An endorsement policy is chosen by permissioned administrators and part of the validate phase. Possible policies are "three out of five" or "$(A \land B) \lor C$". Transactions are sent to nodes specified by the endorsement policy, executed and their output recorded. Unlike other blockchains, Fabric does not order the transactions by input but instead by output combined with state dependencies. The ordered transactions are broadcasted to all peers. Each peer validates the state change with respect to the specified endorsement policy. The transactions are all validated in the same order across the peers and the validation is deterministic. A node in the Hyperledger Fabric network takes up one of three roles: client, peer, orderer. [3, 55]

- *Client*: submits transaction proposals for execution and broadcasts the messages for ordering. A client connects to peers to be able to communicate on behalf of the end user with the blockchain.
- *Peer*: receive ordered state updates from the orderers and are used to execute transaction proposals and validate the transaction. While only a subset of peers (endorser peers) executes all transaction proposals, all of them maintain the blockchain and record all transaction in form of a hash chain and a brief representation of the latest ledger state.
- *Ordering Service Nodes (OSN)*: establish the total order of all transactions while being unaware of the application state. The orderers are not involved in the

---

[3] a deterministic system will always produce the same output given the same starting conditions. Randomness is not involved in the development of the future states.

execution or the validation of transactions. As a result of separate ordering nodes the consensus protocol in Fabric is highly modular and easy to replace.

A transaction starts in form of a transaction proposal created and sent by a client to specific peers. This peers are the endorsers of the transaction. Their task is to verify the signature of the transaction initiator and execute the referenced chaincode functions. The response of the proposal contains the signatures of the endorsers and is sent back to the client. The client adds the endorsement into the payload of the transaction and broadcasts it to an OSN. The OSN orders all transactions into blocks and sends the blocks to all peers, endorsing and non-endorsing. Once a peer receives a transaction it checks if the endorsement policy is fulfilled by checking if the attached endorsement signatures match the specified endorsement peers. Data integrity is confirmed by controlling if the data that was read during the chaincode execution has been changed since the time of endorsement. If the data has been changed by another transaction the current transaction is marked as invalid and the client is notified. After successful checks the transaction is committed to the blockchain. [3, 55]

The interaction of the involved nodes is visualised in Figure 3.1.

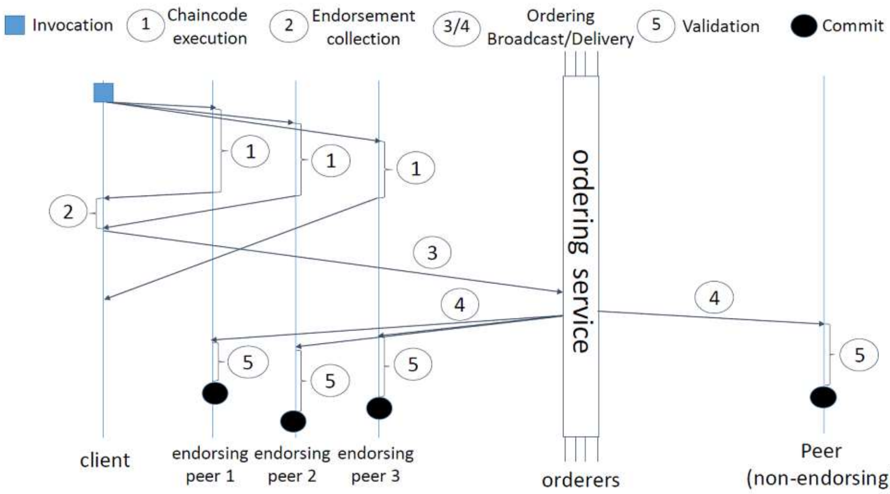


Figure 3.1: Hyperledger Fabric transaction flow (adapted from Androulaki et al. [3])

Fabric also supports channels. A channel is a private layer of communication between specific network members with a separate blockchain ledger. Only invited organisations are able to participate. Other members of the network are not able to see channels they are not invited in. Figure 3.2 provides an overview of the interactions between the different actors in a Hyperledger Fabric network.

---

[4] https://www.itransition.com/blog/hyperledger-fabric-blockchain-payments-problems-and-solutions

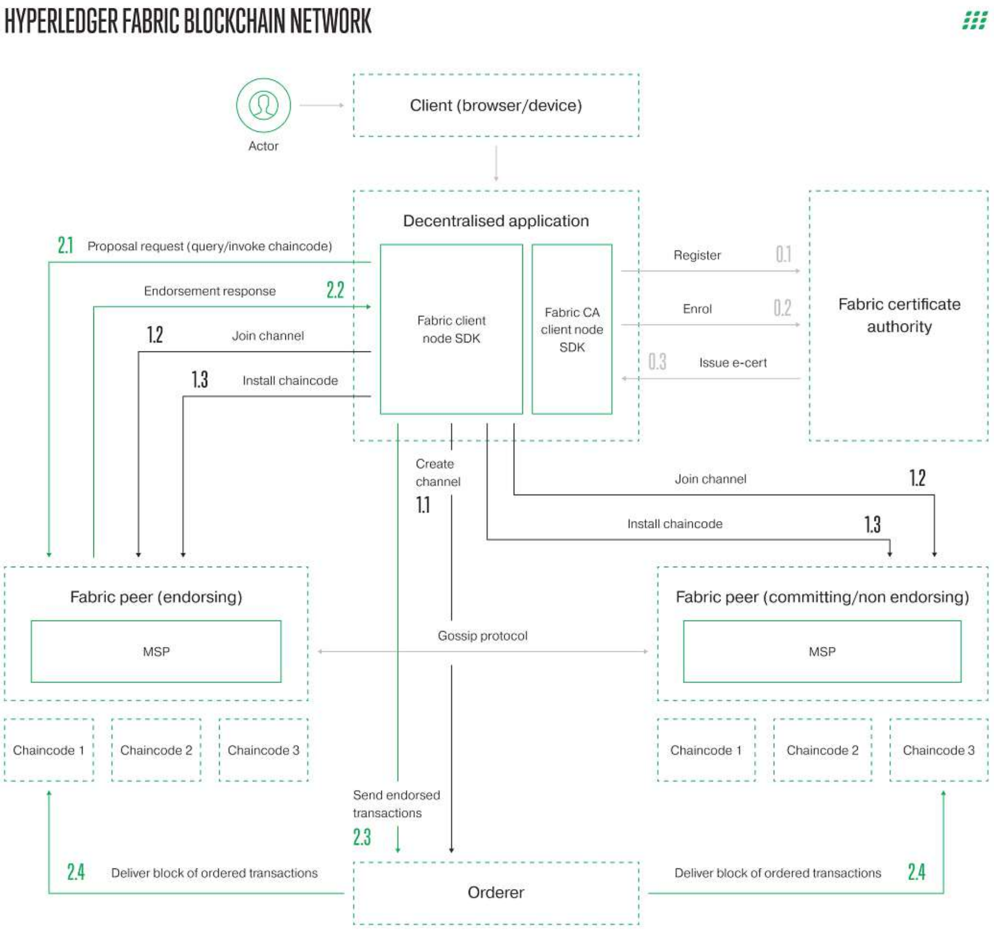


Figure 3.2: Hyperledger Fabric network [4]

3.3 Corda

While Hearn [26] and Travel and Mohanty [46] describe Corda as "blockchain inspired decentralised database platform with some novel features", Valenta and Sandner [47] categorises it as a "specialized distributed ledger platform for the financial industry". Based on the definitions in section 2.2 the mode of operation in Corda is private and permissioned, similar to Hyperledger Fabric. The initial design of the platform was motivated by use cases in the financial service industry and the long term vision is a global logical ledger where all economic partners are able to interact with each other. [26, 12]

While the transaction data in most distributed ledger platforms is broadcasted globally, Corda uses small multi-party sub-protocols called *flows* for all communications. In Corda peers communicate on a point-to-point basis. Flows are light-weight processes used to coordinate the interactions peers require to reach consensus on the ledger. In a flow only relevant parties are involved, therefore message recipients must be specified. Each flow participant has to verify and sign the transaction. This concept is visualised in Figure 3.3. [26, 12]

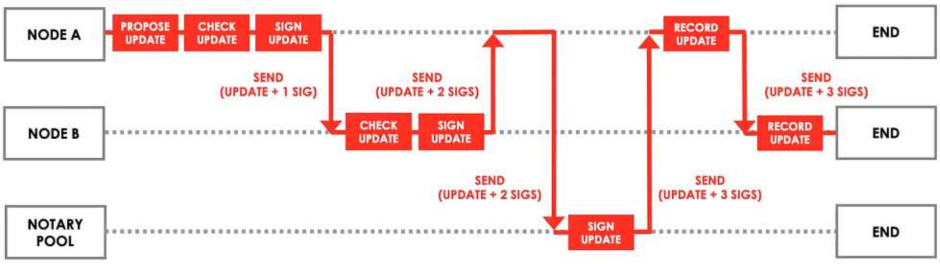


Figure 3.3: Corda flow execution (adapted from the official Corda documentation)

In contrast to Hyperledger Fabric and Ethereum, Corda uses the Unspent Transaction Output (UTXO) model to represent the database, similar to Bitcoin. In the UTXO model transactions contain inputs and outputs, resulting in a set of immutable rows keyed together. The data consumed and added by transactions is called *states*. States are the atomic unit of information and are either current (unspent) or consumed (spent). A transaction reads zero or more states (inputs), consumes zero or more of the read states and creates zero or more states (outputs). SC are responsible to either accept or reject a transaction proposal. [26, 12]

Another significant distinction to other DLT platforms is the design choice to include possible legal references directly in states. Beside containing the contract code used to verify the transaction the output state could also contain a reference (hash) of a legal document. These references do not have any legal weights by them-self but in financial use cases it is expected to include a legal contract that takes precedence over the software implementation. Figure 3.4 shows a cash issuance transaction and provides a detailed view at its output state. The combination of the involved flows, smart contract, state objects, UI components and wallet plugins is called Corda Distributed Application (CorDapp). [26, 12]

The following components are used in the network [26, 46, 12]:

- *Nodes*: are application servers which load CorDapps and give them access to the network, a relational database, key signing and a vault (called wallet in blockchains). One unique feature of a Corda node is the direct SQL access to the ledger. Thus CorDapps are able to e.g. query particular points in time or join company internal data with ledger stored data using a widely understood language. Node communicate with each other using serialised Java types and the binary Advanced Message Queue Protocol (AMQP) of the Apache Artemis message broker.

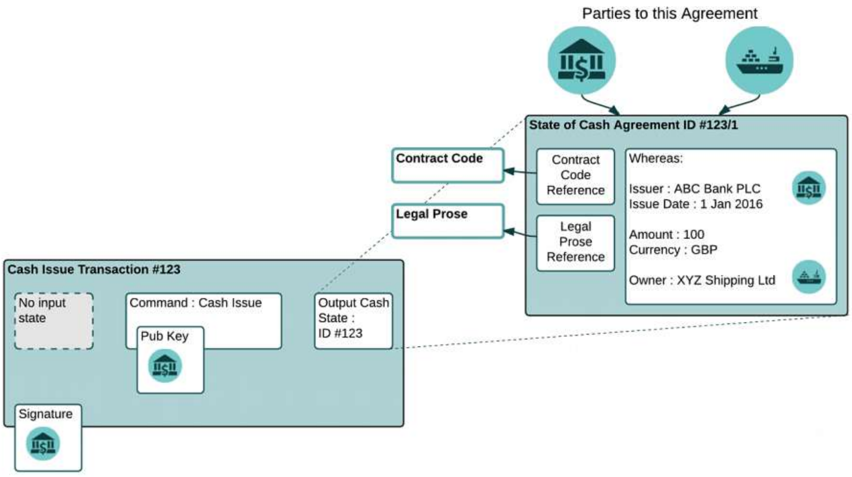


Figure 3.4: Corda cash issuance transaction (adapted from Hearn [26])

- *Identity Service*: ensures all nodes have known identity certificates and therefore is fundamental for keeping the Corda network private. Every discussed DLT system uses asymmetric keys in form of public and private keys to control data access. While the keys are cryptographically secure they are often a target of phishing attacks because they are difficult to remember. Corda's solution to the problem is the usage of X.509 certificates which link keys to human readable names. The names are only used to resolve to public keys or IP addresses and are not required for transactions. These certificates have to be signed by the network operator and allow the node to take part in the top layer of the network.

- *Network Map Service*: distributes information about each node in the network such as its IP addresses, supported protocol version and which identity certificates are hosted. The published data is signed by the identity keys the node hosts and therefore no trust in the network map service is required. Usually only nodes with valid certificates are listed by the network map and by default nodes only accept connections from other nodes included in the network map. This allows to remove malicious actors by revoking their certificates.

- *Notary Service*: consists of one ore more mutually distrusting parties which use a Byzantine Fault Tolerant (BFT) consensus algorithm to perform the role of miners in blockchains. The notary component is pluggable and thus the consensus algorithm exchangeable. Transactions are submitted to notaries and signed or rejected. To avoid double spending the notary service signs a transaction only if all input states are unconsumed. Once a transaction is signed by a notary it is final.

- *Oracle Service*: is used to sign transactions containing statements about the world outside the ledger only if the statements are true. Optionally the statements them-self are also provided. Oracles are used to keep the network fully deterministic. Smart contracts should use an oracle instead of fetching the data directly from the internet because everyone must be able to compute the exact same thing, even in the future when certain websites are no longer reachable.

3.3.1 Corda Enterprise

Corda Enterprise is an interoperable and fully compatible commercial edition of the open-source Corda platform with enhanced features and support by R3, the company backing the development of the platform. The core functionality of both versions are the same. In general the enterprise edition the stability and scalability are higher because of the support of multiple nodes for high availability and recovery compared to the single node support of the core version. It is also possible to use Microsoft SQL Server or Oracle as the vault database instead of just Postgres. The enterprise edition also provides a better performance because of a multi-threaded flow state machine.

CHAPTER 4

Smart Contract Design

This chapter discusses a possible smart contract design for a trade finance transaction. The result is a combination of the payment methods mentioned in subsection 2.1.1 without the involvement of banks. If the platform supports a built-in currency the smart contract will use it. The goal of the process is to keep the interactions as simple as possible and the number of involved stakeholders minimized while reducing the risk for all involved parties. Our SC and process design eliminates the disadvantages of the mentioned payment methods by using an immutable distributed ledger. Banks are not necessary with our design as the SC itself acts as a depositary. The need to trust each other is removed by verifiable code. Figure 4.1 illustrates the following steps:

1. The importer (buyer) places an order request and informs the exporter (seller) which good he wants to buy, the quantity, delivery date and other data useful for that kind of business.
2. In the next step the exporter either deploys the SC to the DLT or reuses an already deployed SC and adds the order. The state of the order is `CREATED`.
3. Now the importer verifies the data of the order and deposits the agreed amount of money if he accepts the conditions. The state of the order changes to `CONFIRMED`.
4. The seller is able to monitor the state of the order and forwards the goods to a freight company after the money is deposited. In this step the freight company gets added to the SC. The state of the order changes to `SHIPPED`.
5. The freight company delivers the goods to the buyer. The freight company is also paid via the SC and therefore has a high interest in delivering the goods to the buyer getting a signature of arrival.

6. The importer and freight company use the SC to sign the arrival of goods. The state of the order changes to `DELIVERED`.

7. Once the state is `DELIVERED` the deposited money is paid to the freight company and the exporter. The state of the order changes to `CLOSED`.

The importer or exporter are able to cancel the order while the status is `CREATED`. If the goods do not arrive within the agreed time specified in the SC conditions the order will get cancelled automatically. Once the goods are shipped user triggered cancellation is not possible any more. In case of cancellation the deposited money will be paid back to the importer and the state of the order is set to `CANCELLED`.

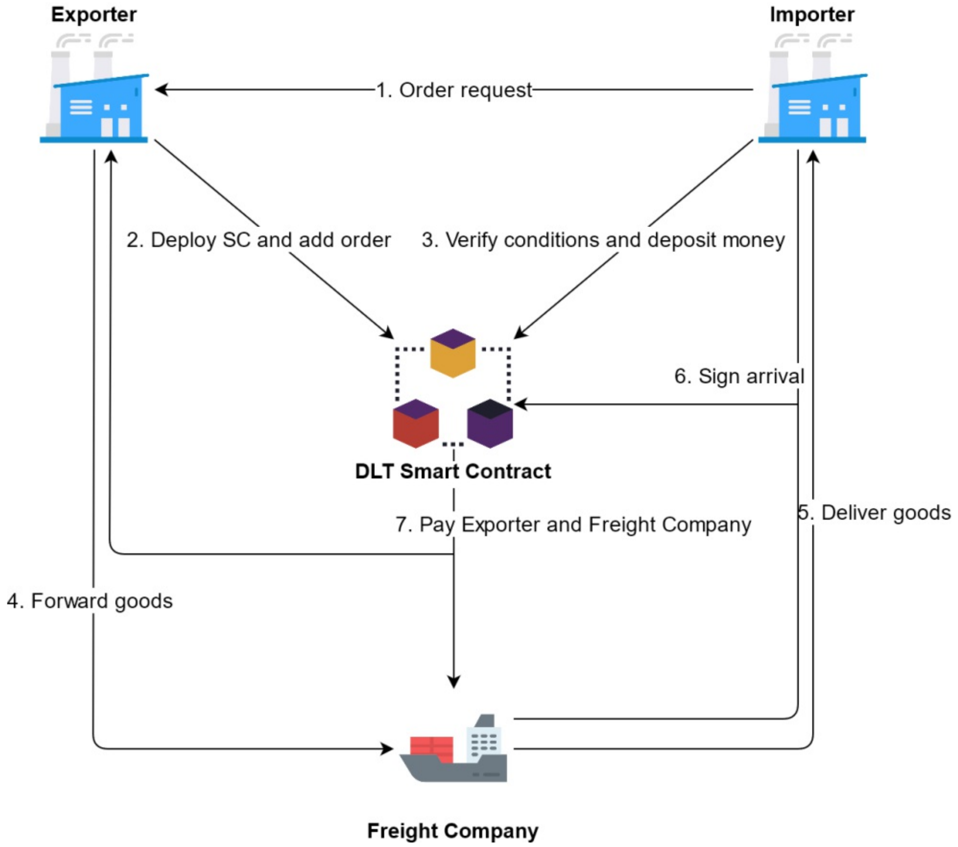


Figure 4.1: Process flow of DLT based trade finance [1]

---

[1] Icons made by Pixel perfect and Good Ware from www.flaticon.com

4.1 Methods

The algorithms in this section present the pseudo code of the smart contract. The prototypes in the following chapters will keep the structure as close as possible to the pseudo code.

Algorithm 4.1: *addOrder* method

```
Input: order id, address or name of the buyer, product id, quantity, price
       (including shipping costs), shipping costs, shipping address, latest
       possible delivery date
Result: a new order is added to the distributed ledger
Required: the seller called this method;
Required: an order with the passed id does not exist yet;
Required: the price must be greater or equal to the shipping costs;
```

Algorithm 4.2: *confirmOrder* method

```
Input: order id, order price as token value (if the distributed ledger has a
       built-in crypto-currency)
Result: order state is set to CONFIRMED
Required: an order with the passed id does exist;
Required: the order state is CREATED;
Required: the buyer called this method;
Required: the order price matches the token value (if the distributed ledger has
  a built-in crypto-currency);
```

Algorithm 4.3: *cancelOrder* method

```
Input: order id
Result: order state is set to CANCELLED, deposited funds are sent back
Required: either the seller or the buyer called this method;
Required: an order with the passed id does exist;
Required: the order state is either CREATED or CONFIRMED;
```

Algorithm 4.4: *deliveryDatePassed* method

```
Input: order id
Result: order state is set to PASSED, deposited funds are sent back
Required: an order with the passed id does exist;
Required: the order state is either CREATED, CONFIRMED or SHIPPED;
Required: the delivery date specified is in the past;
Required: the freight company did not sign the arrival yet;
```

Algorithm 4.5: *shipOrder* method

```
Input: order id, address or name of the freight company, tracking code of the
       shipment
Result: order state is set to SHIPPED, the freight company and tracking code
        are added to the order
Required: an order with the passed id does exist;
Required: the seller called this method;
Required: the order state is CONFIRMED;
```

Algorithm 4.6: *signArrival* method

```
Input: order id
Result: order state is set to SIGNED if the freight company and the buyer
        signed the arrival
Required: an order with the passed id does exist;
Required: either the buyer or freight company called this method;
Required: the order state is SHIPPED;
if method is called by the buyer then
    set buyerSigned flag of the order to true;
end
if method is called by the freight company then
    set freightSigned flag of the order to true;
end
if buyerSigned and freightSigned flags are true then
    set the state of the order to DELIVERED;
    pay the seller and freight company (if the distributed ledger has a built-in
      crypto-currency);
end
```

CHAPTER 5

Prototypes

5.1 Ethereum

The software used to develop and test the Ethereum SC is listed in Table 5.1. The Solidity code of the prototype is appended in Appendix 7.

Table 5.1: Software used for the Ethereum prototype

Software	**Version**	**Description**
Ubuntu	20.04 LTS	Operating system
Truffle Suite	5.1.48	Development environment and testing framework for Ethereum
Solidity	0.6.12	Programming language, Compiler (part of Truffle Suite)
Node.js	12.19.0	JavaScript runtime environment
Web3.js	1.2.1	Collection of libraries that allows to interact with an Ethereum node using HTTP, IPC or WebSocket
Ganache	2.4.0	Local blockchain and GUI hat displays the Ethereum transaction history and chain state.
Visual Studio Code	1.47.0	Source code editor

The prototype was designed with various behavioural and security patterns in mind as described in Fichtinger [23]. The SC acts as a state machine, which means it changes its behaviour depending on its internal state. The functionality and possible function calls provided to the stakeholders differ between the states. We have introduced the state `NONE` as first state instead of `CREATED` to be able to differentiate whether an order exists. The reasoning is because Solidity initialises every variable with 0. Thus, if we check the state of an not yet created order it will always return the first state. The different states are defined as follows:

```
    enum States {
        NONE,
        CREATED,
        CONFIRMED,
        SHIPPED,
        DELIVERED,
        CLOSED,
        CANCELLED,
        PASSED
    }
```

5.1.1 Relevant files

```
.
├── contracts
│   ├── Migrations.sol
│   └── TradeFinanceContract.sol
├── migrations
│   ├── 1_initial_migration.js
│   └── 2_trade_finance_migration.js
├── test
│   └── tradefinance.js
└── truffle-config.js
```

5.1.2 Functions

addOrder

This function is used by the seller to add a new order. It takes as input parameters the order identifier, Ethereum account address of the buyer, sold product identifier, quantity, total price with shipping, shipping address, latest delivery date and the shipping costs. The price parameter is the total sum of items price plus shipping costs. The latest delivery date is a date the buyer and seller agreed to, after which the deposited Ether will be sent back to the buyer if the order did not arrive yet. The `onlySeller` modifier allows only the created of the SC to call the function. As already mentioned, if an order does not exist yet the state is `NONE`. Therefore the modifier `atState(_orderId, States.NONE)` prevents overwriting of another order. Another check is if the price is greater or equal to the shipping costs as this parameter is a sum of both. The SC maintains a map of all added orders, accessible with the order ID. After the code within the function run without problems the state of the order transitions to `CREATED` because of the modifier `transitionNextState(_orderId)`.

```
    function addOrder(
```

```
        uint256 _orderId,
        address payable _buyer,
        uint256 _productId,
        uint256 _quantity,
        uint256 _price,
        string memory _shippingAddress,
        uint256 _latestDeliveryDate,
        uint256 _shippingCosts
    )
        public
        onlySeller
        atState(_orderId, States.NONE)
        transitionNextState(_orderId)
    {
        require(
            orders[_orderId].orderId != _orderId,
            "An order with this ID already exists."
        );
        require(
            _price >= _shippingCosts,
            "The price must be greater or equal to the shipping costs."
        );

        orders[_orderId].orderId = _orderId;
        orders[_orderId].buyer = _buyer;
        orders[_orderId].productId = _productId;
        orders[_orderId].quantity = _quantity;
        orders[_orderId].price = _price;
        orders[_orderId].shippingCosts = _shippingCosts;
        orders[_orderId].shippingAddress = _shippingAddress;
        orders[_orderId].latestDeliveryDate = _latestDeliveryDate;
        orderCount++;
        emit Log(_orderId, "Order has been added");
    }
```

confirmOrder

Only the buyer, the account the seller added when creating the order, is able to confirm an order. The keyword `payable` allows the function the receive Ether and is needed because the buyer deposits the amount specified in the order. The map `balances` is used to keep an overview who owns which amount of Ether deposited.

```
    function confirmOrder(uint256 _orderId)
        public
        payable
        onlyBuyer(_orderId)
        atState(_orderId, States.CREATED)
        transitionNextState(_orderId)
    {
        require(
            orders[_orderId].price == msg.value,
```

```
            "Not enough Ether sent to cover the price of the order."
        );
        balances[orders[_orderId].buyer] += orders[_orderId].price;
        emit Log(_orderId, "Order has been confirmed and money deposited");
    }
```

cancelOrder

Both, the seller and the buyer, are able to cancel the order because of the modifier `onlySellerOrBuyer(_orderId)`. For an order to be cancelled the state has to be either `CREATED` or `CONFIRMED`. If the order is cancelled and the state was `CONFIRMED` the deposited Ether will be sent back to the buyer.

```
    function cancelOrder(uint256 _orderId) public onlySellerOrBuyer(_orderId)
         {
        require(
            orders[_orderId].state == States.CREATED ||
                orders[_orderId].state == States.CONFIRMED,
            "Function cannot be called at this state."
        );

        if (orders[_orderId].state == States.CONFIRMED) {
            orders[_orderId].state = States.CANCELLED;
            balances[orders[_orderId].buyer] -= orders[_orderId].price;
            orders[_orderId].buyer.transfer(orders[_orderId].price);
        } else {
            orders[_orderId].state = States.CANCELLED;
        }
        emit Log(_orderId, "Order has been cancelled");
    }
```

shipOrder

The seller is able to add the account of the freight company used for shipping and the tracking code of the order after the buyer deposited Ether to cover the price of the order and the order state successfully changed to `CONFIRMED`.

```
    function shipOrder(
        uint256 _orderId,
        address payable _freightCompany,
        string memory _trackingCode
    )
        public
        onlySeller
        atState(_orderId, States.CONFIRMED)
        transitionNextState(_orderId)
    {
        orders[_orderId].freightCompany = _freightCompany;
        orders[_orderId].trackingCode = _trackingCode;
```

```
        emit Log(_orderId, "Order has been shipped");
    }
```

deliveryDatePassed

Everyone is able to invoke this function. It is used to refund the deposited money to the buyer once the agreed delivery date is not met. The payout process is only started if the date passed, the order state is not `DELIVERED` and the freight company did not sign the arrival. The last check is to prevent fraud from the buyer by simply not signing the arrival.

```
    function deliveryDatePassed(uint256 _orderId) public {
        require(
            block.timestamp >= orders[_orderId].latestDeliveryDate,
            "Delivery date did not pass yet."
        );
        require(
            orders[_orderId].state < States.DELIVERED,
            "Order got already delivered."
        );
        require(
            orders[_orderId].freightSigned == false,
            "Refund not possible as the freight company already signed the
               arrival."
        );

        orders[_orderId].state = States.PASSED;
        if (orders[_orderId].state >= States.CONFIRMED) {
            balances[orders[_orderId].buyer] -= orders[_orderId].price;
            orders[_orderId].buyer.transfer(orders[_orderId].price);
        }
        emit Log(
            _orderId,
            "Order has been cancelled due passed delivery date."
        );
    }
```

signArrival

The freight company and buyer must sign the arrival of the shipment. This prevents malicious intent of either side. Due to the physical contact during delivery it is part of the duty of the freight company to get the buyer to sign the arrival. Singing the arrival is of great interest to the freight company as otherwise they will not get paid. After both parties signed the arrival the order transitions into the state `DELIVERED` and the function `payout` is called.

```
    function signArrival(uint256 _orderId)
        public
```

```
        onlyFreightCompanyOrBuyer(_orderId)
        atState(_orderId, States.SHIPPED)
    {
        if (msg.sender == orders[_orderId].buyer) {
            orders[_orderId].buyersigned = true;
            emit Log(_orderId, "Order arrival has been signed by the buyer");
        }

        if (msg.sender == orders[_orderId].freightCompany) {
            orders[_orderId].freightSigned = true;
            emit Log(
                _orderId,
                "Order arrival has been signed by the freight company"
            );
        }

        if (orders[_orderId].buyersigned && orders[_orderId].freightSigned) {
            nextState(_orderId);
            emit Log(
                _orderId,
                "Order arrival has been signed by the buyer and freight company"
            );
            payout(_orderId);
        }
    }
```

payout

This function is called automatically after the buyer and freight company signed the arrival of the shipment. It transfers the amount of Ether specified in the order as shipping costs to the freight company and the price minus shipping cost to the seller. Finally, the order transitions into the state `CLOSED`.

```
    function payout(uint256 _orderId)
        private
        atState(_orderId, States.DELIVERED)
        transitionNextState(_orderId)
    {
        balances[orders[_orderId].buyer] -= orders[_orderId].price;
        balances[seller] =
            balances[seller] +
            orders[_orderId].price -
            orders[_orderId].shippingCosts;
        balances[orders[_orderId].freightCompany] += orders[_orderId]
            .shippingCosts;

        seller.transfer(
            orders[_orderId].price - orders[_orderId].shippingCosts
        );
        orders[_orderId].freightCompany.transfer(
```

```
            orders[_orderId].shippingCosts
        );

        emit Log(_orderId, "Payout finished.");
    }
```

5.1.3 Testing

Steps to set-up the working environment after installing the software mentioned in Table 5.1:

- Create an empty project directory and run `truffle init` to initialize a new truffle project.
- Start Ganache, create a new workspace and link it with the previously created Truffle project.
- Place the code of Appendix 7 in the sub directory *contracts* of the project directory and name it *TradeFinanceContract.sol*.
- In the project directory run `truffle compile` to compile the .sol file.
- The next step is to deploy the smart contract to the Ethereum network. We are using a local blockchain while developing, thus the default settings of truffle do not need to be changed. The output of `truffle migrate -reset` should look like this:

```
Compiling your contracts...
===========================
  Fetching solc version list from solc-bin. Attempt #1
> Compiling ./contracts/TradeFinanceContract.sol
  Fetching solc version list from solc-bin. Attempt #1
> Artifacts written to /media/fichtinger/Data/GoogleDrive/github/code/
    ethereum/build/contracts
> Compiled successfully using:
   - solc: 0.6.12+commit.27d51765.Emscripten.clang



Starting migrations...
======================
> Network name:    'ganache'
> Network id:      5777
> Block gas limit: 6721975 (0x6691b7)


1_initial_migration.js
======================

```

```
   Replacing 'Migrations'
   ----------------------
   > transaction hash:    0x5af68fa21d4330f9e0b1a5ec823cd3e2390cee4db948c051f145a737cd9b34c8
   > Blocks: 0            Seconds: 0
   > contract address:    0xe072EE78056437801299a856d4bd58e5334A4C0a
   > block number:        77
   > block timestamp:     1602778993
   > account:             0xb72c72b67aBFA77f20077873862C175F71cd2a07
   > balance:             107.57062826000000019
   > gas used:            159195 (0x26ddb)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.0031839 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:           0.0031839 ETH


2_trade_finance_migration.js
============================

   Replacing 'TradeFinanceContract'
   --------------------------------
   > transaction hash:    0xd5028b516327a4907ac875751d938d227955a0bb699577bcf0f00a23c2febbaa
   > Blocks: 0            Seconds: 0
   > contract address:    0x939fe49a52b9Be9ED00d9E2d1CA2DfcC15389C88
   > block number:        79
   > block timestamp:     1602778993
   > account:             0xb72c72b67aBFA77f20077873862C175F71cd2a07
   > balance:             107.52649324000000019
   > gas used:            2164413 (0x2106bd)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.04328826 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.04328826 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.04647216 ETH
```

Manual

Once the smart contract is deployed to the Ethereum network we are able to interact with it. For this we have to open a console via `truffle console`.

- Before we start we link the default accounts to our seller, buyer and freight company.

  ```
  let accounts = await web3.eth.getAccounts()
  let seller = accounts[0]
  let buyer = accounts[1]
  let freightCompany = accounts[2]
  ```

- We initialise an instance variable to be able to call the functions more easily.

  ```
  TradeFinanceContract.deployed().then(inst => {instance = inst})
  ```

- Then we add a new order.

  ```
  instance.addOrder(1, buyer, 100, 2, web3.utils.toWei("10", "ether"), "
      Karlsplatz 13, 1040 Wien", 1594771200, web3.utils.toWei("2", "ether"
      ), {from: seller})
  ```

- Now the buyer has to confirm the order and send enough ether to cover the costs.

  ```
  instance.confirmOrder(1, {from: buyer, value: web3.utils.toWei("10", "
      ether")})
  ```

- After that the seller is able to ship the order and adds the freight company used and tracking code to the contract.

  ```
  instance.shipOrder(1, freightCompany, "1AXCAW311", {from: seller})
  ```

- The last step of the buyer and freight company is to sign the arrival of the goods. After both parties signed the payout process will be invoked.

  ```
  instance.signArrival(1, {from: buyer})
  instance.signArrival(1, {from: freightCompany})
  ```

- It is possible to cancel the order using the following command. Cancellation is possible if the state of the order is either `CREATED` or `CONFIRMED`.

  ```
  instance.cancelOrder(1, {from: seller})
  ```

Automated

Truffle also supports the execution of tests written in either Solidity or JavaScript. For JavaScript testing the Mocha[1] framework is used. The following code snippet demonstrates the *addOrder* test.

---

[1] `https://mochajs.org/`

```
...
    it("add order test", () => {
        let instance;

        return TradeFinanceContract.deployed()
            .then(inst => {
                instance = inst;
                return instance.addOrder(1, buyer, 100, 2, web3.utils.toWei("10", "ether"), "Karlsplatz 13, 1040 Wien", 1594771200, web3.utils.toWei("2", "ether"), { from: seller });
            })
            .then(() => instance.getOrderCount())
            .then(orderCount => {
                assert.equal(
                    orderCount.toNumber(),
                    1,
                    "the order count after adding an order was not 1"
                );
            })
            .then(() => instance.getOrderState(1))
            .then(orderState => {
                assert.equal(
                    orderState.toNumber(),
                    1,
                    "the order state after adding was not CREATED (1)."
                );
            })
    });
...
```

To start all defined tests we simply execute `truffle test` in the terminal. We have also extened the Mocha test framework with the ETH Gas Reporter[2] to be able to estimate the costs of the method calls. The output should look like this:

```
Using network 'test'.

Compiling your contracts...
===========================
  Fetching solc version list from solc-bin. Attempt #1
> Everything is up to date, there is nothing to compile.

  Contract: TradeFinanceContract
     check test environment
     create order test (215094 gas)
     confirm order test (256442 gas)
     sign arrival test (493366 gas)
     delivery date passed test (367840 gas)

<Table 5.2>

  5 passing (4s)
```

---

[2] https://www.npmjs.com/package/eth-gas-reporter

Table 5.2: Deployment and execution costs of our Ethereum prototype

Solc version: 0.6.12	Optimizer enabled: false	Runs: 200	Block limit:	6718946 gas		
Methods	20 gwei/gas	327.74 eur/eth				
Contract	**Method**	**Min**	**Max**	**Avg**	**# calls**	**eur (avg)**
Migrations	setCompleted	-	-	27338	2	0.18
TradeFinanceContract	addOrder	200094	215094	206113	5	1.35
TradeFinanceContract	confirmOrder	41348	56348	48848	4	0.32
TradeFinanceContract	deliveryDatePassed	-	-	50679	1	0.33
TradeFinanceContract	shipOrder	-	-	75671	2	0.50
TradeFinanceContract	signArrival	51470	124735	100313	3	0.66
Deployments					**% of limit**	
Migrations		-	-	159195	2.4 %	1.04
TradeFinanceContract		-	-	2164413	32.2 %	14.19

5.2 Hyperledger Fabric

In contrast to the Ethereum blockchain, Hyperledger Fabric does not require a token for transactions. It does not have a built-in native crypto-currency like Ethereum's token Ether. Therefore a solution based on Fabric only simplifies the process and is not able to remove the involvement of banks. In general, assets on a Hyperledger blockchain are useful because they are redeemable for something with real world value. The redeemability is usually agreed on in a traditional paper contract. An example is a contract where all involved parties agree that the outcome of commiting to pay on a certain chaincode is a legally binding debt. As a result the SC design has to be adapted. The software used to develop and test the Hyperledger Fabric SC is listed in Table 5.3.

Table 5.3: Software used for the Hyperledger Fabric prototype

Software	**Version**	**Description**
Ubuntu	20.04 LTS	Operating system
cURL	7.68.0	Tool for transferring data using various network protocols
Docker	19.03.12	OS-level virtualisation to deliver software in packages called containers
Docker Compose	1.26.2	Tool for defining and running multi-container Docker applications
Hyperledger Fabric	2.2.0	DLT platform
Hyperledger Fabric CA	1.4.8	Certificate Authority for Fabric
Node.js	12.8.1	TypeScript runtime environment
Visual Studio Code	1.47.0	Source code editor

5.2.1 Installation

Before we are able to deploy the SC we have to set-up the test network.

- The first step is to clone the Hyperledger Fabric samples repository which comes with an already pre-configured test-network.

  ```
  git clone https://github.com/hyperledger/fabric-samples.git
  ```

- To install the docker images of the latest production release of the Fabric platform the following command is used. After everything has been downloaded the executables have to be added to the environment path.

  ```
  curl -sSL https://bit.ly/2ysbOFE | bash -s
  ```

  Alternatively, to install exactly the same versions as mentioned in Table 5.3:

```
curl -sSL https://bit.ly/2ysbOFE | bash -s -- 2.2.0 1.4.8
```

- Now we are able to create a new channel within the network. This channel is only usable by our three organisations, the seller, buyer and freight company. We will be operating from the root of the `test-network` subdirectory in our local clone of the `fabric-samples`. Using the following script brings the network up with one channel named `mychannel`.

```
./network.sh up createChannel
```

- The default test-network contains a channel with two organisations, Org1 and Org2. Our SC design involves three different parties and therefore we have to add a third organisation to the channel. The file `startFabric.sh` in our `code/fabric/trade-finance` directory provides an all-in-one solution for creating the network with three organisations (named seller, buyer and freight), creating a channel, adding all three parties to the channel and deploying our SC to the peers.

5.2.2 Relevant files

Based on the `fabric-samples` the following files have to be added or changed (as shown in our github repository) for the script to work properly:

```
.
├── test-network
│   ├── configtx
│   │   └── configtx.yaml
│   ├── docker
│   │   ├── docker-compose-ca.yaml
│   │   ├── docker-compose-couch.yaml
│   │   └── docker-compose-test-net.yaml
│   ├── network.sh
│   ├── organizations
│   │   ├── ccp-generate.sh
│   │   ├── ccp-template.json
│   │   ├── ccp-template.yaml
│   │   ├── cryptogen
│   │   │   ├── crypto-config-buyer.yaml
│   │   │   ├── crypto-config-freight.yaml
│   │   │   ├── crypto-config-orderer.yaml
│   │   │   └── crypto-config-seller.yaml
│   │   └── fabric-ca
│   │       └── registerEnroll.sh
│   ├── scripts
│   │   ├── createChannel.sh
│   │   ├── deployCC.sh
│   │   └── envVar.sh
└── trade-finance
    ├── application
```

```
│   ├── buyer
│   │   └── java
│   │       ├── pom.xml
│   │       ├── src
│   │       │   ├── main
│   │       │   │   └── java
│   │       │   │       └── org
│   │       │   │           └── example
│   │       │   │               ├── AddToWallet.java
│   │       │   │               ├── ClientApp.java
│   │       │   │               └── Order.java
│   │       │   └── test
│   │       │       └── java
│   │       │           └── org
│   │       │               └── example
│   │       │                   └── ClientTest.java
│   ├── freight
│   │   └── java
│   │       ├── pom.xml
│   │       ├── src
│   │       │   ├── main
│   │       │   │   └── java
│   │       │   │       └── org
│   │       │   │           └── example
│   │       │   │               ├── AddToWallet.java
│   │       │   │               ├── ClientApp.java
│   │       │   │               └── Order.java
│   │       │   └── test
│   │       │       └── java
│   │       │           └── org
│   │       │               └── example
│   │       │                   └── ClientTest.java
│   └── seller
│       └── java
│           ├── pom.xml
│           ├── src
│           │   ├── main
│           │   │   └── java
│           │   │       └── org
│           │   │           └── example
│           │   │               ├── AddToWallet.java
│           │   │               ├── ClientApp.java
│           │   │               └── Order.java
│           │   └── test
│           │       └── java
│           │           └── org
│           │               └── example
│           │                   └── ClientTest.java
├── chaincode
│   └── typescript
│       ├── package.json
│       ├── src
│       │   ├── index.ts
```

```
    │       │   ├── order.ts
    │       │   └── trade.ts
    │       ├── tsconfig.json
    │       └── tslint.json
    ├── networkDown.sh
    └── startFabric.sh
```

- The file `registerEnroll.sh` generates the certificates of the users based on the specified Certificate Authority (CA) of each organisation. To be able to restrict access to certain methods we use the Attribute-Based Access Control [3] in Hyperledger Fabric. Two things have to be changed in the script. First, we have to add an affiliation called *seller*. We could get a finer restriction based on departments and for example add *seller.sales* to have even more control about the function calls.

  ```
  fabric-ca-client affiliation add seller --tls.certfiles ${PWD}/
      organizations/fabric-ca/seller/tls-cert.pem
  ```

  The second step is to define which department a user is part of when registering them at the CA.

  ```
  fabric-ca-client register --caname ca-seller --id.name user1 --id.secret
       user1pw --id.type client --id.affiliation seller --tls.certfiles ${
      PWD}/organizations/fabric-ca/seller/tls-cert.pem
  ```

- The created test-network runs locally with several docker containers. The peers, CA, and applications communicate via DNS, therefore we have to edit the `hosts` file of our operating system to redirect the calls correctly:

  ```
  127.0.0.1       peer0.seller.example.com
  127.0.0.1       peer0.freight.example.com
  127.0.0.1       peer0.buyer.example.com
  127.0.0.1       orderer.example.com
  ```

After executing the listed steps the test-network contains one peer of each organisation, one CA of each organisation, one SC instance deployed to each organisation and one orderer.

5.2.3 Functions

We first describe the most important functions of our chaincode (SC), written in TypeScript and found in `trade-finance/chaincode/typescript/src/trade.ts` in our repository. The structure of most methods is similar to the ones we've implemented in Solidity. Later we comment on the application which does not run on the DLT and is written in Java.

---

[3] https://medium.com/coinmonks/attribute-based-access-control-abac-in-hyperledger-fabric-1eb81330f67a

restrictedCall

For our Ethereum smart contract we used modifiers to restrict the possible function calls for certain users. In Hyperledger Fabric modifiers do not exist but the concept to restrict the calls is still similar. As already mentioned in subsection 5.2.1 the certificate of the user contains the attribute *affiliation*. We are able to check the value of the certificate of the user within our chaincode and determine if he is allowed to call this function or not.

```
private restrictedCall(ctx: Context, allowedAffiliation: string) {
  if (!ctx.clientIdentity.assertAttributeValue("hf.Affiliation", allowedAffiliation)) {
    throw new Error("Only users with affiliation " + allowedAffiliation + " are allowed to call this function");
  }
}

private restrictedCall2(ctx: Context, allowedAffiliation1: string, allowedAffiliation2: string) {
  if (!ctx.clientIdentity.assertAttributeValue("hf.Affiliation", allowedAffiliation1) && !ctx.clientIdentity.assertAttributeValue("hf.Affiliation", allowedAffiliation2)) {
    throw new Error("Only users with affiliation " + allowedAffiliation1 + " or " + allowedAffiliation2 + " are allowed to call this function.");
  }
}
```

createOrder

Despite allowing different data types as parameters, the arguments passed are always of the data type string and have to be manually converted. That kind of conversion is done in lines 17-20. Line 11 calls our `restrictedCall` method and allows only users with affiliation *seller*. In lines 12-15 we use `ctx.stub.getState` to query the ledger and check if an order with the passed *orderId* is already stored. After parsing the date and creating the *Order* object we finally store it at the ledger in line 45.

```
 1 public async createOrder(ctx: Context,
 2   _orderId: string,
 3   _productId: number,
 4   _quantity: number,
 5   _price: number,
 6   _shippingCosts: number,
 7   _shippingAddress: string,
 8   _latestDeliveryDate: string) {
 9   console.info("============= START : Create Order ===========");
10
11   this.restrictedCall(ctx, "seller");
12   const orderAsBytes = await ctx.stub.getState(_orderId);
13   if (orderAsBytes.length > 0) {
14     throw new Error("An order with ID " + _orderId + " does already exist");
15   }
```

```

  _productId = Number(_productId);
  _quantity = Number(_quantity);
  _price = Number(_price);
  _shippingCosts = Number(_shippingCosts);

  if (_price < _shippingCosts) {
    throw new Error("The price must be greater or equal to the shipping costs.
    ");
  }

  var splittedDate = _latestDeliveryDate.split("-"); // date given in yyyy-mm
    -dd format
  var parsedDate = new Date(parseInt(splittedDate[0]), parseInt(splittedDate
    [1]) - 1, parseInt(splittedDate[2]));
  //console.info("parsedDate:" + parsedDate.toLocaleString());

  const order: Order = {
    docType: "order",
    state: State.CREATED,
    orderId: _orderId,
    productId: _productId,
    quantity: _quantity,
    price: _price,
    shippingCosts: _shippingCosts,
    shippingAddress: _shippingAddress,
    latestDeliveryDate: parsedDate,
    trackingCode: undefined,
    buyerSigned: undefined,
    freightSigned: undefined
  };

  await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
  console.info("============= END : Create Order ===========");
}

```

cancelOrder

The method to cancel the order is again very short, it only contains two checks. We first restrict the calls to only allow users from the organisations seller and buyer. Then we verify that the state of the order is either `CREATED` or `CONFIRMED`. If both conditions are met we set the state to `CANCELLED` and put the order back onto the ledger.

```
public async cancelOrder(ctx: Context, _orderId: string) {
  console.info("============= START : cancelOrder ===========");

  this.restrictedCall2(ctx, "seller", "buyer");
  const order = await this.getOrder(ctx, _orderId);

```

```
  if (order.state == State.DELIVERED || order.state == State.SHIPPED || order.state == State.CANCELLED || order.state == State.PASSED) {
    throw new Error("The state of order " + _orderId + "  does not allow this action");
  }

  order.state = State.CANCELLED;

  await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
  console.info("Order " + _orderId + " has been cancelled.");
  console.info("============= END : cancelOrder ===========");
}

```

deliveryDatePassed

This method does not contain any affiliation check as we allow everyone in the channel to test if the delivery of the order is on time. The tricky part is in line 12 as the order is stored as a JSON on the ledger. To be able to compare the current date with the delivery date specified in the order we first have to parse the string to a Date. Otherwise the check does not work as intended because we would use a string compare instead of comparing two dates.

```
1  public async deliveryDatePassed(ctx: Context, _orderId: string): Promise<boolean> {
2    console.info("============= START : deliveryDatePassed ===========");
3    var passed = false;
4
5    const order = await this.getOrder(ctx, _orderId);
6
7    if (order.state >= State.DELIVERED) {
8      throw new Error("The state of order " + _orderId + "  does not allow this action");
9    }
10
11   var currentDate = new Date();
12   if (currentDate > new Date(order.latestDeliveryDate)) {
13     order.state = State.PASSED;
14     await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
15     passed = true;
16     console.info("Order " + _orderId + " has been cancelled due passed delivery date.");
17   }
18
19   console.info("============= END : deliveryDatePassed ===========");
20   return passed;
21 }
22
```

confirmOrder

Only a user of the organisation *buyer* is authorised to confirm an order. We throw an error if the state of the order is anything else than `CREATED`.

```
public async confirmOrder(ctx: Context, _orderId: string) {
  console.info("============= START : confirmOrder ===========");

  this.restrictedCall(ctx, "buyer");
  const order = await this.getOrder(ctx, _orderId);

  if (order.state != State.CREATED) {
    throw new Error("The state of order " + _orderId + "  does not allow this action");
  }

  order.state = State.CONFIRMED;

  await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
  console.info("Order " + _orderId + " has been confirmed.");
  console.info("============= END : confirmOrder ===========");
}

```

shipOrder

The ship order method is similar to the other methods previously described but restricts the calls to users from *seller* and only continues if the state of the order is `CONFIRMED`. In addition to the order id it also takes the tracking code of the shipment as another parameter. The value of this parameter gets added to the order and the order finally put onto the ledger again.

```
public async shipOrder(ctx: Context, _orderId: string, _trackingCode: string) {
  console.info("============= START : shipOrder ===========");

  this.restrictedCall(ctx, "seller");
  const order = await this.getOrder(ctx, _orderId);

  if (order.state != State.CONFIRMED) {
    throw new Error("The state of order " + _orderId + " does not allow this action");
  }

  order.state = State.SHIPPED;
  order.trackingCode = _trackingCode;

  await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
  console.info("Order " + _orderId + " has been shipped.");
  console.info("============= END : shipOrder ===========");
}
```

signArrival

Signing the arrival is the second method that is allowed to be called by users of two different organisations, *freight* and *buyer*. Depending on which organisation the user is part of either the *freightSigned* or *buyerSigned* variable gets set to true. After both organisations signed the arrival the state of the order transitions into its final state `DELIVERED`.

```
public async signArrival(ctx: Context, _orderId: string) {
  console.info("============= START : signArrival ===========");

  this.restrictedCall2(ctx, "freight", "buyer");
  const order = await this.getOrder(ctx, _orderId);

  if (order.state != State.SHIPPED) {
    throw new Error("The state of order " + _orderId + " does not allow this action");
  }

  if (ctx.clientIdentity.assertAttributeValue("hf.Affiliation", "buyer")) {
    order.buyerSigned = true;
    console.info("Order " + _orderId + " arrival has been signed by the buyer.");
  }

  if (ctx.clientIdentity.assertAttributeValue("hf.Affiliation", "freight")) {
    order.freightSigned = true;
    console.info("Order " + _orderId + " arrival has been signed by the freight company.");
  }

  if (order.buyerSigned && order.freightSigned) {
    order.state = State.DELIVERED;
    console.info("Order " + _orderId + " has been delivered.");
  }

  await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
  console.info("============= END : signArrival ===========");
}

```

5.2.4 Testing

After running the script `startFabric.sh` the distributed ledger is set up and the chaincode deployed. Our repository contains three slightly different Java applications that interact with the SC in the directory `fabric/trade-finance/application`.

To test different scenarios we will open three terminals, one in each of the seller, freight and buyer sub directories and run `mvn test`.

The seller test will invoke the sample ClientApp and perform the following:

- Enroll User1.seller and import it into the wallet (if it does not already exist there)
- Query all orders
- Add three new orders
- Output the added orders
- Wait until the order with id 2 is confirmed and ship it

The buyer test will invoke the sample ClientApp and perform the following:

- Enroll User1.buyer and import it into the wallet (if it does not already exist there)
- Query all orders
- Cancel the order with id 1
- Confirm the order with id 2
- Check if the delivery date of the order with id 3 passed
- Sign the arrival of the order with id 2

The freight test will invoke the sample ClientApp and perform the following:

- Enroll User1.freight and import it into the wallet (if it does not already exist there)
- Query all orders
- Sign the arrival of the order with id 2

The basic structure of the ClientApp is the same for all three organisations. The first step is to select the user. To be able to connect to the network we have to use the connection configuration file of the organisation. This file is automatically generated by our `startFabric.sh` script and can be found in a sub directory of `test-network`. After establishing a connection to the gateway we will select the channel in line 18 and the SC in line 19.

The code in the following lines interacts with the SC, either via evaluateTransaction or submitTransaction. The difference is that submitTransaction submits the returned proposal results from invoking the smart contract and waits until the transaction is committed. As a consequence the proposal results will be ordered, delivered to the peers for validating and later committed to the blockchain. The general consensus is to use submitTransaction for transactions that change the ledger (world state) and evaluateTransaction for transactions that only query the ledger. The parameters of either variant are of data type string. The first parameter is always the name of the called method. Relevant code of ClientApp.java:

```java
// A wallet stores a collection of identities
final Path walletPath = Paths.get(".", "wallet");
final Wallet wallet = Wallets.newFileSystemWallet(walletPath);
System.out.println("Read wallet info from: " + walletPath);

final String userName = "user1";

final Path connectionProfile = Paths.get("..", "..", "..", "..", "test-
    network", "organizations",
    "peerOrganizations", "seller.example.com", "connection-seller.yaml");

// Set connection options on the gateway builder
builder.identity(wallet, userName).networkConfig(connectionProfile).discovery
    (false);

// Connect to gateway using application specified parameters
try (Gateway gateway = builder.connect()) {

  // get the network and contract
  final Network network = gateway.getNetwork(channelName);
  final Contract contract = network.getContract(contractName);

  byte[] result;

  result = contract.evaluateTransaction("queryAllOrders");
  System.out.println("List of all orders:");
  System.out.println(new String(result));
  System.out.println("------------------------------------");
  contract.submitTransaction("createOrder", "1", "100", "2", "10", "2", "
      Karlsplatz 13, 1040 Wien",
      "2020-09-20");
  contract.submitTransaction("createOrder", "2", "123587", "5", "750", "4", "
      Ballhausplatz 2, 1010 Wien",
      "2020-12-01");
  contract.submitTransaction("createOrder", "3", "68754", "1", "1337", "2", "
      Michaelerkuppel, 1010 Wien",
      "2020-08-15");

  result = contract.evaluateTransaction("queryAllOrders");
  System.out.println("List of all orders:");
  System.out.println(new String(result));
  System.out.println("------------------------------------");
  System.out.println("Wait until order with id 2 is set to state CONFIRMED");
  result = contract.evaluateTransaction("queryOrder", "2");
  Order order = Order.deserialize(result);
  System.out.println(Order.deserialize(result));
```

```
  while (order.getState() != Order.State.CONFIRMED) {
    System.out.println("order 2 state is:" + order.getState());
    Thread.sleep(5000);
    result = contract.evaluateTransaction("queryOrder", "2");
    order = Order.deserialize(result);
  }

  contract.submitTransaction("shipOrder", "2", "1AXCAW311");
  System.out.println("shipped order 2");
  result = contract.evaluateTransaction("queryOrder", "2");
  System.out.println(Order.deserialize(result));
  System.out.println("------------------------------------");

}
```

5.3 Corda

In section 3.3 we have described the platform differences between Corda and the other introduced platforms. As a result of the UTXO transaction model and the missing currency token the SC design has to be adapted. The software used to develop and test the Corda SC is listed in Table 5.4.

Table 5.4: Software used for the Corda prototype

Software	**Version**	**Description**
Ubuntu	20.04 LTS	Operating system
Docker	19.03.12	OS-level virtualisation to deliver software in packages called containers
Docker Compose	1.26.2	Tool for defining and running multi-container Docker applications
Java	1.8.0_265	Java Development Kit
IntelliJ	2020.2	Integrated development environment (IDE)
Corda	4.5	DLT platform
JUnit	4.12	Java testing framework
Corda Node Explorer	0.1.1	Stand alone desktop app for connecting to a Corda node, examine transactions, run flows and view node and network properties.

5.3.1 Installation

The installation process is simpler than the one of the Hyperledger Fabric prototype. While we had to edit a lot of scripts in subsection 5.2.1, the Corda set-up does not involve any action like that.

- We will start again with a template project provided by the developers of the DLT platform.

```
git clone https://github.com/corda/cordapp-template-java.git
```

- The nodes configurations are found in the projects `build.grade` in the `deployNodes` and `prepareDockerNodes` tasks. Our CorDapp contains definitions of the notary node that is running the network map service and the three nodes involved in our smart contract, the seller, buyer and freight company. The name of the node contains the organistaions name, the location, the country and is internally parsed as a CordaX500Name[4] to create the certificates. It is also possible to restrict the access of the users connecting via Remote Procedure Call (RPC) to a certain set of flows. In our case we allow the user to execute all flows.

- One way to build the Corda project is via `./gradlew clean deployNodes`. This task will package the projects source files into a CorDapp JAR and create a new node in `build/nodes` with our CorDapp already installed. To start the nodes we execute the command `build/nodes/runnodes` from the projects root directory. This will start a terminal window for each node and allows the user to interact with the deployed CorDapp. An extract of the task definition:

```
task deployNodes(type: net.corda.plugins.Cordform, dependsOn: ['jar']) {
  ...
    node {
        name "O=Notary,L=London,C=GB"
        notary = [validating: false]
        p2pPort 10002
        rpcSettings {
            address("localhost:10003")
            adminAddress("localhost:10043")
        }
    }
    node {
        name "O=Seller,L=Berlin,C=DE"
        p2pPort 10005
        rpcSettings {
            address("localhost:10006")
            adminAddress("localhost:10046")
        }
        rpcUsers = [[user: "user1", "password": "test", "permissions": [
            "ALL"]]]
    }
    ...
}
```

- Another possibilty is the usage of Docker containers via `./gradlew clean prepareDockerNodes`. The gradle task is more or less the same, the only

---

[4] `https://api.corda.net/api/corda-os/4.5/html/api/javadoc/net/corda/core/identity/CordaX500Name.html`

difference is the type of it. Instead of `Cordform` we are using `Dockerform`. The result of the task is also similar to the non-Docker version but also contains the file `docker-compose.yml`. To start the nodes we simply open a terminal in the directory of the YML file and run the command `docker-compose up -d`.

```
task prepareDockerNodes(type: net.corda.plugins.Dockerform, dependsOn: [
    ’jar’]) {
  ...
    node {
        name "O=Notary,L=London,C=GB"
        notary = [validating: false]
        p2pPort 10002
        rpcSettings {
            address("localhost:10003")
            adminAddress("localhost:10043")
        }
        projectCordapp {
            deploy = false
        }
        cordapps.clear()
        sshdPort 2222
    }
    node {
        name "O=Seller,L=Berlin,C=DE"
        p2pPort 10005
        rpcSettings {
            address("localhost:10006")
            adminAddress("localhost:10046")
        }
        rpcUsers = [[user: "user1", "password": "test", "permissions": [
            "ALL"]]]
        sshdPort 2223
    }
    ...
    dockerImage = "corda/corda-zulu-java1.8-4.5"
}
```

- Each node directory has the following structure:

```
. nodeName
├── additional-node-infos
├── certificates
├── corda.jar              // The Corda node runtime
├── cordapps               // The node’s CorDapps
│   ├── config
│   ├── accounts-contracts-1.0.jar  // Corda accounts contracts
│   ├── accounts-workflows-1.0.jar  // Corda accounts workflows
│   ├── ci-workflows-1.0.jar
│   ├── contracts-0.1.jar   // Our contract
│   └── workflows-0.1.jar   // Our workflows
├── djvm
├── drivers
├── logs
```

```
├── network-parameters
├── node.conf               // The node's configuration file
├── nodeInfo-<HASH>         // The hash will be different each time you
    generate a node
├── persistence.mv.db       // The node's database
└── persistence.trace.db    // The node's database
```

We first describe the basic structure of our SC and how the parts interact with each other. Then we examine the functions to implement the work-flow designed in chapter 4 and finally review the testing. While the other prototypes involve a mix of different programming languages every part of our CorDapp is written in Java.

5.3.2 Relevant files

```
.
├── build.gradle
├── contracts
│   └── src
│       ├── main
│       │   └── java
│       │       └── com
│       │           └── template
│       │               ├── contracts
│       │               │   └── TradeFinanceContract.java
│       │               └── states
│       │                   └── OrderState.java
└── workflows
    └── src
        ├── main
        │   └── java
        │       └── com
        │           └── template
        │               ├── flows
        │               │   ├── CancelOrder.java
        │               │   ├── CancelOrderResponder.java
        │               │   ├── CheckDeliveryDate.java
        │               │   ├── CheckDeliveryDateResponder.java
        │               │   ├── ConfirmOrder.java
        │               │   ├── ConfirmOrderResponder.java
        │               │   ├── CreateOrder.java
        │               │   ├── CreateOrderResponder.java
        │               │   ├── ShipOrder.java
        │               │   ├── ShipOrderResponder.java
        │               │   ├── SignArrival.java
        │               │   └── SignArrivalResponder.java
        │               └── utils
        │                   └── DataUtils.java
        └── test
            └── java
                └── com
```

```
                    └── template
                        └── FlowTests.java
```

5.3.3 OrderState

As already mentioned Corda uses the UTXO transaction model. This means transactions either have to consume or produce (or both) a state. In our case the state contains all the order information. The `OrderState` class itself implements the class `LinearState`. A LinearState is used to store states that "evolve by superseding itself" [5]. This means when the state is updated the original state should be included as the transaction input and the updated one as transaction output. The state included as transaction input will now be marked as `CONSUMED` while the output is `UNCONSUMED`. Therefore a vault query to get the `UNCONSUMED` OrderState with a certain ID will always result in the most recent data.

The method `getParticipants()` returns a list of involved parties. This list is later used as the list of required signers in a Command. The involved parties need to verify and sign the transaction and change depending on the state of the order. If an order is not shipped yet the freight company does not need to sign the transactions as it is not involved in the order process for now. The signers also store a copy of the state in their vault. If we would exclude the buyer from this list he would not be able to view any of the order data.

5.3.4 TradeFinanceContract

A transaction is valid if the `verify()` function of the contract does not throw an exception. In simple terms the verify function is a checklist and validates if the required conditions are met. The input parameter of the verify function is the transaction, which itself contains the `Command`. A command indicates the transaction's intent. If we want to create a new order we will create a transaction with a `Create` command. The following code snippet shows the `Commands` class:

```java
public abstract static class Commands implements CommandData {
  private Party initiator;

  public Commands(Party initiator) {
    this.initiator = initiator;
  }

  public Party getInitiator() {
    return initiator;
  }

```

---

[5] https://api.corda.net/api/corda-os/4.5/html/api/kotlin/corda/net.corda.core.contracts/-linear-state/index.html

```
  public static class Create extends Commands {
    public Create(Party initiator) {
      super(initiator);
    }
  }

  public static class Cancel extends Commands {
    public Cancel(Party initiator) {
      super(initiator);
    }
  }
...
```

In the verify method we get the data of the command and use our custom logic to decide if the transaction is valid or invalid. Based on the given command the logic differs. The following code snippet shows the validation logic of the `Create` and `Cancel` commands. Using the Corda DSL function `requireThat` allows for easier readable conditions. Our `Create` command is only valid if there is no input state to consume, the flow is initiated by the seller and if the price of the order is greater or equal to the shipping costs.

The `Cancel` command on the other hand requires exactly one input (because of the LinearState) and checks if the current state of the order is either `CREATED` or `CONFIRMED`. A cancel transaction is also only valid if either the seller or buyer started the flow.

```
public void verify(LedgerTransaction tx) {
...
  if (command.getValue() instanceof Commands.Create) {
    requireThat(require -> {
      require.using("No inputs should be consumed when adding a new order.",
          tx.getInputStates().size() == 0);
      require.using("Only the seller is allowed to start this flow.", command
          .getValue().getInitiator().getOwningKey().equals(output.getSeller()
          .getOwningKey()));
      require.using("The price must be greater or equal to the shipping costs
          .", output.getPrice().compareTo(output.getShippingCosts()) >= 0);
      return null;
    });
  } else if (command.getValue() instanceof Commands.Cancel) {
    requireThat(require -> {
      require.using("Exactly one input should be consumed when cancelling an
          order.", tx.getInputStates().size() == 1);
      require.using("Function cannot be called at this state: " + input.
          getOrderState(), Stream.of(input.getOrderState()).anyMatch(Arrays.
          asList(OrderState.State.CREATED, OrderState.State.CONFIRMED)::
          contains));
      require.using("Only the the seller or the buyer are allowed to start
          this flow.", Arrays.asList(output.getSeller().getOwningKey(),
          output.getBuyer().getOwningKey()).contains(command.getValue().
          getInitiator().getOwningKey()));
      return null;
    });
  }
```

```
...
}
```

5.3.5 Flows

The purpose of a flow is to create a transaction which either creates or updates an order. A flow session is a channel across the Corda network involving the signers of the Command. Each flow contains a constructor class which is used to specify and parse the flow input parameters. When a flow is started its `call()` method is executed. The differences between the different flows are usually in step three and step five and will be clarified on their own. All our flows use the same structure to build a transaction.

1. Check if an order with the given ID already exists (if we want to create a new oder) or if an order with the given ID does not exist (all other cases).

   ```
   QueryCriteria.LinearStateQueryCriteria queryCriteria = new QueryCriteria
       .LinearStateQueryCriteria().withExternalId(Collections.singletonList
       (this.orderId));
   List<StateAndRef<OrderState>> results = getServiceHub().getVaultService
       ().queryBy(OrderState.class, queryCriteria).getStates();
   if (results.size() != 0) {
     throw new IllegalArgumentException("An order with ID " + this.orderId
         + " already exists.");
   }
   ```

2. Get a reference to the notary service on our network and our key pair. In this case we know our test network only contains one notary and therefore we have no problem by choosing the first entry. For real world cases it is better to either select a notary randomly or one specific notary by name oder key.

   ```
   final Party notary = getServiceHub().getNetworkMapCache().
       getNotaryIdentities().get(0);
   ```

3. Compose the State that carries the order data

4. Create a new TransactionBuilder object.

   ```
   final TransactionBuilder builder = new TransactionBuilder(notary);
   ```

5. Add the order as an output state, as well as a command to the transaction builder.

6. Verify and sign it with our KeyPair.

   ```
   builder.verify(getServiceHub());
   final SignedTransaction ptx = getServiceHub().signInitialTransaction(
       builder);
   ```

7. Collect the other party's signature using the SignTransactionFlow.

```
List<Party> otherParties = outputOrderState.getParticipants().stream().
    map(el -> (Party) el).collect(Collectors.toList());
otherParties.remove(getOurIdentity());
List<FlowSession> sessions = otherParties.stream().map(this::
    initiateFlow).collect(Collectors.toList());

SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx, sessions)
    );
```

8. Assuming no exceptions, we can now finalise the transaction

```
subFlow(new FinalityFlow(stx, sessions));
```

createOrder

Because of the usage of the data type `Amount<Currency>` for the variables price and shippingCosts we are able to pass strings like "€10" and "10 EUR" to the flow. The date format of the latestDeliveryDate is 'YYYY-MM-DD', all the other parameters are standard strings, double or integers.

3. Compose the State that carries the order data.

```
Party buyerParty = getServiceHub().getIdentityService().partiesFromName(
    this.buyer, true).stream().findFirst().get();
final OrderState output = new OrderState(this.seller, buyerParty, this.
    orderId, this.productId, this.quantity, this.price, this.
    shippingCosts, this.shippingAddress, this.latestDeliveryDate);
```

5. Add the order as an output state, as well as a command to the transaction builder.

```
builder.addOutputState(output, TradeFinanceContract.ID);
builder.addCommand(new TradeFinanceContract.Commands.Create(
    getOurIdentity()), output.getParticipants().stream().map(
    AbstractParty::getOwningKey).collect(Collectors.toList()));
```

cancelOrder

This flow only has one parameter, the orderId as a string.

3. Compose the State that carries the order data.

```
OrderState outputOrderState = inputOrderState.copy();
outputOrderState.setOrderState(OrderState.State.CANCELLED);
```

5. Add the order as an output state, as well as a command to the transaction builder.

```
builder.addInputState(inputOrderStateAndRef);
builder.addOutputState(outputOrderState);
builder.addCommand(new TradeFinanceContract.Commands.Cancel(
    getOurIdentity()), outputOrderState.getParticipants().stream().map(
    AbstractParty::getOwningKey).collect(Collectors.toList()));
```

deliveryDatePassed

This flow only has one parameter, the orderId as a string.

3. Compose the State that carries the order data.

```
OrderState outputOrderState = inputOrderState.copy();
outputOrderState.setOrderState(OrderState.State.PASSED);
```

5. Add the order as an output state, as well as a command to the transaction builder.

```
builder.addInputState(inputOrderStateAndRef);
builder.addOutputState(outputOrderState);
builder.addCommand(new TradeFinanceContract.Commands.CheckDate(
    getOurIdentity()), outputOrderState.getParticipants().stream().map(
    AbstractParty::getOwningKey).collect(Collectors.toList()));
```

confirmOrder

This flow only has one parameter, the orderId as a string.

3. Compose the State that carries the order data.

```
OrderState outputOrderState = inputOrderState.copy();
outputOrderState.setOrderState(OrderState.State.CONFIRMED);
```

5. Add the order as an output state, as well as a command to the transaction builder.

```
builder.addInputState(inputOrderStateAndRef);
builder.addOutputState(outputOrderState);
builder.addCommand(new TradeFinanceContract.Commands.Confirm(
    getOurIdentity()), outputOrderState.getParticipants().stream().map(
    AbstractParty::getOwningKey).collect(Collectors.toList()));
```

shipOrder

This flow has three parameters, the orderId, the trackingCode and name of the freight company. All of them are strings.

3. Compose the State that carries the order data.

```
final Party freightParty = getServiceHub().getIdentityService().
    partiesFromName(this.freightCompany, true).stream().findFirst().get
    ();
OrderState outputOrderState = inputOrderState.copy();
outputOrderState.setOrderState(OrderState.State.SHIPPED);
outputOrderState.setFreightCompany(freightParty);
outputOrderState.setTrackingCode(this.trackingCode);
```

5. Add the order as an output state, as well as a command to the transaction builder.

```
builder.addInputState(inputOrderStateAndRef);
builder.addOutputState(outputOrderState);
builder.addCommand(new TradeFinanceContract.Commands.Ship(getOurIdentity
    ()), outputOrderState.getParticipants().stream().map(AbstractParty::
    getOwningKey).collect(Collectors.toList()));
```

signArrival

This flow only has one parameter, the orderId as a string.

3. Compose the State that carries the order data.

```
OrderState outputOrderState = inputOrderState.copy();
if (getOurIdentity().getOwningKey().equals(outputOrderState.getBuyer().
    getOwningKey())) {
  outputOrderState.setBuyerSigned(true);
  signer = outputOrderState.getBuyer().getName().toString();
} else if (getOurIdentity().getOwningKey().equals(outputOrderState.
    getFreightCompany().getOwningKey())) {
  outputOrderState.setFreightSigned(true);
  signer = outputOrderState.getFreightCompany().getName().toString();
}

if (outputOrderState.isBuyerSigned() && outputOrderState.isFreightSigned
    ()) {
  outputOrderState.setOrderState(OrderState.State.DELIVERED);
}
```

5. Add the order as an output state, as well as a command to the transaction builder.

```
builder.addInputState(inputOrderStateAndRef);
builder.addOutputState(outputOrderState);
builder.addCommand(new TradeFinanceContract.Commands.Sign(getOurIdentity
    ()), outputOrderState.getParticipants().stream().map(AbstractParty::
    getOwningKey).collect(Collectors.toList()));
```

5.3.6 Testing

Manual

To manually test a flow we have to start the nodes as described in subsection 5.3.1.

- To create a new order we switch into the seller terminal and execute the following command:

  ```
  flow start CreateOrder buyer: Buyer, orderId: 1, productId: 100,
      quantity: 2, price: "10 EUR", shippingCosts: "2 EUR, shippingAddress
      : "Karlsplatz 13, 1040 Wien", latestDeliveryDate: "2020-09-30"
  ```

- Now we could switch to the buyer terminal and execute:

  ```
  flow start ConfirmOrder orderId: 1
  ```

- Once the order is confirmed the seller is able to ship it using:

  ```
  flow start ShipOrder orderId: 1, trackingCode: XAFDWEQ, freightCompany:
      'Freight Company'
  ```

- To finish the trade the buyer and freight company both have to sign the arrival of the order:

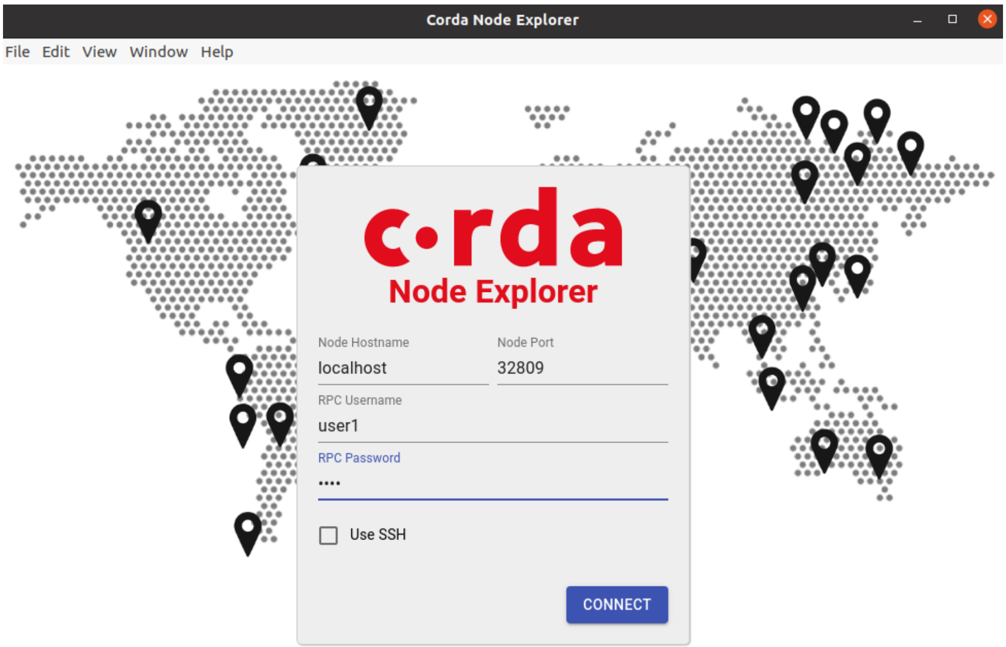
  ```
  flow start SignArrival orderId: 1
  ```

Once the nodes are running we are also able to connect via RPC. for example with the Corda Node Explorer[6]. The Node Explorer is a user interface that allows to inspect the nodes vault, to look-up the transaction history and to start flows.

- We have started the nodes as Docker containers. To be able to connect to the seller we put as hostname 'localhost' and as port '32809'. The command 'docker ps' shows us the ports associated with each node.

---

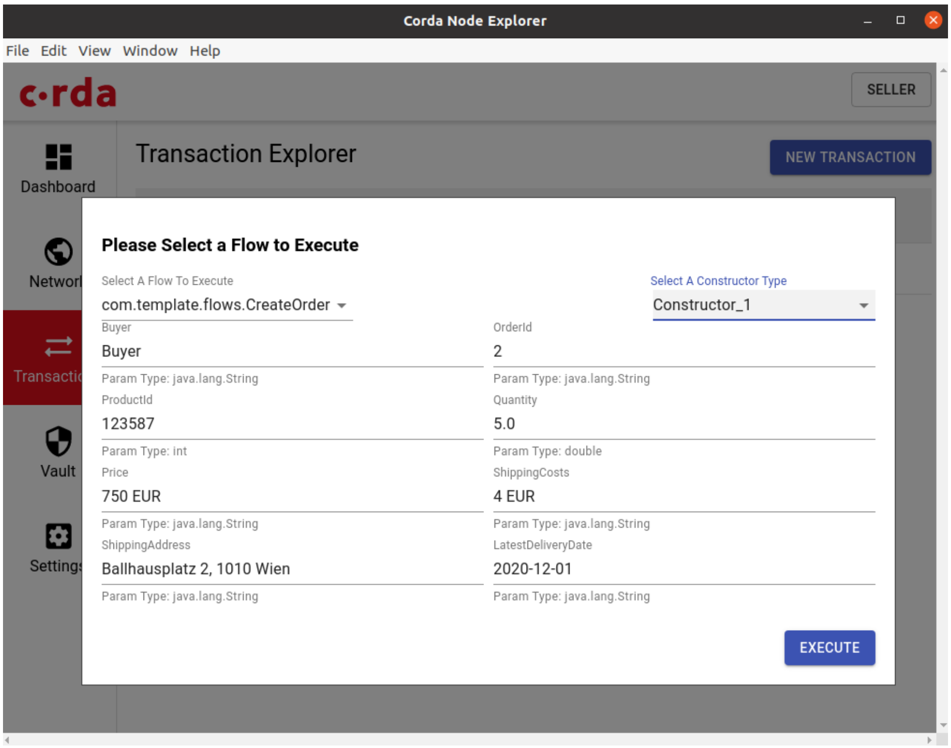
[6] `https://github.com/corda/node-explorer`



Node Hostname: localhost
Node Port: 32809
RPC Username: user1
RPC Password: ••••
☐ Use SSH

Figure 5.1: Corda Node Explorer: Login

- To create a new order we first have to select ’Transactions’ on the left side tab and then click the button ’New Transaction’. We are able to select the ’CreateOrder’ flow and enter some parameters.



Transaction Explorer

Please Select a Flow to Execute

Select A Flow To Execute: com.template.flows.CreateOrder

Select A Constructor Type: Constructor_1

Buyer: Buyer
Param Type: java.lang.String

OrderId: 2
Param Type: java.lang.String

ProductId: 123587
Param Type: int

Quantity: 5.0
Param Type: double

Price: 750 EUR
Param Type: java.lang.String

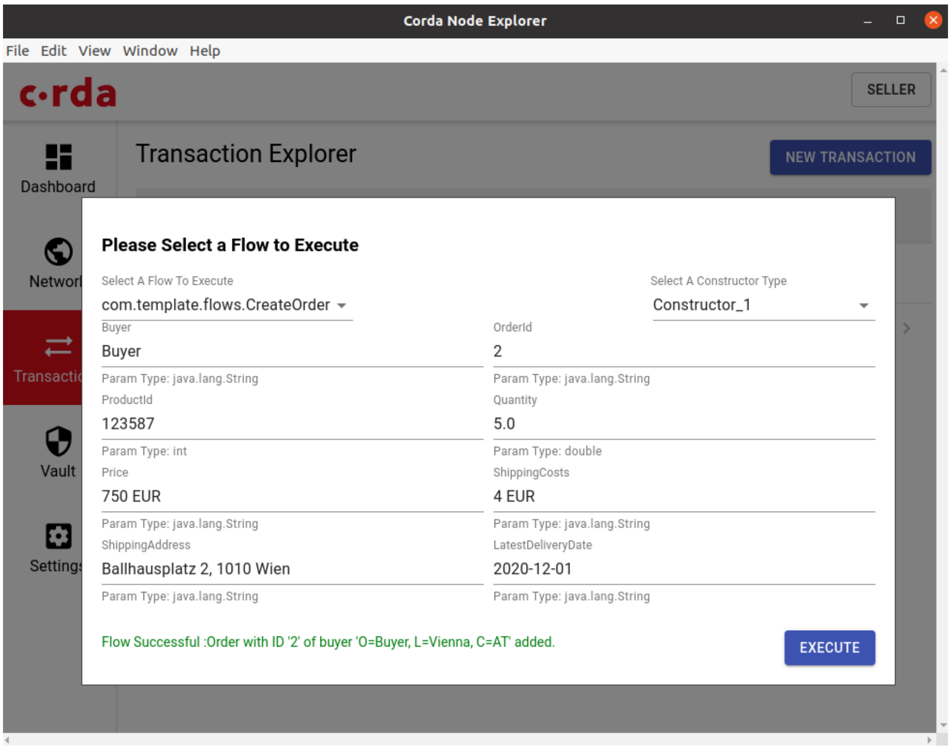
ShippingCosts: 4 EUR
Param Type: java.lang.String

ShippingAddress: Ballhausplatz 2, 1010 Wien
Param Type: java.lang.String

LatestDeliveryDate: 2020-12-01
Param Type: java.lang.String

Figure 5.2: Corda Node Explorer: Create a new order

- Once the transaction was successful we see the return value of the flow in the lower left corner.



Transaction Explorer

Please Select a Flow to Execute

Select A Flow To Execute: com.template.flows.CreateOrder

Select A Constructor Type: Constructor_1

Buyer: Buyer
Param Type: java.lang.String

OrderId: 2
Param Type: java.lang.String

ProductId: 123587
Param Type: int

Quantity: 5.0
Param Type: double

Price: 750 EUR
Param Type: java.lang.String

ShippingCosts: 4 EUR
Param Type: java.lang.String

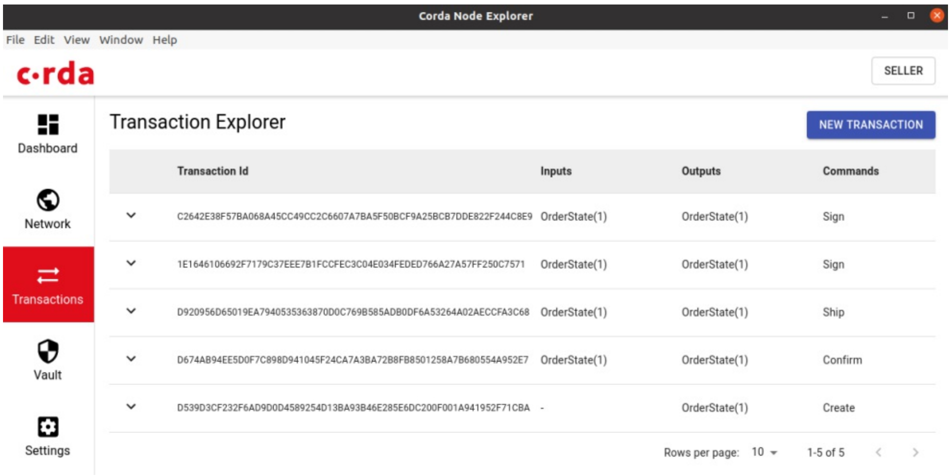
ShippingAddress: Ballhausplatz 2, 1010 Wien
Param Type: java.lang.String

LatestDeliveryDate: 2020-12-01
Param Type: java.lang.String

Flow Successful :Order with ID '2' of buyer 'O=Buyer, L=Vienna, C=AT' added.

Figure 5.3: Corda Node Explorer: Successfully created a new order

- The transactions tab gives an overview of passed transactions but also offers the ability to get a detailed look at the transaction data.



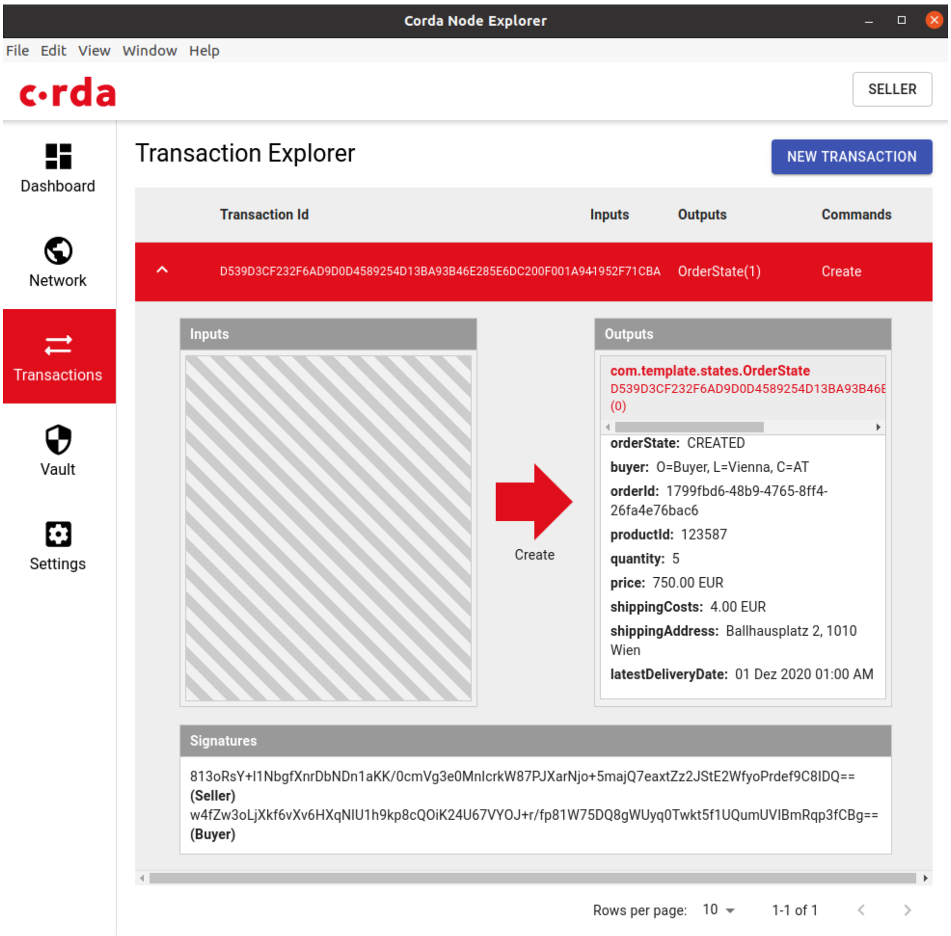
Transaction Explorer

Transaction Id	Inputs	Outputs	Commands
C2642E38F57BA068A45CC49CC2C6607A7BA5F50BCF9A25BCB7DDE822F244C8E9	OrderState(1)	OrderState(1)	Sign
1E1646106692F7179C37EEE7B1FCCFEC3C04E034FEDED766A27A57FF250C7571	OrderState(1)	OrderState(1)	Sign
D920956D65019EA7940535363870D0C769B585ADB0DF6A53264A02AECCFA3C68	OrderState(1)	OrderState(1)	Ship
D674AB94EE5D0F7C898D941045F24CA7A3BA72B8FB8501258A7B680554A952E7	OrderState(1)	OrderState(1)	Confirm
D539D3CF232F6AD9D0D4589254D13BA93B46E285E6DC200F001A941952F71CBA	-	OrderState(1)	Create

Rows per page: 10 1-5 of 5

Figure 5.4: Corda Node Explorer: Overview of valid transactions

- In this case we see the 'CreateOrder' transaction does not have any input state but produces an output state.



Transaction Explorer

Transaction Id	Inputs	Outputs	Commands
D539D3CF232F6AD9D0D4589254D13BA93B46E285E6DC200F001A941952F71CBA		OrderState(1)	Create

com.template.states.OrderState
D539D3CF232F6AD9D0D4589254D13BA93B46E
(0)

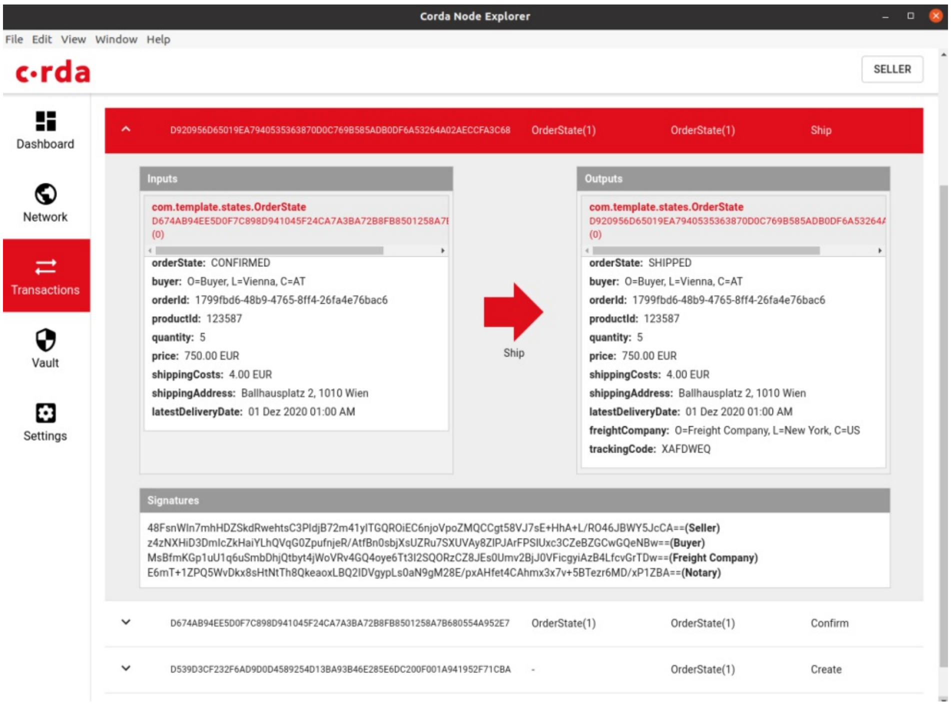
orderState: CREATED
buyer: O=Buyer, L=Vienna, C=AT
orderId: 1799fbd6-48b9-4765-8ff4-26fa4e76bac6
productId: 123587
quantity: 5
price: 750.00 EUR
shippingCosts: 4.00 EUR
shippingAddress: Ballhausplatz 2, 1010 Wien
latestDeliveryDate: 01 Dez 2020 01:00 AM

Signatures

813oRsY+l1NbgfXnrDbNDn1aKK/0cmVg3e0MnlcrkW87PJXarNjo+5majQ7eaxtZz2JStE2WfyoPrdef9C8IDQ==
(Seller)
w4fZw3oLjXkf6vXv6HXqNIU1h9kp8cQOiK24U67VYOJ+r/fp81W75DQ8gWUyq0Twkt5f1UQumUVIBmRqp3fCBg==
(Buyer)

Figure 5.5: Corda Node Explorer: Create order transaction

- Once the seller ships the order the output state contains two more variables, the tracking code and the CordaX500Name of the freight company. Another change is the freight company is now included in the list of neccessary signers to confirm the transaction.



D920956D65019EA7940535363870D0C769B585ADB0DF6A53264A02AECCFA3C68 OrderState(1) OrderState(1) Ship

Inputs

com.template.states.OrderState
D674AB94EE5D0F7C898D941045F24CA7A3BA72B8FB8501258A7E
(0)

orderState: CONFIRMED
buyer: O=Buyer, L=Vienna, C=AT
orderId: 1799fbd6-48b9-4765-8ff4-26fa4e76bac6
productId: 123587
quantity: 5
price: 750.00 EUR
shippingCosts: 4.00 EUR
shippingAddress: Ballhausplatz 2, 1010 Wien
latestDeliveryDate: 01 Dez 2020 01:00 AM

Outputs

com.template.states.OrderState
D920956D65019EA7940535363870D0C769B585ADB0DF6A53264A
(0)

orderState: SHIPPED
buyer: O=Buyer, L=Vienna, C=AT
orderId: 1799fbd6-48b9-4765-8ff4-26fa4e76bac6
productId: 123587
quantity: 5
price: 750.00 EUR
shippingCosts: 4.00 EUR
shippingAddress: Ballhausplatz 2, 1010 Wien
latestDeliveryDate: 01 Dez 2020 01:00 AM
freightCompany: O=Freight Company, L=New York, C=US
trackingCode: XAFDWEQ

Signatures

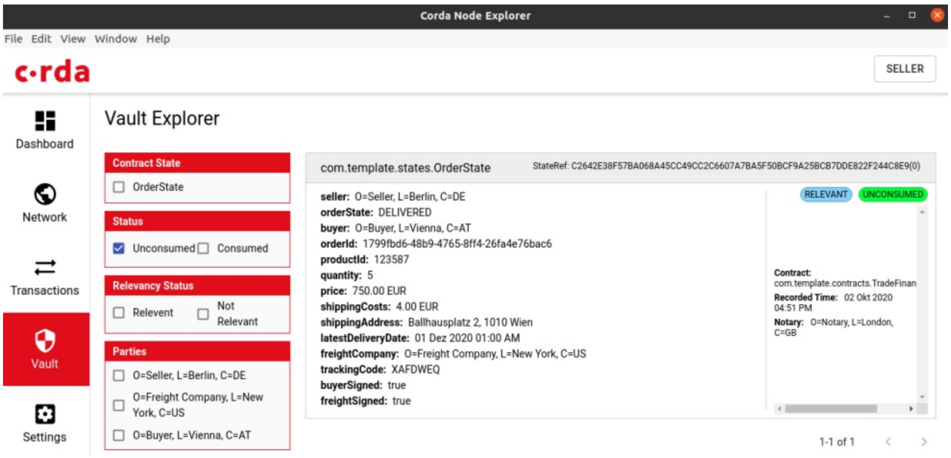
48FsnWIn7mhHDZSkdRwehtsC3PldjB72m41yITGQROiEC6njoVpoZMQCCgt58VJ7sE+HhA+L/RO46JBWY5JcCA==**(Seller)**
z4zNXHiD3DmIcZkHaiYLhQVqG0ZpufnjeR/AtfBn0sbjXsUZRu7SXUVAy8ZIPJArFPSIUxc3CZeBZGCwGQeNBw==**(Buyer)**
MsBfmKGp1uU1q6uSmbDhjQtbyt4jWoVRv4GQ4oye6Tt3I2SQORzCZ8JEs0Umv2BjJ0VFicgyiAzB4LfcvGrTDw==**(Freight Company)**
E6mT+1ZPQ5WvDkx8sHtNtTh8QkeaoxLBQ2IDVgypLs0aN9gM28E/pxAHfet4CAhmx3x7v+5BTezr6MD/xP1ZBA==**(Notary)**

D674AB94EE5D0F7C898D941045F24CA7A3BA72B8FB8501258A7B680554A952E7 OrderState(1) OrderState(1) Confirm

D539D3CF232F6AD9D0D4589254D13BA93B46E285E6DC200F001A941952F71CBA - OrderState(1) Create

Figure 5.6: Corda Node Explorer: Ship order transaction

- The Node Explorer also provides the user a comfortable way to check the content of the nodes vault. In our case we are able to filter for `UNCONSOMED` entries because of the usage of `LinearState` in our `OrderState` class. This way we always get the most recent data of an order.



Vault Explorer

Contract State
☐ OrderState

Status
☑ Unconsumed ☐ Consumed

Relevancy Status
☐ Relevent ☐ Not Relevant

Parties
☐ O=Seller, L=Berlin, C=DE
☐ O=Freight Company, L=New York, C=US
☐ O=Buyer, L=Vienna, C=AT

com.template.states.OrderState StateRef: C2642E38F57BA068A45CC49CC2C6607A7BA5F50BCF9A25BCB7DDE822F244C8E9(0)

seller: O=Seller, L=Berlin, C=DE
orderState: DELIVERED
buyer: O=Buyer, L=Vienna, C=AT
orderId: 1799fbd6-48b9-4765-8ff4-26fa4e76bac6
productId: 123587
quantity: 5
price: 750.00 EUR
shippingCosts: 4.00 EUR
shippingAddress: Ballhausplatz 2, 1010 Wien
latestDeliveryDate: 01 Dez 2020 01:00 AM
freightCompany: O=Freight Company, L=New York, C=US
trackingCode: XAFDWEQ
buyerSigned: true
freightSigned: true

Contract: com.template.contracts.TradeFinan
Recorded Time: 02 Okt 2020 04:51 PM
Notary: O=Notary, L=London, C=GB

1-1 of 1

Figure 5.7: Corda Node Explorer: Vault content

Automated

Corda also supports JUnit tests. We are able to mock a network as shown in the following code snippet. First we have to add the package of the SC and flows to be able to create the mocked network. The tests do not rely on the `build.gradle` file. Therefore we need to define the details of the nodes of the network again.

```
@Before
public void setup() {
  network = new MockNetwork(new MockNetworkParameters().
      withCordappsForAllNodes(ImmutableList.of(
      TestCordapp.findCordapp("com.template.contracts"),
      TestCordapp.findCordapp("com.template.flows"))));
  sellerNode = network.createPartyNode(new CordaX500Name("Seller", "Berlin",
      "DE"));
  buyerNode = network.createPartyNode(new CordaX500Name("Buyer", "Vienna", "
      AT"));
  freightNode = network.createPartyNode(new CordaX500Name("Freight Company",
      "New York", "US"));
  // For real nodes this happens automatically, but we have to manually
      register the flow for tests.
  for (StartedMockNode node : ImmutableList.of(sellerNode, buyerNode,
      freightNode)) {
    node.registerInitiatedFlow(CancelOrderResponder.class);
    node.registerInitiatedFlow(CheckDeliveryDateResponder.class);
    node.registerInitiatedFlow(ConfirmOrderResponder.class);
    node.registerInitiatedFlow(CreateOrderResponder.class);
    node.registerInitiatedFlow(ShipOrderResponder.class);
    node.registerInitiatedFlow(SignArrivalResponder.class);
  }
  network.runNetwork();
}
```

To create a flow we first have to call its constructor. A flow gets started by a node after it got created. All our flows return a string in case of a successful execution. We use the existence of such a string to check if the flow executed correct. Testing the whole standard process of our SC could be implemented like this:

```
@Test
public void signArrivalTest() throws ExecutionException, InterruptedException
     {
  FlowLogic<String> flow = new CreateOrder("Buyer", "2", 123587, 5.0, "750
      EUR", "4 EUR", "Ballhausplatz 2, 1010 Wien", "2020-12-01");
  CordaFuture<String> future = sellerNode.startFlow(flow);
  network.runNetwork();
  assert future.get().contains("Order with ID ’2’ of buyer ’" + buyerNode.
      getInfo().getLegalIdentities().get(0).getName() + "’ added.");

  flow = new ConfirmOrder("2");
  future = buyerNode.startFlow(flow);
  network.runNetwork();
```

```
  assert future.get().contains("Confirm order flow for order with ID ’2’ of
      buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0).getName() +
      "’ executed.");

  flow = new ShipOrder("2", "Freight Company", "XAFDWEQ");
  future = sellerNode.startFlow(flow);
  network.runNetwork();
  assert future.get().contains("Ship order flow for order with ID ’2’ of
      buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0).getName() +
      "’ executed.");

  flow = new SignArrival("2");
  future = buyerNode.startFlow(flow);
  network.runNetwork();
  assert future.get().contains("The arrival of the order with ID ’2’ has been
       signed by ’" + buyerNode.getInfo().getLegalIdentities().get(0).getName
      () + "’");

  flow = new SignArrival("2");
  future = freightNode.startFlow(flow);
  network.runNetwork();
  assert future.get().contains("The arrival of the order with ID ’2’ has been
       signed by ’" + freightNode.getInfo().getLegalIdentities().get(0).
      getName() + "’");

  // We check the recorded order in all three vaults.
  for (StartedMockNode node : ImmutableList.of(sellerNode, buyerNode,
      freightNode)) {
    node.transaction(() -> {
      List<StateAndRef<OrderState>> orders = node.getServices().
          getVaultService().queryBy(OrderState.class).getStates();
      assertEquals(1, orders.size());
      OrderState recordedState = orders.get(0).getState().getData();
      assertEquals(recordedState.getOrderState(), OrderState.State.DELIVERED)
          ;
      return null;
    });
  }
}
```

CHAPTER 6

Comparison

In this chapter we will evaluate the DLT platforms and prototype development based on the criteria established in subsection 1.5.2. A platform will get one to three points for each criterion while three points are the best possible result. In the end we will sum up the distributed points and rank the platforms from the highest to the lowest number.

6.1 Platform

The platform evaluation section will for the most part be based on theoretical considerations given the platform architectures. Detailed performance measurements and experiments are beyond the scope of this work.

6.1.1 Performance

How long does it take to finalise a transaction? Scalability?

Based on the architectural differences between private-permissioned (Hyperledger Fabric, Corda) and public-permissionless (Ethereum) DLT platforms a difference in performance and scalability is evident. "Assessment shows that Hyper-ledger Fabric achieves higher throughput and lower latency compared to Ethereum when the workloads are varied upto 10,000 transactions. Also, differences between these two platforms in respect to execution time and average latency become more significant as the number of transactions grow." [40]

For Corda we were not able to find academic literature on performance measurements. In theory the performance of Corda Enterprise should be faster than Ethereum and similar to Hyperledger Fabric. This assumption is supported by a[1] number[2] of blog posts of Corda developers. The open-source Corda version does not contain all performance enhancements but is theoretically still faster than Ethereum because of the limited number of transaction validators needed. Performance should not be the highest priority when developing a SC on the Ethereum platform.

Table 6.1: Performance rating

Platform	Points
Hyperledger Fabric	3
Corda	2
Ethereum	1

6.1.2 Confidentiality

Prevention of unauthorised information access?

All information stored on the Ethereum blockchain is public. This includes all order data our SC processed. To prevent unauthorised access the developer would have to implement a solution on their own. One possible solution would be to hash confidential data. This would need an off-chain database to be able to compare the hashes with the hidden values.

Hyperledger Fabric and Corda both use private channels between the involved parties. This design concept eliminates possible problems with confidentiality of stored information and does not need any extra work of the developers.

Table 6.2: Confidentiality rating

Platform	Points
Corda	3
Hyperledger Fabric	3
Ethereum	1

6.1.3 Costs

What are the costs for participating in the network?

Deploying a SC or executing a transaction on Hyperledger Fabric and Corda does not cost extra. On Ethereum transaction costs are based on their complexity and paid with

---

[1] `https://medium.com/corda/transactions-per-second-tps-de3fb55d60e3`

[2] `https://www.corda.net/blog/performance-improvements-in-corda-enterprise-4-4-and-4-5/`

the platform token Ether (ETH). A typical contract creation transaction includes the base costs for any transaction (C_{tx}), the costs for allocating a new address (C_{addr}), the contract payload ($C_{payload}$, contract bytecode size multiplied by gas per byte) and any extra gas used up by the opcodes in the function definition ($C_{fn_{def}}$). The formula is shown in Equation 6.2. All costs follow a fixed pricing table as specified in Wood [52]. [55]

$$C_{payload} = \text{payload (in bytes)} \cdot C_{gas/byte} \tag{6.1}$$

$$C_{create} = C_{tx} + C_{addr} + C_{payload} + C_{fn_{def}} \tag{6.2}$$

The output of the truffle migrate command in subsection 5.1.3 estimates the expected costs. Depending on how much gas we are willing to pay the faster the deploy transaction will get confirmed by the miners. Truffle uses 20 gwei as default value, one gwei is 10^{-9} Ether. According to ETH Gas Station[3] the median confirmation time when using 20 gwei is 0.6 minutes when deployed at 2020-10-28 17:42. The exchange rate of 1 ETH is 327.74 EUR at the time of writing as per Coinmarketcap[4]. This results in costs of about 15 EUR to publish our smart contract for worldwide usage.

In addition each transaction (confirming the order, shipping, et cetera) will cost some amount of gas based on the default transaction costs (C_{tx}), the costs of the data payload ($C_{payload}$) and the gas consumed by the opcodes during the function execution ($C_{fn_{exec}}$). Each method call costs about 0.50 EUR. The foundation of our cost calculation is as also shown in Table 5.2.

$$C_{exec} = C_{tx} + C_{payload} + C_{fn_{exec}} \tag{6.3}$$

A convenient approach to set-up and run the platforms is the usage of a cloud environment. IBM and Amazon both offer flexible pricing to deploy Hyperledger Fabric 1.4 as a Software as a Service (SaaS). Hyperledger Fabric 1.4 is the first long time support version and was released in January 2019. Unfortunately our prototype builds on the most recent version, 2.2. The IBM Cloud deployment option costs \$0.29 per hour[28], Amazon Managed Blockchain costs \$0.676 per hour [2].

Another option could be the Amazon Web Services (AWS) Marketplace. Various sellers offer an AWS backed solution for Ethereum (\$0.17/hr)[5], Hyperledger Fabric (\$0.063/hr)[6]

---

[3] `https://ethgasstation.info/`

[4] `https://coinmarketcap.com/currencies/ethereum/`

[5] `https://aws.amazon.com/marketplace/pp/B07KWH13Y8`

[6] `https://aws.amazon.com/marketplace/pp/B07S8CBV65`

and Corda ($0.096/hr)[7]. A Corda Enterprise license has to be acquired to be able to use the marketplace offer. The pricing information is not published. In our opinion Hyperledger Fabric is the most cost efficient option. The ongoing costs of Ethereum transactions coupled with the volatile exchange rate (97€-410€ within the last year[8]) are problematic for cost estimations.

Table 6.5 lists the costs of a number of steps involved in the traditional, bank-based L/C. The least expensive bank charges at least 675 EUR per L/Cand does not include any amendments or additional expenses as shown in Table 6.4. If we take the costs of a professionally managed Hyperledger Fabric environment by IBM or Amazon and multiple the hourly rate with 8760 to get the operational costs of one common Gregorian calendar year we get fixed costs between 2000 EUR and 5000 EUR, as shown in Table 6.3.

Such an environment lets us perform an unlimited number of trades as long as our partner also participates within the Hyperledger Fabric network. Therefore the gain from the investment would be within a few trades. We think the labour costs of administrating and developing the DLT platform are similar to the costs of a person handling the L/C process with the banks. The hourly costs of the IT employee is probably higher but the number of hours in total is lower as the steps of each single L/C application take much longer.

Table 6.3: Hyperledger Fabric yearly costs

Type	**USD / hour**	**USD / year**
IBM Cloud [28]	0.29	2540.4
Amazon Managed Blockchain [2]	0.676	5921.76

Table 6.4: Additional expenses of traditional Letter of Credit [41]

Type	**Costs**
Letter, fax, e-mail	10 EUR
SWIFT	15 EUR
Shipping (EMS)	25 EUR
Shipping (DHL)	50 EUR

---

[7] https://aws.amazon.com/marketplace/pp/B07RLRDXL8

[8] https://coinmarketcap.com/currencies/ethereum/

Table 6.5: Traditional Letter of Credit costs (export)

Activity	**Raiffeisen Bank OÖ**[41]	**Bank Austria**[5]	**Commerzbank**[19]
Pre-advising documentary credits	-	-	50 EUR
Advising of documentary credits	75 EUR	80 EUR	0.15 %; at least 200 EUR
Confirmation of documentary credits	250 EUR	0.1% per 30 days; at least 150 EUR	at least 1.2% per year; at least 140 EUR per 90 days
Confirmation commission for the deferred payment period or acceptance commission as from the taking up of documents	0.15%; at least 75 EUR	at least 0.1% per 30 days; at least 150 EUR	1.8% per year, at least 140 EUR per 90 days
Taking up of documents	0.25%, at least 75 EUR	150 EUR	0.2%; at least 250 EUR
Amendment commission	75 EUR	80 EUR	150 EUR
Transference of Letter of Credit	0.4%; at least 200 EUR	0.4%; at least 200 EUR	0.3%; at least 200 EUR

Table 6.6: Costs rating

Platform	**Points**
Hyperledger Fabric	3
Corda	2
Ethereum	1

6.1.4 Governance

Open-source? Adoption of appropriate license necessary? How are decisions about changes to the platform made?

Ethereum is an open-source project and releasing a SC or running a node by yourself does not require any licence. In theory everyone is able to contribute changes to the platform via so called Ethereum Improvement Proposals (EIPs). In practise the process is not that open to changes of everyone. In the end the "All Core Devs" will either accept or reject an EIP after a discussion. Participants of the All Core Devs meeting are members of a number of projects who play a major role in the Ethereum ecosystem. The meeting itself is more of a technical nature. The GNU Lesser General Public License (LGPL) is used which allows developers to integrate Ethereum into their own software without being required to release the source code of their own components. [4, 30]

Hyperledger Fabric is also managed under an open governance model. Everyone is able to contribute by adding feature proposals[9], reporting bugs, updating translations and helping the development. A contributor may become a maintainer after a majority approval by existing maintainers. Projects and sub-projects are lead by a set of maintainers. All Hyperledger Fabric business and marketing matters are overseen by the Governing Board, which consists of up to twenty-one premier members [45]. A premier membership and therefore representation in the Governing Board can be purchased and has a cost of 250,000 USD[10]. The Apache License Version 2.0 is used which allows developers to use the software for any purpose, modify it and distribute it.

R3 governed by default the Corda Network (along with Corda) and was accountable for key decisions. However, the Corda Network Foundation was established to be able to provide more transparent decisions moving forward [21]. The structure is similar to Hyperledger Fabric and includes a Governing Board with elected members. Members of the Governing Board are two employees of R3 and seven external members[11] from various financial services and blockchain technology companies. Users are able to contribute code[12] comparable to the other platforms. Corda is also using the Apache License Version

---

[9] `https://hyperledger-fabric.readthedocs.io/en/latest/CONTRIBUTING.html`

[10] `https://www.hyperledger.org/about/join`

[11] `https://corda.network/governance/board-election/`

[12] `https://docs.corda.net/docs/corda-os/4.5/contributing.html`

2.0 but in addition R3 provides a Corda Enterprise license which adds professional support among other features.

Table 6.7: Governance rating

Platform	**Points**
Ethereum	3
Hyperledger Fabric	2
Corda	1

6.2 Prototype Development

The prototype development evaluation section will be rated based on the experiences we have made while developing the three prototypes.

6.2.1 Usability

Comprehensive documentation of the platform available? A lot of effort to set-up the development environment?

The documentation of all three platforms is comprehensive enough for developers without any DLT background to dive into the matter fast. Each platform has in-depth descriptions about the components involved and provides code examples. The most user friendly way to start programming is the Remix IDE[13] for Ethereum. Remix is an officialy supported, browser-based compiler and IDE and allows the user to build Solidity SC without any local set-up.

With Hyperledger Fabric on the other hand it was not that simple to start the development, especially not our proposed use-case with three participants sharing a channel. The steps to get a test-network up and running involved cloning the samples github repository, installing the correct docker images of the latest production release of the platform, adding the executables to the environment path and editing a lot scripts. Compared with the other two platforms Fabric needed the most time and effort to get started.

Corda does not offer any browser-based IDE like Ethereum but the set-up process is very simple and nothing compared to Hyperledger Fabric. The first step was to clone the template github repository. Inside this project are gradle tasks in which the test-network is defined. Adding the third party for our trade finance process was completed with adding a few new lines in this single file.

---

[13] `https://github.com/ethereum/remix-project`

Table 6.8: Usability rating

Platform	**Points**
Ethereum	3
Corda	2
Hyperledger Fabric	1

6.2.2 Functionality

Is is possible to implement all methods as specified in chapter 4?

Our original vision was to remove intermediaries and allow companies to go on with their business without much overhead. All three platforms help to avoid unnecessary expenses like sending documents to confirm details and manual processing. But only Ethereum also allows to complete the payment between the stakeholders with the built-in token. The token itself may be rather volatile compared to the Dollar or Euro and therefore an unwanted risk for the companies involved. This problem is solved by so called stable-coins but adds administrative overhead itself. When using Hyperledger Fabric[14] or Corda[15] the developers have to include a token on their own. Such a custom token does not have any real world value without agreements stating otherwise. One advantage of a custom token is the possibility to move the agreed value from one party to another without any proprietary functions.

Table 6.9: Functionality rating

Platform	**Points**
Ethereum	3
Hyperledger Fabric	2
Corda	2

6.2.3 Testability

How to test the correctness of a SC? Are there any official tools?

We have used the Truffle Suite to develop the trade finance SC for the Ethereum platform. This framework also allows automated testing with either JavaScript or Solidity. While Truffle is not officially supported by Ethereum, the testing process is straight forward and easy to use.

---

[14] `https://medium.com/@blockchain_simplified/creating-tokens-on-hyperledger-fabric-2-0-us`

[15] `https://github.com/corda/token-sdk`

For Hyperledger Fabric we had to first deploy the chaincode to some docker containers and then start the client applications manually. For the client applications themselves we had to specify some configuration paths and again had to invest more time to get it working as intended compared to the other two platforms. In the end the platform allows testing without any problems.

The Corda template project already has some pre-configured JUnit tests and integrating our three nodes test network was completed within a few lines of code. A difference with testing the Corda SC compared to the other two platforms is that we do not call the methods directly as the programming logic is split into the transaction validation and flow. Instead we instantiate flow objects and execute them. This difference is architecture based and in the end does not really affect the outcome of the testing.

Table 6.10: Testability rating

Platform	**Points**
Corda	3
Ethereum	2
Hyperledger Fabric	2

6.2.4 Flexibility

General-purpose or domain-specific programming language? Introducing new trading partners into the network?

With Solidity Ethereum introduced a domain-specific language to develop SC on its platform. There are multiple problems with domain-specific languages. It slows down the innovative process as the programmers have to study the language first and are not able to instantly produce SCs to solve problems. Another aspect is the possibility of security issues because the developer is not used to possible pitfalls and introduces some vulnerabilities, as it happened with the DAO attack[16] in 2016. With Ethereum we deploy the SC only once and are able to use the deployment with multiple different trading partners.

Hyperledger Fabric and Corda both allow SCs written in general-purpose languages like Go, Java, TypeScript, Kotlin and so on. Therefore we classify them as more beginner friendly. Corda and Fabric also provide Docker support out of the box to isolate the nodes and SC execution environment from the operating system. The virtualisation based on Docker also enables fast deployment to new physical machines. With Hyperledger Fabric and Corda we would have to create a new channel for each trading partner and deploy the SC again to be able to use our process with different partners.

---

[16] `https://en.wikipedia.org/wiki/The_DAO_(organization)`

Table 6.11: Flexibility rating

Platform	**Points**
Corda	3
Hyperledger Fabric	3
Ethereum	1

6.3 Result

Based on our findings the recommended DLT platform is Hyperledger Fabric. The main advantages over Ethereum are caused by the different platform architecture which result in massive performance differences and easier cost estimations. Ethereum would be the pick if the transactions per second or average time for block confirmation measurements are less relevant for the user than an easy set-up procedure or the possibility to remove banks completely from the process. In many areas Corda is similar to Hyperledger Fabric but it lacked transparency at the start of the project and as the Governing Board always includes two employees of R3 the platform decisions are to some extend within the control of R3. In addition, to reach a comparable performance to Hyperledger Fabric a purchase of the closed-source commercial Corda Enterprise version is necessary. Corda provides an excellent out-of-the-box testing support with JUnit integration and the ability to mock a complete network within a few lines of code.

Table 6.12: Evaluation results

Criteria	**Ethereum**	**Hyperledger Fabric**	**Corda**
Performance	1	3	2
Confidentiality	1	3	3
Costs	1	3	2
Governance	3	2	1
Usability	3	1	2
Functionality	3	2	2
Testability	2	2	3
Flexibility	1	3	3
Result	**15**	**19**	**18**

CHAPTER 7

Conclusion

In this work, we analysed the challenges and trade-offs in developing financial instruments on a blockchain. In particular, we compared the process of implementing a relevant application, Letter of Credit (L/C), on three prominent blockchains. We relied on an assortment of criteria from existing catalogues selected according to relevance to our use case.

All considered platforms provide the technological means to develop L/C workflows with reasonable effort. Ethereum, due to its large user base and its age, has the most mature tools and the most comprehensive documentation. Hyperledger Fabric and Corda offer modularity, e.g. regarding the choice of programming language and consensus mechanism.

Major differences surface in regards to performance, costs, confidentiality, and governance due to the different nature of private and public blockchains. Ethereum as a highly distributed public blockchain is transparent both in daily operation as well as in its governance structures. Moreover, it provides an established crypto-currency for the exchange of values. Private blockchains like Hyperledger Fabric and Corda benefit from low transaction costs and high transaction rates. However, structures have to be built on a case-by-case basis and are as strong as the parties involved.

None of the three platforms supports a L/C implementation without any drawbacks. While only Ethereum with its crypto-currency fulfils the requirement of removing the banks from the process and therefore helps to decentralise trade-finance, the gap in performance to private-permissioned blockchain platforms is evident.

In contrast to Ethereum, deploying code or executing a transaction on Hyperledger Fabric and Corda does not incur any costs. Despite this difference, based on our analysis, a cost saving within a low number of trades compared to a traditional L/C process can be achieved.

It would be interesting to extend our comparison both in breadth and depth. On the one hand, new platforms keep emerging that aim at overcoming known limitations. On the other hand, financial instruments on the blockchain should strive for support of aspects that are not purely technical but involve incentives and governance, like dispute resolution.

List of Figures

List of Tables

Acronyms

AMQP Advanced Message Queue Protocol. 30

AWS Amazon Web Services. 79

B/L Bill of Lading. 12, 13

BFT Byzantine Fault Tolerant. 31

CA Certificate Authority. 51

CIA Cash in Advance. 12

CorDapp Corda Distributed Application. 30

DApps Decentralized Applications. 20, 21, 23, 25, 26

DLT Distributed Ledger Technology. 2–4, 6, 8, 9, 11, 14, 16–21, 23, 30, 31, 33, 48, 51, 59, 60, 77, 80, 83, 86, 89, 91

DoS denial-of-service. 27

DSR Design Science Research. 8

EIP Ethereum Improvement Proposal. 82

EVM Ethereum Virtual Machine. 23

GHOST Greedy Heaviest Observed Subtree. 24

ICO Initial Coin Offering. 6

L/C Letter of Credit. 1, 2, 4, 6, 9, 12–14, 80, 87, 89

LGPL GNU Lesser General Public License. 82

MSP Membership Service Provider. 26

OA	Open Account. 12
OSN	Ordering Service Nodes. 27, 28
PoW	Proof-of-Work. 23, 24
RPC	Remote Procedure Call. 60, 69
SaaS	Software as a Service. 79
SC	Smart Contract. 2, 3, 7–9, 11, 17, 21, 24, 30, 33, 34, 37, 38, 48, 49, 51, 56, 57, 59, 62, 75, 78, 82–85
SLR	Scientific Literature Review. 5
UTXO	Unspent Transaction Output. 30, 59, 63

References

[1] S. Aggarwal, R. Chaudhary, G. S. Aujla, N. Kumar, K.-K. R. Choo, and A. Y. Zomaya. Blockchain for smart communities: Applications, challenges and opportunities. *Journal of Network and Computer Applications*, 144:13 – 48, 2019. ISSN 1084-8045. doi: https://doi.org/10.1016/j.jnca.2019.06.018. URL `http://www.sciencedirect.com/science/article/pii/S1084804519302231`.

[2] Amazon. Amazon managed blockchain pricing. `https://aws.amazon.com/managed-blockchain/pricing/`, 2020. Accessed: 2020-11-01.

[3] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[4] A. M. Antonopoulos and G. Wood. *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[5] Bank Austria. Konditionen für Dokumenten-Akkreditive. `https://www.bankaustria.at/files/PB_Konditionen_Dok.PDF`, 2009. Accessed: 2020-11-05.

[6] S. Beck, R. Bunting, and C. Sutken. Effective practices in trade finance examinations. *Asian Development Bank*, 2019. URL: `http://dx.doi.org/10.22617/BRF190582-2`, Accessed on 2020-11-24.

[7] M. Belotti, N. Božić, G. Pujolle, and S. Secci. A vademecum on blockchain technologies: When, which, and how. *IEEE Communications Surveys & Tutorials*, 21 (4):3796–3838, 2019.

[8] T. Bhogal and A. Trivedi. Blockchain technology and trade finance. In *International Trade Finance*, pages 303–312. Springer, 2019.

[9] A. Blum. Blockchain and trade finance: A smart contract-based solution. *University of Basel*, 2019. Master Thesis, URL: `https://wwz.unibas.ch/fileadmin/user_upload/wwz/00_Professuren/Schaer_DLTFintech/Lehre/Blum_2019.pdf`, Accessed on 2020-11-24.

[10] A. Bogucharskov, I. Pokamestov, K. Adamova, and Z. Tropina. Adoption of blockchain technology in trade finance process. *Journal of Reviews on Global Economics*, 7, 2018.

[11] A. Botta, N. Digiacomo, and R. Ritter. Technology innovations driving change in transaction banking. 2016. URL: `https://www.mckinsey.com/industries/financial-services/our-insights/technology-innovations-driving-change-in-transaction-banking#`, Accessed on 2020-04-25.

[12] R. G. Brown. The corda platform: An introduction. *R3 CEV*, 2018. URL: `https://www.r3.com/wp-content/uploads/2019/06/corda-platform-whitepaper.pdf`, Accessed: 2020-04-15.

[13] D. Burkhardt, M. Werling, and H. Lasi. Distributed ledger. In *2018 IEEE international conference on engineering, technology and innovation (ICE/ITMC)*, pages 1–9. IEEE, 2018.

[14] V. Buterin. Ethereum: A next-generation smart contract and decentralized application platform. `https://ethereum.org/whitepaper/`, 2014. Accessed: 2020-04-15.

[15] B. Cant, A. Khadikar, A. Ruiter, J. B. Bronebakk, J. Coumaros, J. Buvat, and A. Gupta. Smart contracts in financial services: Getting from hype to reality. *Capgemini consulting*, pages 1–24, 2016.

[16] S. Chang, Y.-C. Chen, and T.-C. Wu. Exploring blockchain technology in international trade: Business process re-engineering for letter of credit. *Industrial Management and Data Systems*, 119, 2019.

[17] S. Chang, H. Luo, and Y. Chen. Blockchain-enabled trade finance innovation: A potential paradigm shift on using letter of credit. *Sustainability (Switzerland)*, 12, 2020.

[18] P. B. Checkland. Soft systems methodology. *Human systems management*, 8(4): 273–289, 1989.

[19] Commerzbank. Konditionsliste für Firmenkunden der Commerzbank Zrt. `https://www.firmenkunden.commerzbank.de/portal/media/corporatebanking/auslandsseiten/ungarn-informationen/news-3/DE_Commerzbank_Standard_Konditionliste_20190401.pdf`, 2019. Accessed: 2020-11-05.

[20] L. W. Cong and Z. He. Blockchain disruption and smart contracts. *The Review of Financial Studies*, 32(5):1754–1797, 2019.

[21] Corda Network Foundation. Governance guidelines. `https://corda.network/governance/governance-guidelines/`, 2020. Accessed: 2020-10-17.

[22] V. A. Ermakov, E. M. Burmistrova, N. B. Bodin, A. A. Chursin, and E. A. Shevereva. A letter of credit as an instrument to mitigate risks and improve the efficiency of foreign trade transaction. *Espacios*, 39, 2018.

[23] P. Fichtinger. Solidity design patterns. *TU Wien*, 2018. Bachelor Thesis.

[24] S. Ganesh, T. Olsen, J. Kroeker, and V. P. Rebooting a digital solution to trade finance. 2018. URL: `https://www.bain.com/insights/rebooting-a-digital-solution-to-trade-finance/`, Accessed on 2020-04-25.

[25] A. Grath. *The handbook of international trade and finance: the complete guide to risk management, international payments and currency management, bonds and guarantees, credit insurance and trade finance.* Kogan Page Publishers, 2011.

[26] M. Hearn. Corda: A distributed ledger. *Corda Technical White Paper*, 2016. URL: `https://www.r3.com/wp-content/uploads/2019/08/corda-technical-whitepaper-August-29-2019.pdf`, Accessed: 2020-04-15.

[27] A. Hevner, S. T. March, J. Park, S. Ram, et al. Design science research in information systems. *MIS quarterly*, 28(1):75–105, 2004.

[28] IBM. Ibm blockchain platform – pricing. `https://www.ibm.com/cloud/blockchain-platform/pricing`, 2020. Accessed: 2020-11-01.

[29] L. Ismail and H. Materwala. A review of blockchain architecture and consensus protocols: Use cases, challenges, and solutions. *Symmetry*, 11(10):1198, 2019.

[30] H. Jameson. Ethereum protocol development governance and network upgrade coordination. `https://hudsonjameson.com/2020-03-23-ethereum-protocol-development-governance-and-network-upgrade-coor`
2020. Accessed: 2020-10-20.

[31] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev. What does not fit can be made to fit! trade-offs in distributed ledger technology designs. In *Proceedings of the 52nd Hawaii International Conference on System Sciences*, 2019.

[32] S. Kim, S. Park, Y. B. Park, J. A. Kim, Y. Cho, J. Choi, and C. Kim. A feature based content analysis of blockchain platforms. In *2018 Tenth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 791–793, 2018.

[33] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. *EBSE Technical Report*, 2007.

[34] Y. G. Liang. Blockchain application and outlook in the banking industry. *Financial Innovation*, 2, 2016.

[35] F. Murshudli and B. Loguinov. Digitalization challenges to global banking industry. *Economic and Social Development: Book of Proceedings*, pages 786–794, 2019.

[36] R. B. Myerson. Game theory: Analysis of conflict (6. print ed.). *Harvard Univ. Press, Cambridge, Mass*, 2004.

[37] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009. URL `http://www.bitcoin.org/bitcoin.pdf`.

[38] J. F. Nunamaker Jr, M. Chen, and T. D. Purdin. Systems development in information systems research. *Journal of management information systems*, 7(3):89–106, 1990.

[39] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.

[40] S. Pongnumkul, C. Siripanpornchana, and S. Thajchayapong. Performance analysis of private blockchain platforms in varying workloads. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2017. doi: 10.1109/ICCCN.2017.8038517.

[41] Raiffeisen Bank Oberösterreich. Preisaushang 01.10.2020. `https://www.raiffeisen.at/ooe/gampern/de/meine-bank/schalteraushang/_jcr_content/root/responsivegrid/tabaccordioncontaine/tabAccordionElements/tabaccordionelement_2086307275/items/downloadlist.download.html/0/Preisaushang.pdf`, 2020. Accessed: 2020-11-05.

[42] C. Saraf and S. Sabadra. Blockchain platforms: A compendium. In *2018 IEEE International Conference on Innovative Research and Development (ICIRD)*, pages 1–6, 2018.

[43] A. Sunyaev. Distributed ledger technology. In *Internet Computing*, pages 265–299. Springer, 2020.

[44] N. Szabo. Formalizing and securing relationships on public networks. *First Monday – Peer-reviewed Journal on the Internet*, 2(9), September 1997.

[45] The Linux Foundation. Hyperledger project charter. `https://www.hyperledger.org/about/charter`, 2019. Accessed: 2020-10-17.

[46] T. Travel and D. Mohanty. R3 corda for architects and developers.

[47] M. Valenta and P. Sandner. Comparison of ethereum, hyperledger fabric and corda. *no. June*, pages 1–8, 2017. Accessed: 2020-04-15.

[48] J. Venable. A framework for design science research activities. In *Emerging Trends and Challenges in Information Technology Management: Proceedings of the 2006 Information Resource Management Association Conference*, pages 184–187. Idea Group Publishing, 2006.

[49] J. Venable, J. Pries-Heje, and R. Baskerville. A comprehensive framework for evaluation in design science research. In *International Conference on Design Science Research in Information Systems*, pages 423–438. Springer, 2012.

[50] M. Vinayak, H. A. P. S. Panesar, S. dos Santos, R. K. Thulasiram, P. Thulasiraman, and S. Appadoo. Analyzing financial smart contracts for blockchain. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1701–1706. IEEE, 2018.

[51] M. Vinayak, S. Santos, R. Thulasiram, P. Thulasiraman, and S. Appadoo. Design and implementation of financial smart contract services on blockchain. *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference, IEMCON 2019*, 2019.

[52] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `https://ethereum.github.io/yellowpaper/paper.pdf`, 2017. Accessed: 2020-04-15.

[53] World Trade Organization. World trade statistical review 2019. 2019. URL: `https://www.wto.org/english/res_e/statis_e/wts2019_e/wts19_toc_e.htm`, Accessed on 2020-04-25.

[54] X. Xu, I. Weber, M. Staples, L. Zhu, J. Bosch, L. Bass, C. Pautasso, and P. Rimba. A taxonomy of blockchain-based systems for architecture design. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 243–252. IEEE, 2017.

[55] X. Xu, I. Weber, and M. Staples. *Architecture for blockchain applications*. Springer, 2019.

[56] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang. Blockchain challenges and opportunities: A survey. *International Journal of Web and Grid Services*, 14(4): 352–375, 2018.

Appendix A - Ethereum Prototype Code

TradeFinanceContract.sol

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.6.0;

contract TradeFinanceContract {
    enum States {
        NONE,
        CREATED,
        CONFIRMED,
        SHIPPED,
        DELIVERED,
        CLOSED,
        CANCELLED,
        PASSED
    }

    address payable internal seller;

    struct Order {
        States state;
        address payable buyer;
        uint256 orderId;
        uint256 productId;
        uint256 quantity;
        uint256 price;
        string shippingAddress;
        uint256 latestDeliveryDate;
        address payable freightCompany;
        uint256 shippingCosts;
        string trackingCode;
        bool buyerSigned;
        bool freightSigned;
    }

    uint256 orderCount;
    mapping(uint256 => Order) public orders;
```

```
    mapping(address => uint256) public balances;

    event Log(uint256 orderId, string text);

    constructor() public {
        seller = msg.sender;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only the seller is allowed to call this function."
        );
        _;
    }

    modifier onlyBuyer(uint256 orderId) {
        require(
            msg.sender == orders[orderId].buyer,
            "Only the buyer is allowed to call this function."
        );
        _;
    }

    modifier onlySellerOrBuyer(uint256 orderId) {
        require(
            msg.sender == seller || msg.sender == orders[orderId].buyer,
            "Only the buyer and seller are allowed to call this function."
        );
        _;
    }

    modifier onlyFreightCompanyOrBuyer(uint256 _orderId) {
        require(
            msg.sender == orders[_orderId].freightCompany ||
                msg.sender == orders[_orderId].buyer,
            "Only the buyer and freight company are allowed to call this function."
        );
        _;
    }

    modifier atState(uint256 _orderId, States _state) {
        require(
            orders[_orderId].state == _state,
            "Function cannot be called at this state."
        );
        _;
    }

    modifier transitionNextState(uint256 _orderId) {
        _;
        nextState(_orderId);
```

```
    }

    function nextState(uint256 _orderId) internal {
        orders[_orderId].state = States(uint256(orders[_orderId].state) + 1);
    }

    function getOrderCount() public view returns (uint256) {
        return orderCount;
    }

    function getOrderState(uint256 _orderId) public view returns (States) {
        return orders[_orderId].state;
    }

    function addOrder(
        uint256 _orderId,
        address payable _buyer,
        uint256 _productId,
        uint256 _quantity,
        uint256 _price,
        string memory _shippingAddress,
        uint256 _latestDeliveryDate,
        uint256 _shippingCosts
    )
        public
        onlySeller
        atState(_orderId, States.NONE)
        transitionNextState(_orderId)
    {
        require(
            orders[_orderId].orderId != _orderId,
            "An order with this ID already exists."
        );
        require(
            _price >= _shippingCosts,
            "The price must be greater or equal to the shipping costs."
        );

        orders[_orderId].orderId = _orderId;
        orders[_orderId].buyer = _buyer;
        orders[_orderId].productId = _productId;
        orders[_orderId].quantity = _quantity;
        orders[_orderId].price = _price;
        orders[_orderId].shippingCosts = _shippingCosts;
        orders[_orderId].shippingAddress = _shippingAddress;
        orders[_orderId].latestDeliveryDate = _latestDeliveryDate;
        orderCount++;
        emit Log(_orderId, "Order has been added");
    }

    function cancelOrder(uint256 _orderId) public onlySellerOrBuyer(_orderId)
         {
        require(
```

```
            orders[_orderId].state == States.CREATED ||
                orders[_orderId].state == States.CONFIRMED,
            "Function cannot be called at this state."
        );

        if (orders[_orderId].state == States.CONFIRMED) {
            orders[_orderId].state = States.CANCELLED;
            balances[orders[_orderId].buyer] -= orders[_orderId].price;
            orders[_orderId].buyer.transfer(orders[_orderId].price);
        } else {
            orders[_orderId].state = States.CANCELLED;
        }
        emit Log(_orderId, "Order has been cancelled");
    }

    function deliveryDatePassed(uint256 _orderId) public {
        require(
            block.timestamp >= orders[_orderId].latestDeliveryDate,
            "Delivery date did not pass yet."
        );
        require(
            orders[_orderId].state < States.DELIVERED,
            "Order got already delivered."
        );
        require(
            orders[_orderId].freightSigned == false,
            "Refund not possible as the freight company already signed the arrival."
        );

        orders[_orderId].state = States.PASSED;
        if (orders[_orderId].state >= States.CONFIRMED) {
            balances[orders[_orderId].buyer] -= orders[_orderId].price;
            orders[_orderId].buyer.transfer(orders[_orderId].price);
        }
        emit Log(
            _orderId,
            "Order has been cancelled due passed delivery date."
        );
    }

    function confirmOrder(uint256 _orderId)
        public
        payable
        onlyBuyer(_orderId)
        atState(_orderId, States.CREATED)
        transitionNextState(_orderId)
    {
        require(
            orders[_orderId].price == msg.value,
            "Not enough Ether sent to cover the price of the order."
        );
        balances[orders[_orderId].buyer] += orders[_orderId].price;
```

```
        emit Log(_orderId, "Order has been confirmed and money deposited");
    }

    function shipOrder(
        uint256 _orderId,
        address payable _freightCompany,
        string memory _trackingCode
    )
        public
        onlySeller
        atState(_orderId, States.CONFIRMED)
        transitionNextState(_orderId)
    {
        orders[_orderId].freightCompany = _freightCompany;
        orders[_orderId].trackingCode = _trackingCode;
        emit Log(_orderId, "Order has been shipped");
    }

    function signArrival(uint256 _orderId)
        public
        onlyFreightCompanyOrBuyer(_orderId)
        atState(_orderId, States.SHIPPED)
    {
        if (msg.sender == orders[_orderId].buyer) {
            orders[_orderId].buyerSigned = true;
            emit Log(_orderId, "Order arrival has been signed by the buyer");
        }

        if (msg.sender == orders[_orderId].freightCompany) {
            orders[_orderId].freightSigned = true;
            emit Log(
                _orderId,
                "Order arrival has been signed by the freight company"
            );
        }

        if (orders[_orderId].buyerSigned && orders[_orderId].freightSigned) {
            nextState(_orderId);
            emit Log(
                _orderId,
                "Order arrival has been signed by the buyer and freight company"
            );
            payout(_orderId);
        }
    }

    function payout(uint256 _orderId)
        private
        atState(_orderId, States.DELIVERED)
        transitionNextState(_orderId)
    {
        balances[orders[_orderId].buyer] -= orders[_orderId].price;
```

```
        balances[seller] =
            balances[seller] +
            orders[_orderId].price -
            orders[_orderId].shippingCosts;
        balances[orders[_orderId].freightCompany] += orders[_orderId]
            .shippingCosts;

        seller.transfer(
            orders[_orderId].price - orders[_orderId].shippingCosts
        );
        orders[_orderId].freightCompany.transfer(
            orders[_orderId].shippingCosts
        );

        emit Log(_orderId, "Payout finished.");
    }
}
```

tradefinance.js (Tests)

```
const TradeFinanceContract = artifacts.require("TradeFinanceContract");

contract("TradeFinanceContract", accounts => {
    let seller = accounts[0];
    let buyer = accounts[1];
    let freightCompany = accounts[2];

    it("check test environment", () => {
        TradeFinanceContract.deployed()
            .then(instance => instance.getOrderCount())
            .then(orderCount => {
                assert.equal(
                    orderCount.toNumber(),
                    0,
                    "the order count after adding an order was not 0"
                );
            });
    });

    it("create order test", () => {
        let instance;

        return TradeFinanceContract.deployed()
            .then(inst => {
                instance = inst;
                return instance.addOrder(1, buyer, 100, 2, web3.utils.toWei("
    10", "ether"), "Karlsplatz 13, 1040 Wien", 1594771200, web3.utils.toWei("
    2", "ether"), { from: seller });
            })
            .then(() => instance.getOrderCount())
            .then(orderCount => {
                assert.equal(
```

```
                orderCount.toNumber(),
                1,
                "the order count after adding an order was not 1"
            );
        })
        .then(() => instance.getOrderState(1))
        .then(orderState => {
            assert.equal(
                orderState.toNumber(),
                1,
                "the order state after adding was not CREATED (1)."
            );
        })
});

it("confirm order test", () => {
    let instance;

    return TradeFinanceContract.deployed()
        .then(inst => {
            instance = inst;
            return instance.addOrder(2, buyer, 100, 2, web3.utils.toWei("10", "ether"), "Karlsplatz 13, 1040 Wien", 1594771200, web3.utils.toWei("2", "ether"), { from: seller });
        })
        .then(() => instance.getOrderCount())
        .then(orderCount => {
            assert.equal(
                orderCount.toNumber(),
                2,
                "the order count after adding an order was not 1"
            );
        })
        .then(() => instance.confirmOrder(2, { from: buyer, value: web3.utils.toWei("10", "ether") }))
        .then(() => instance.getOrderState(2))
        .then(orderState => {
            assert.equal(
                orderState.toNumber(),
                2,
                "the order state after confirming was not CONFIRMED (2)."
            );
        })
});

it("sign arrival test", () => {
    let instance;

    return TradeFinanceContract.deployed()
        .then(inst => {
            instance = inst;
            return instance.addOrder(3, buyer, 123587, 5.0, web3.utils.toWei("15", "ether"), "Ballhausplatz 2, 1010 Wien", 1594771200, web3.
```

```
utils.toWei("3", "ether"), { from: seller });
            })
            .then(() => instance.confirmOrder(3, { from: buyer, value: web3.
utils.toWei("15", "ether") }))
            .then(() => instance.shipOrder(3, freightCompany, "1AXCAW311", {
from: seller }))
            .then(() => instance.signArrival(3, { from: buyer }))
            .then(() => instance.signArrival(3, { from: freightCompany }))
            .then(() => instance.getOrderState(3))
            .then(orderState => {
                assert.equal(
                    orderState.toNumber(),
                    5,
                    "the order state after confirming was not CLOSED (5)."
                );
            })
    });

    it("delivery date passed test", () => {
        let instance;

        return TradeFinanceContract.deployed()
            .then(inst => {
                instance = inst;
                return instance.addOrder(4, buyer, 123587, 5.0, web3.utils.
toWei("15", "ether"), "Ballhausplatz 2, 1010 Wien", 1594771200, web3.
utils.toWei("3", "ether"), { from: seller });
            })
            .then(() => instance.confirmOrder(4, { from: buyer, value: web3.
utils.toWei("15", "ether") }))
            .then(() => instance.shipOrder(4, freightCompany, "1AXCAW311", {
from: seller }))
            .then(() => instance.deliveryDatePassed(4, { from: buyer }))
            .then(() => instance.getOrderState(4))
            .then(orderState => {
                assert.equal(
                    orderState.toNumber(),
                    7,
                    "the order state after confirming was not PASSED (7)."
                );
            })
    });

});
```

Appendix B - Hyperledger Fabric Prototype Code

index.ts

```
/*
 * SPDX-License-Identifier: Apache-2.0
 */

import { TradeFinance } from "./trade";
export { TradeFinance } from "./trade";

export const contracts: any[] = [TradeFinance];
```

order.ts

```
/*
 * SPDX-License-Identifier: Apache-2.0
 */

export enum State {
    CREATED,
    CONFIRMED,
    SHIPPED,
    DELIVERED,
    CANCELLED,
    PASSED
}

export class Order {
    public docType?: string;
    public state: State;
    public orderId: string;
    public productId: number;
    public quantity: number;
    public price: number;
    public shippingCosts: number;
    public shippingAddress: string;
```

```
    public latestDeliveryDate: Date;
    public trackingCode: string;
    public buyerSigned: boolean;
    public freightSigned: boolean;
}
```

trade.ts

```
/*
 * SPDX-License-Identifier: Apache-2.0
 */

import { Context, Contract } from "fabric-contract-api";
import { Order, State } from "./order";

export class TradeFinance extends Contract {

    private restrictedCall(ctx: Context, allowedAffiliation: string) {
        if (!ctx.clientIdentity.assertAttributeValue("hf.Affiliation", allowedAffiliation)) {
            throw new Error("Only users with affiliation " + allowedAffiliation + " are allowed to call this function");
        }
    }

    private restrictedCall2(ctx: Context, allowedAffiliation1: string, allowedAffiliation2: string) {
        if (!ctx.clientIdentity.assertAttributeValue("hf.Affiliation", allowedAffiliation1) && !ctx.clientIdentity.assertAttributeValue("hf.Affiliation", allowedAffiliation2)) {
            throw new Error("Only users with affiliation " + allowedAffiliation1 + " or " + allowedAffiliation2 + " are allowed to call this function.");
        }
    }

    private async getOrder(ctx: Context, _orderId: string): Promise<Order> {
        const orderAsBytes = await ctx.stub.getState(_orderId);
        if (orderAsBytes.length === 0) {
            throw new Error("An order with ID " + _orderId + " does not exist");
        }
        const order: Order = JSON.parse(orderAsBytes.toString());
        return order;
    }

    public async queryOrder(ctx: Context, _orderId: string): Promise<string> {
        const order = await this.getOrder(ctx, _orderId);
        //console.log(order.toString());
        return JSON.stringify(order);
    }
```

```

public async queryAllOrders(ctx: Context): Promise<string> {
    const startKey = "";
    const endKey = "";
    const allResults = [];
    for await (const { key, value } of ctx.stub.getStateByRange(startKey,
 endKey)) {
        const strValue = Buffer.from(value).toString("utf8");
        let record;
        try {
            record = JSON.parse(strValue);
        } catch (err) {
            console.log(err);
            record = strValue;
        }
        allResults.push({ Key: key, Record: record });
    }
    //console.info(allResults);
    return JSON.stringify(allResults);
}

public async createOrder(ctx: Context,
    _orderId: string,
    _productId: number,
    _quantity: number,
    _price: number,
    _shippingCosts: number,
    _shippingAddress: string,
    _latestDeliveryDate: string) {
    console.info("============= START : Create Order ===========");

    this.restrictedCall(ctx, "seller");
    const orderAsBytes = await ctx.stub.getState(_orderId);
    if (orderAsBytes.length > 0) {
        throw new Error("An order with ID " + _orderId + " does already
exist");
    }

    _productId = Number(_productId);
    _quantity = Number(_quantity);
    _price = Number(_price);
    _shippingCosts = Number(_shippingCosts);

    if (_price < _shippingCosts) {
        throw new Error("The price must be greater or equal to the
shipping costs.");
    }

    var splittedDate = _latestDeliveryDate.split("-"); // date given in
yyyy-mm-dd format
    var parsedDate = new Date(parseInt(splittedDate[0]), parseInt(
splittedDate[1]) - 1, parseInt(splittedDate[2]));
    //console.info("parsedDate:" + parsedDate.toLocaleString());
```

```

    const order: Order = {
        docType: "order",
        state: State.CREATED,
        orderId: _orderId,
        productId: _productId,
        quantity: _quantity,
        price: _price,
        shippingCosts: _shippingCosts,
        shippingAddress: _shippingAddress,
        latestDeliveryDate: parsedDate,
        trackingCode: undefined,
        buyerSigned: undefined,
        freightSigned: undefined
    };

    await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)))
;
    console.info("============= END : Create Order ===========");
}

public async cancelOrder(ctx: Context, _orderId: string) {
    console.info("============= START : cancelOrder ===========");

    this.restrictedCall2(ctx, "seller", "buyer");
    const order = await this.getOrder(ctx, _orderId);

    if (order.state == State.DELIVERED || order.state == State.SHIPPED ||
 order.state == State.CANCELLED || order.state == State.PASSED) {
        throw new Error("The state of order " + _orderId + "  does not
allow this action");
    }

    order.state = State.CANCELLED;

    await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)))
;
    console.info("Order " + _orderId + " has been cancelled.");
    console.info("============= END : cancelOrder ===========");
}

public async deliveryDatePassed(ctx: Context, _orderId: string): Promise<
boolean> {
    console.info("============= START : deliveryDatePassed ===========");
    var passed = false;

    const order = await this.getOrder(ctx, _orderId);

    if (order.state >= State.DELIVERED) {
        throw new Error("The state of order " + _orderId + "  does not
allow this action");
    }

```

```
    var currentDate = new Date();
    if (currentDate > new Date(order.latestDeliveryDate)) {
        order.state = State.PASSED;
        await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
        passed = true;
        console.info("Order " + _orderId + " has been cancelled due passed delivery date.");
    }

    console.info("============= END : deliveryDatePassed ===========");
    return passed;
}

public async confirmOrder(ctx: Context, _orderId: string) {
    console.info("============= START : confirmOrder ===========");

    this.restrictedCall(ctx, "buyer");
    const order = await this.getOrder(ctx, _orderId);

    if (order.state != State.CREATED) {
        throw new Error("The state of order " + _orderId + "  does not allow this action");
    }

    order.state = State.CONFIRMED;

    await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
    console.info("Order " + _orderId + " has been confirmed.");
    console.info("============= END : confirmOrder ===========");
}

public async shipOrder(ctx: Context, _orderId: string, _trackingCode: string) {
    console.info("============= START : shipOrder ===========");

    this.restrictedCall(ctx, "seller");
    const order = await this.getOrder(ctx, _orderId);

    if (order.state != State.CONFIRMED) {
        throw new Error("The state of order " + _orderId + " does not allow this action");
    }

    order.state = State.SHIPPED;
    order.trackingCode = _trackingCode;

    await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order)));
    console.info("Order " + _orderId + " has been shipped.");
    console.info("============= END : shipOrder ===========");
}
```

```

    public async signArrival(ctx: Context, _orderId: string) {
        console.info("============= START : signArrival ===========");

        this.restrictedCall2(ctx, "freight", "buyer");
        const order = await this.getOrder(ctx, _orderId);

        if (order.state != State.SHIPPED) {
            throw new Error("The state of order " + _orderId + " does not
    allow this action");
        }

        if (ctx.clientIdentity.assertAttributeValue("hf.Affiliation", "buyer"
    )) {
            order.buyerSigned = true;
            console.info("Order " + _orderId + " arrival has been signed by
    the buyer.");
        }

        if (ctx.clientIdentity.assertAttributeValue("hf.Affiliation", "
    freight")) {
            order.freightSigned = true;
            console.info("Order " + _orderId + " arrival has been signed by
    the freight company.");
        }

        if (order.buyerSigned && order.freightSigned) {
            order.state = State.DELIVERED;
            console.info("Order " + _orderId + " has been delivered.");
        }

        await ctx.stub.putState(_orderId, Buffer.from(JSON.stringify(order))
    ;
        console.info("============= END : signArrival ===========");
    }
}
```

Order.java

```
package org.example;

import java.text.SimpleDateFormat;
import java.util.Date;

import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.annotations.SerializedName;

import org.bouncycastle.util.Strings;

public class Order implements java.io.Serializable {
    /**
```

```
     *
     */
    private static final long serialVersionUID = -1774134125317583092L;
    private static SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ");
    private static Gson gson = new GsonBuilder().setPrettyPrinting().setDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSSZ")
            .create();

    public enum State {
        @SerializedName("0")
        CREATED,

        @SerializedName("1")
        CONFIRMED,

        @SerializedName("2")
        SHIPPED,

        @SerializedName("3")
        DELIVERED,

        @SerializedName("4")
        CANCELLED,

        @SerializedName("5")
        PASSED
    }

    private State state;
    private String orderId;
    private int productId;
    private double quantity;
    private double price;
    private double shippingCosts;
    private String shippingAddress;
    private Date latestDeliveryDate;
    private String trackingCode;
    private boolean buyerSigned;
    private boolean freightSigned;

    public State getState() {
        return this.state;
    }

    public void setState(State state) {
        this.state = state;
    }

    public String getOrderId() {
        return this.orderId;
    }

```

```
    public void setOrderId(String orderId) {
        this.orderId = orderId;
    }

    public int getProductId() {
        return this.productId;
    }

    public void setProductId(int productId) {
        this.productId = productId;
    }

    public double getQuantity() {
        return this.quantity;
    }

    public void setQuantity(double quantity) {
        this.quantity = quantity;
    }

    public double getPrice() {
        return this.price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public double getShippingCosts() {
        return this.shippingCosts;
    }

    public void setShippingCosts(double shippingCosts) {
        this.shippingCosts = shippingCosts;
    }

    public String getShippingAddress() {
        return this.shippingAddress;
    }

    public void setShippingAddress(String shippingAddress) {
        this.shippingAddress = shippingAddress;
    }

    public Date getLatestDeliveryDate() {
        return this.latestDeliveryDate;
    }

    public void setLatestDeliveryDate(Date latestDeliveryDate) {
        this.latestDeliveryDate = latestDeliveryDate;
    }

    public String getTrackingCode() {
```

```java
        return this.trackingCode;
    }

    public void setTrackingCode(String trackingCode) {
        this.trackingCode = trackingCode;
    }

    public boolean isBuyerSigned() {
        return this.buyerSigned;
    }

    public boolean getBuyerSigned() {
        return this.buyerSigned;
    }

    public void setBuyerSigned(boolean buyerSigned) {
        this.buyerSigned = buyerSigned;
    }

    public boolean isFreightSigned() {
        return this.freightSigned;
    }

    public boolean getFreightSigned() {
        return this.freightSigned;
    }

    public void setFreightSigned(boolean freightSigned) {
        this.freightSigned = freightSigned;
    }

    /**
     * Deserialize a string to order object
     *
     * @param data data to form back into the object
     */
    public static Order deserialize(String data) {
        return Order.gson.fromJson(data, Order.class);
    }

    public static Order deserialize(byte[] data) {
        return Order.gson.fromJson(Strings.fromByteArray(data), Order.class);
    }

    /**
     * Serialize an order object to string
     *
     * @param order data to form back into the object
     */
    public static String serialize(Order order) {
        return Order.gson.toJson(order);
    }

```

```java
    @Override
    public String toString() {
        return "{" + " state='" + getState() + "'" + ", orderId='" +
            getOrderId() + "'" + ", productId='"
                + getProductId() + "'" + ", quantity='" + getQuantity() + "'"
                    + ", price='" + getPrice() + "'"
                + ", shippingCosts='" + getShippingCosts() + "'" + ",
                    shippingAddress='" + getShippingAddress() + "'"
                + ", latestDeliveryDate='" + sdf.format(getLatestDeliveryDate
                    ()) + "'" + ", trackingCode='"
                + getTrackingCode() + "'" + ", buyerSigned='" + isBuyerSigned
                    () + "'" + ", freightSigned='"
                + isFreightSigned() + "'" + "}";
    }

}
```

AddToWallet.java (Seller)

```java
package org.example;

import java.io.IOException;
import java.io.Reader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.InvalidKeyException;
import java.security.PrivateKey;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import java.util.stream.Stream;

import javax.naming.InvalidNameException;
import javax.naming.ldap.LdapName;

import org.hyperledger.fabric.gateway.Identities;
import org.hyperledger.fabric.gateway.Identity;
import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;

public class AddToWallet {

  private static X509Certificate readX509Certificate(final Path
      certificatePath)
      throws IOException, CertificateException {
    try (Reader certificateReader = Files.newBufferedReader(certificatePath,
        StandardCharsets.UTF_8)) {
      return Identities.readX509Certificate(certificateReader);
    }
  }

```

```java
private static PrivateKey getPrivateKey(final Path privateKeyPath) throws IOException, InvalidKeyException {
  try (Reader privateKeyReader = Files.newBufferedReader(privateKeyPath, StandardCharsets.UTF_8)) {
    return Identities.readPrivateKey(privateKeyReader);
  }
}

public static void main(final String[] args) {
  try {
    // A wallet stores a collection of identities
    final Path walletPath = Paths.get(".", "wallet");
    final Wallet wallet = Wallets.newFileSystemWallet(walletPath);

    final Path credentialPath = Paths.get("..", "..", "..", "..", "test-network", "organizations",
      "peerOrganizations", "seller.example.com", "users", "User1@seller.example.com", "msp");
    System.out.println("credentialPath: " + credentialPath.toString());
    final Path certificatePath = credentialPath.resolve(Paths.get("signcerts", "cert.pem"));
    System.out.println("certificatePem: " + certificatePath.toString());

    Path privateKeyPath = null;
    try (Stream<Path> paths = Files.find(credentialPath.resolve(Paths.get("keystore")), Integer.MAX_VALUE,
      (path, attrs) -> attrs.isRegularFile() && path.toString().endsWith("_sk"))) {
      privateKeyPath = paths.findAny().get();
    }

    final X509Certificate certificate = readX509Certificate(certificatePath);

    final String identityLabel = new LdapName(certificate.getSubjectX500Principal().getName()).getRdns().stream()
      .filter(i -> i.getType().equalsIgnoreCase("CN")).findFirst().get().getValue().toString();

    final PrivateKey privateKey = getPrivateKey(privateKeyPath);
    final Identity identity = Identities.newX509Identity("SellerMSP", certificate, privateKey);

    wallet.put(identityLabel, identity);

    System.out.println("Write wallet info into " + walletPath.toString() + " successfully.");

  } catch (IOException | CertificateException | InvalidKeyException | InvalidNameException e) {
    System.err.println("Error adding to wallet");
    e.printStackTrace();
  }
```

```java
  }

}
```

ClientApp.java (Seller)

```java
package org.example;

import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.concurrent.TimeoutException;

import org.hyperledger.fabric.gateway.Contract;
import org.hyperledger.fabric.gateway.Gateway;
import org.hyperledger.fabric.gateway.GatewayException;
import org.hyperledger.fabric.gateway.Network;
import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;

public class ClientApp {
  private static final String CONTRACT = "CONTRACT_NAME";
  private static final String CHANNEL = "CHANNEL_NAME";

  public static void main(final String[] args) {
    final Gateway.Builder builder = Gateway.createBuilder();

    String contractName = "trade-finance";
    String channelName = "mychannel";
    // get the name of the contract, in case it is overridden
    final Map<String, String> envvar = System.getenv();
    if (envvar.containsKey(CONTRACT)) {
      contractName = envvar.get(CONTRACT);
    }
    if (envvar.containsKey(CHANNEL)) {
      channelName = envvar.get(CHANNEL);
    }

    try {
      // A wallet stores a collection of identities
      final Path walletPath = Paths.get(".", "wallet");
      final Wallet wallet = Wallets.newFileSystemWallet(walletPath);
      System.out.println("Read wallet info from: " + walletPath);

      final String userName = "user1";

      final Path connectionProfile = Paths.get("..", "..", "..", "..", "test-network", "organizations",
          "peerOrganizations", "seller.example.com", "connection-seller.yaml");

```

```
      // Set connection options on the gateway builder
      builder.identity(wallet, userName).networkConfig(connectionProfile).discovery(false);

      // Connect to gateway using application specified parameters
      try (Gateway gateway = builder.connect()) {

        // get the network and contract
        final Network network = gateway.getNetwork(channelName);
        final Contract contract = network.getContract(contractName);

        byte[] result;

        result = contract.evaluateTransaction("queryAllOrders");
        System.out.println("List of all orders:");
        System.out.println(new String(result));
        System.out.println("------------------------------------");
        // if (false) {
        contract.submitTransaction("createOrder", "1", "100", "2", "10", "2", "Karlsplatz 13, 1040 Wien",
            "2020-09-20");
        contract.submitTransaction("createOrder", "2", "123587", "5", "750", "4", "Ballhausplatz 2, 1010 Wien",
            "2020-12-01");
        contract.submitTransaction("createOrder", "3", "68754", "1", "1337", "2", "Michaelerkuppel, 1010 Wien",
            "2020-08-15");

        result = contract.evaluateTransaction("queryAllOrders");
        System.out.println("List of all orders:");
        System.out.println(new String(result));
        System.out.println("------------------------------------");
        // }
        System.out.println("Wait until order with id 2 is set to state CONFIRMED");
        result = contract.evaluateTransaction("queryOrder", "2");
        Order order = Order.deserialize(result);
        System.out.println(Order.deserialize(result));
        while (order.getState() != Order.State.CONFIRMED) {
          System.out.println("order 2 state is:" + order.getState());
          Thread.sleep(5000);
          result = contract.evaluateTransaction("queryOrder", "2");
          order = Order.deserialize(result);
        }

        contract.submitTransaction("shipOrder", "2", "1AXCAW311");
        System.out.println("shipped order 2");
        result = contract.evaluateTransaction("queryOrder", "2");
        System.out.println(Order.deserialize(result));
        System.out.println("------------------------------------");

      }
    } catch (GatewayException | IOException | TimeoutException |
```

```
        InterruptedException e) {
      e.printStackTrace();
      System.exit(-1);
    }
  }

}
```

AddToWallet.java (Freight Company)

```
package org.example;

import java.io.IOException;
import java.io.Reader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.InvalidKeyException;
import java.security.PrivateKey;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import java.util.stream.Stream;

import javax.naming.InvalidNameException;
import javax.naming.ldap.LdapName;

import org.hyperledger.fabric.gateway.Identities;
import org.hyperledger.fabric.gateway.Identity;
import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;

public class AddToWallet {

  private static X509Certificate readX509Certificate(final Path
      certificatePath)
      throws IOException, CertificateException {
    try (Reader certificateReader = Files.newBufferedReader(certificatePath,
        StandardCharsets.UTF_8)) {
      return Identities.readX509Certificate(certificateReader);
    }
  }

  private static PrivateKey getPrivateKey(final Path privateKeyPath) throws
      IOException, InvalidKeyException {
    try (Reader privateKeyReader = Files.newBufferedReader(privateKeyPath,
        StandardCharsets.UTF_8)) {
      return Identities.readPrivateKey(privateKeyReader);
    }
  }

  public static void main(String[] args) {
```

```
    try {
      // A wallet stores a collection of identities
      final Path walletPath = Paths.get(".", "wallet");
      final Wallet wallet = Wallets.newFileSystemWallet(walletPath);

      final Path credentialPath = Paths.get("..", "..", "..", "..", "test-
          network", "organizations",
          "peerOrganizations", "freight.example.com", "users", "User1@freight
              .example.com", "msp");
      System.out.println("credentialPath: " + credentialPath.toString());
      // final Path certificatePath = credentialPath.resolve(Paths.get("
          signcerts",
      // "User1@freight.example.com-cert.pem"));
      final Path certificatePath = credentialPath.resolve(Paths.get("
          signcerts", "cert.pem"));
      System.out.println("certificatePem: " + certificatePath.toString());

      Path privateKeyPath = null;
      try (Stream<Path> paths = Files.find(credentialPath.resolve(Paths.get("
          keystore")), Integer.MAX_VALUE,
          (path, attrs) -> attrs.isRegularFile() && path.toString().endsWith(
              "_sk"))) {
        privateKeyPath = paths.findAny().get();
      }

      // final Path privateKeyPath = credentialPath.resolve(Paths.get("
          keystore",
      // "priv_sk"));

      final X509Certificate certificate = readX509Certificate(certificatePath
          );

      final String identityLabel = new LdapName(certificate.
          getSubjectX500Principal().getName()).getRdns().stream()
          .filter(i -> i.getType().equalsIgnoreCase("CN")).findFirst().get().
              getValue().toString();

      final PrivateKey privateKey = getPrivateKey(privateKeyPath);
      final Identity identity = Identities.newX509Identity("FreightMSP",
          certificate, privateKey);

      wallet.put(identityLabel, identity);

      System.out.println("Write wallet info into " + walletPath.toString() +
          " successfully.");

    } catch (IOException | CertificateException | InvalidKeyException |
        InvalidNameException e) {
      System.err.println("Error adding to wallet");
      e.printStackTrace();
    }
  }

```

```
}
```

ClientApp.java (Freight Company)

```java
package org.example;

import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.concurrent.TimeoutException;

import org.hyperledger.fabric.gateway.Contract;
import org.hyperledger.fabric.gateway.Gateway;
import org.hyperledger.fabric.gateway.GatewayException;
import org.hyperledger.fabric.gateway.Network;
import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;

public class ClientApp {
  private static final String CONTRACT = "CONTRACT_NAME";
  private static final String CHANNEL = "CHANNEL_NAME";

  public static void main(final String[] args) {
    final Gateway.Builder builder = Gateway.createBuilder();

    String contractName = "trade-finance";
    String channelName = "mychannel";
    // get the name of the contract, in case it is overridden
    final Map<String, String> envvar = System.getenv();
    if (envvar.containsKey(CONTRACT)) {
      contractName = envvar.get(CONTRACT);
    }
    if (envvar.containsKey(CHANNEL)) {
      channelName = envvar.get(CHANNEL);
    }

    try {
      // A wallet stores a collection of identities
      final Path walletPath = Paths.get(".", "wallet");
      final Wallet wallet = Wallets.newFileSystemWallet(walletPath);
      System.out.println("Read wallet info from: " + walletPath);

      final String userName = "user1";

      final Path connectionProfile = Paths.get("..", "..", "..", "..", "test-
          network", "organizations",
          "peerOrganizations", "freight.example.com", "connection-freight.
              yaml");

      // Set connection options on the gateway builder
```

```
      builder.identity(wallet, userName).networkConfig(connectionProfile).
          discovery(false);

      // Connect to gateway using application specified parameters
      try (Gateway gateway = builder.connect()) {

        // get the network and contract
        final Network network = gateway.getNetwork(channelName);
        final Contract contract = network.getContract(contractName);

        byte[] result;

        result = contract.evaluateTransaction("queryAllOrders");
        System.out.println("List of all orders:");
        System.out.println(new String(result));
        System.out.println("------------------------------------");

        System.out.println("Wait until order with id 2 is set to state
            SHIPPED");
        result = contract.evaluateTransaction("queryOrder", "2");
        Order order = Order.deserialize(result);
        System.out.println(Order.deserialize(result));
        while (order.getState() != Order.State.SHIPPED) {
          System.out.println("order 2 state is:" + order.getState());
          Thread.sleep(5000);
          result = contract.evaluateTransaction("queryOrder", "2");
          order = Order.deserialize(result);
        }

        contract.submitTransaction("signArrival", "2");
        System.out.println("Signed arrival of order 2");
        result = contract.evaluateTransaction("queryOrder", "2");
        System.out.println(new String(result));
        System.out.println("------------------------------------");
      }
    } catch (GatewayException | IOException | TimeoutException |
        InterruptedException e) {
      e.printStackTrace();
      System.exit(-1);
    }
  }

}
```

AddToWallet.java (Buyer)

```
package org.example;

import java.io.IOException;
import java.io.Reader;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
```

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.InvalidKeyException;
import java.security.PrivateKey;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import java.util.stream.Stream;

import javax.naming.InvalidNameException;
import javax.naming.ldap.LdapName;

import org.hyperledger.fabric.gateway.Identities;
import org.hyperledger.fabric.gateway.Identity;
import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;

public class AddToWallet {

  private static X509Certificate readX509Certificate(final Path certificatePath)
      throws IOException, CertificateException {
    try (Reader certificateReader = Files.newBufferedReader(certificatePath, StandardCharsets.UTF_8)) {
      return Identities.readX509Certificate(certificateReader);
    }
  }

  private static PrivateKey getPrivateKey(final Path privateKeyPath) throws IOException, InvalidKeyException {
    try (Reader privateKeyReader = Files.newBufferedReader(privateKeyPath, StandardCharsets.UTF_8)) {
      return Identities.readPrivateKey(privateKeyReader);
    }
  }

  public static void main(String[] args) {
    try {
      // A wallet stores a collection of identities
      final Path walletPath = Paths.get(".", "wallet");
      final Wallet wallet = Wallets.newFileSystemWallet(walletPath);

      final Path credentialPath = Paths.get("..", "..", "..", "..", "test-network", "organizations",
          "peerOrganizations", "buyer.example.com", "users", "User1@buyer.example.com", "msp");
      System.out.println("credentialPath: " + credentialPath.toString());
      // final Path certificatePath = credentialPath.resolve(Paths.get("signcerts",
      // "User1@buyer.example.com-cert.pem"));
      final Path certificatePath = credentialPath.resolve(Paths.get("signcerts", "cert.pem"));
      System.out.println("certificatePem: " + certificatePath.toString());

```

```java
      Path privateKeyPath = null;
      try (Stream<Path> paths = Files.find(credentialPath.resolve(Paths.get("keystore")), Integer.MAX_VALUE,
          (path, attrs) -> attrs.isRegularFile() && path.toString().endsWith("_sk"))) {
        privateKeyPath = paths.findAny().get();
      }

      // final Path privateKeyPath = credentialPath.resolve(Paths.get("keystore",
      // "priv_sk"));

      final X509Certificate certificate = readX509Certificate(certificatePath);

      final String identityLabel = new LdapName(certificate.getSubjectX500Principal().getName()).getRdns().stream()
          .filter(i -> i.getType().equalsIgnoreCase("CN")).findFirst().get().getValue().toString();

      final PrivateKey privateKey = getPrivateKey(privateKeyPath);
      final Identity identity = Identities.newX509Identity("BuyerMSP", certificate, privateKey);

      wallet.put(identityLabel, identity);

      System.out.println("Write wallet info into " + walletPath.toString() + " successfully.");

    } catch (IOException | CertificateException | InvalidKeyException | InvalidNameException e) {
      System.err.println("Error adding to wallet");
      e.printStackTrace();
    }
  }

}
```

ClientApp.java (Buyer)

```java
package org.example;

import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Map;
import java.util.concurrent.TimeoutException;

import org.hyperledger.fabric.gateway.Contract;
import org.hyperledger.fabric.gateway.Gateway;
import org.hyperledger.fabric.gateway.GatewayException;
import org.hyperledger.fabric.gateway.Network;
```

```java
import org.hyperledger.fabric.gateway.Wallet;
import org.hyperledger.fabric.gateway.Wallets;

public class ClientApp {
  private static final String CONTRACT = "CONTRACT_NAME";
  private static final String CHANNEL = "CHANNEL_NAME";

  public static void main(final String[] args) {
    final Gateway.Builder builder = Gateway.createBuilder();

    String contractName = "trade-finance";
    String channelName = "mychannel";
    // get the name of the contract, in case it is overridden
    final Map<String, String> envvar = System.getenv();
    if (envvar.containsKey(CONTRACT)) {
      contractName = envvar.get(CONTRACT);
    }
    if (envvar.containsKey(CHANNEL)) {
      channelName = envvar.get(CHANNEL);
    }

    try {
      // A wallet stores a collection of identities
      final Path walletPath = Paths.get(".", "wallet");
      final Wallet wallet = Wallets.newFileSystemWallet(walletPath);
      System.out.println("Read wallet info from: " + walletPath);

      final String userName = "user1";

      final Path connectionProfile = Paths.get("..", "..", "..", "..", "test-
          network", "organizations",
          "peerOrganizations", "buyer.example.com", "connection-buyer.yaml");

      // Set connection options on the gateway builder
      builder.identity(wallet, userName).networkConfig(connectionProfile).
          discovery(false);

      // Connect to gateway using application specified parameters
      try (Gateway gateway = builder.connect()) {

        // get the network and contract
        final Network network = gateway.getNetwork(channelName);
        final Contract contract = network.getContract(contractName);

        byte[] result;

        result = contract.evaluateTransaction("queryAllOrders");
        System.out.println("List of all orders:");
        System.out.println(new String(result));
        System.out.println("------------------------------------");

        result = contract.evaluateTransaction("queryAllOrders");
        System.out.println("Result of 1st transaction:");
```

```
        System.out.println(new String(result));
        System.out.println("------------------------------------");

        contract.submitTransaction("cancelOrder", "1");
        System.out.println("Cancelled order 1");
        result = contract.evaluateTransaction("queryOrder", "1");
        System.out.println(new String(result));
        System.out.println("------------------------------------");

        contract.submitTransaction("confirmOrder", "2");
        System.out.println("Confirmed order 2");
        result = contract.evaluateTransaction("queryOrder", "2");
        System.out.println(new String(result));
        System.out.println("------------------------------------");

        System.out.println("Check if delivery date of order 3 has passed");
        result = contract.submitTransaction("deliveryDatePassed", "3");
        System.out.println(new String(result));
        result = contract.evaluateTransaction("queryOrder", "3");
        System.out.println(new String(result));
        System.out.println("------------------------------------");

        System.out.println("Wait until order with id 2 is set to state
            SHIPPED");
        result = contract.evaluateTransaction("queryOrder", "2");
        Order order = Order.deserialize(result);
        System.out.println(Order.deserialize(result));
        while (order.getState() != Order.State.SHIPPED) {
          System.out.println("order 2 state is:" + order.getState());
          Thread.sleep(5000);
          result = contract.evaluateTransaction("queryOrder", "2");
          order = Order.deserialize(result);
        }

        contract.submitTransaction("signArrival", "2");
        System.out.println("Signed arrival of order 2");
        result = contract.evaluateTransaction("queryOrder", "2");
        System.out.println(new String(result));
        System.out.println("------------------------------------");
      }
    } catch (GatewayException | IOException | TimeoutException |
        InterruptedException e) {
      e.printStackTrace();
      System.exit(-1);
    }
  }

}
```

Appendix C - Corda Prototype Code

OrderState.java

```java
package com.template.states;

import com.template.contracts.TradeFinanceContract;
import net.corda.core.contracts.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.serialization.ConstructorForDeserialization;
import net.corda.core.serialization.CordaSerializable;
import org.jetbrains.annotations.NotNull;

import java.time.Instant;
import java.util.Arrays;
import java.util.Currency;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;
import java.util.stream.Stream;

// *********
// * State *
// *********
@BelongsToContract(TradeFinanceContract.class)
public class OrderState implements LinearState {

    @NotNull
    @Override
    public UniqueIdentifier getLinearId() {
        return this.orderId;
    }

    @CordaSerializable
    public enum State {
        CREATED,
        CONFIRMED,
        SHIPPED,
```

```
        DELIVERED,
        CANCELLED,
        PASSED
    }

    //private variables
    private Party seller;
    private State orderState;
    private Party buyer;
    private UniqueIdentifier orderId;
    private int productId;
    private double quantity;
    private Amount<Currency> price;
    private Amount<Currency> shippingCosts;
    private String shippingAddress;
    private Instant latestDeliveryDate;
    private Party freightCompany;
    private String trackingCode;
    private boolean buyerSigned;
    private boolean freightSigned;

    /* Constructor of your Corda state */
    @ConstructorForDeserialization
    public OrderState(Party seller, State orderState, Party buyer, UniqueIdentifier orderId, int productId, double quantity, Amount<Currency> price, Amount<Currency> shippingCosts, String shippingAddress, Instant latestDeliveryDate, Party freightCompany, String trackingCode, boolean buyerSigned, boolean freightSigned) {
        this.seller = seller;
        this.orderState = orderState;
        this.buyer = buyer;
        this.orderId = orderId;
        this.productId = productId;
        this.quantity = quantity;
        this.price = price;
        this.shippingCosts = shippingCosts;
        this.shippingAddress = shippingAddress;
        this.latestDeliveryDate = latestDeliveryDate;
        this.freightCompany = freightCompany;
        this.trackingCode = trackingCode;
        this.buyerSigned = buyerSigned;
        this.freightSigned = freightSigned;
    }

    public OrderState(Party seller, Party buyer, String orderId, int productId, double quantity, Amount<Currency> price, Amount<Currency> shippingCosts, String shippingAddress, Instant latestDeliveryDate) {
        this.seller = seller;
        this.buyer = buyer;
        this.orderId = new UniqueIdentifier(orderId);
        this.productId = productId;
        this.quantity = quantity;
        this.price = price;
```

```
        this.shippingCosts = shippingCosts;
        this.shippingAddress = shippingAddress;
        this.latestDeliveryDate = latestDeliveryDate;
        this.orderState = State.CREATED;
    }

    //getters
    public Party getSeller() {
        return seller;
    }

    public State getOrderState() {
        return orderState;
    }

    public void setOrderState(State orderState) {
        this.orderState = orderState;
    }

    public Party getBuyer() {
        return buyer;
    }

    public UniqueIdentifier getOrderId() {
        return orderId;
    }

    public int getProductId() {
        return productId;
    }

    public double getQuantity() {
        return quantity;
    }

    public Amount<Currency> getPrice() {
        return price;
    }

    public String getShippingAddress() {
        return shippingAddress;
    }

    public Instant getLatestDeliveryDate() {
        return latestDeliveryDate;
    }

    public Party getFreightCompany() {
        return freightCompany;
    }

    public void setFreightCompany(Party freightCompany) {
        this.freightCompany = freightCompany;
```

```java
    }

    public Amount<Currency> getShippingCosts() {
        return shippingCosts;
    }

    public String getTrackingCode() {
        return trackingCode;
    }

    public void setTrackingCode(String trackingCode) {
        this.trackingCode = trackingCode;
    }

    public boolean isBuyerSigned() {
        return buyerSigned;
    }

    public void setBuyerSigned(boolean buyerSigned) {
        this.buyerSigned = buyerSigned;
    }

    public boolean isFreightSigned() {
        return freightSigned;
    }

    public void setFreightSigned(boolean freightSigned) {
        this.freightSigned = freightSigned;
    }

    /* This method will indicate who are the participants and required
        signers when
     * this state is used in a transaction. */
    @NotNull
    @Override
    public List<AbstractParty> getParticipants() {
        return Stream.of(this.seller, this.buyer, this.freightCompany).filter
            (Objects::nonNull).collect(Collectors.toList());
    }

    public OrderState copy() {
        return new OrderState(this.seller, this.orderState, this.buyer, this.
            orderId, this.productId, this.quantity, this.price, this.
            shippingCosts, this.shippingAddress, this.latestDeliveryDate,
            this.freightCompany, this.trackingCode, this.buyerSigned, this.
            freightSigned);
    }
}
```

TradeFinanceContract.java

```java
package com.template.contracts;
```

```java

import com.template.states.OrderState;
import net.corda.core.contracts.CommandData;
import net.corda.core.contracts.CommandWithParties;
import net.corda.core.contracts.Contract;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.transactions.LedgerTransaction;

import java.time.Instant;
import java.util.Arrays;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import static net.corda.core.contracts.ContractsDSL.requireSingleCommand;
import static net.corda.core.contracts.ContractsDSL.requireThat;

// ************
// * Contract *
// ************
public class TradeFinanceContract implements Contract {
    // This is used to identify our contract when building a transaction.
    public static final String ID = "com.template.contracts.
        TradeFinanceContract";

    // A transaction is valid if the verify() function of the contract of all
         the transaction's input and output states
    // does not throw an exception.
    @Override
    public void verify(LedgerTransaction tx) {

        /* We can use the requireSingleCommand function to extract command
            data from a transaction.
         * However, it is possible to have multiple commands in a single
             transaction.*/
        final CommandWithParties<Commands> command = requireSingleCommand(tx.
            getCommands(), Commands.class);

        //Retrieve the input and output states of the transaction
        OrderState input = tx.getInputs().size() != 0 ? tx.inputsOfType(
            OrderState.class).get(0) : null;
        OrderState output = tx.outputsOfType(OrderState.class).get(0);
        if (command.getValue() instanceof Commands.Create) {
            //Using Corda DSL function requireThat to replicate conditions-
                checks
            requireThat(require -> {
                require.using("No inputs should be consumed when adding a new
                     order.", tx.getInputStates().size() == 0);
                require.using("Only the seller is allowed to start this flow.
                    ", command.getValue().getInitiator().getOwningKey().
                    equals(output.getSeller().getOwningKey()));
                require.using("The price must be greater or equal to the
                    shipping costs.", output.getPrice().compareTo(output.
```

```
                    getShippingCosts()) >= 0);
                return null;
            });
        } else if (command.getValue() instanceof Commands.Cancel) {
            //Using Corda DSL function requireThat to replicate conditions-
                checks
            requireThat(require -> {
                require.using("Exactly one input should be consumed when
                    cancelling an order.", tx.getInputStates().size() == 1);
                require.using("Function cannot be called at this state: " +
                    input.getOrderState(), Stream.of(input.getOrderState()).
                    anyMatch(Arrays.asList(OrderState.State.CREATED,
                    OrderState.State.CONFIRMED)::contains));
                require.using("Only the the seller or the buyer are allowed
                    to start this flow.", Arrays.asList(output.getSeller().
                    getOwningKey(), output.getBuyer().getOwningKey()).
                    contains(command.getValue().getInitiator().getOwningKey()
                    ));
                return null;
            });
        } else if (command.getValue() instanceof Commands.CheckDate) {
            //Using Corda DSL function requireThat to replicate conditions-
                checks
            requireThat(require -> {
                require.using("Exactly one input should be consumed when
                    checking the order delivery date.", tx.getInputStates().
                    size() == 1);
                require.using("Function cannot be called at this state: " +
                    input.getOrderState(), input.getOrderState() !=
                    OrderState.State.DELIVERED);
                require.using("Delivery date did not pass yet.", Instant.now
                    ().isAfter(input.getLatestDeliveryDate()));
                require.using("Refund not possible as the freight company
                    already signed the arrival.", !input.isFreightSigned());
                return null;
            });
        } else if (command.getValue() instanceof Commands.Confirm) {
            //Using Corda DSL function requireThat to replicate conditions-
                checks
            requireThat(require -> {
                require.using("Exactly one input should be consumed when
                    confirming an order.", tx.getInputStates().size() == 1);
                require.using("Function cannot be called at this state: " +
                    input.getOrderState(), input.getOrderState() ==
                    OrderState.State.CREATED);
                require.using("Only the buyer is allowed to start this flow."
                    , command.getValue().getInitiator().getOwningKey().equals
                    (output.getBuyer().getOwningKey()));
                return null;
            });
        } else if (command.getValue() instanceof Commands.Ship) {
            //Using Corda DSL function requireThat to replicate conditions-
                checks
```

```
            requireThat(require -> {
                require.using("Exactly one input should be consumed when
                    shipping an order.", tx.getInputStates().size() == 1);
                require.using("Function cannot be called at this state: " +
                    input.getOrderState(), input.getOrderState() ==
                    OrderState.State.CONFIRMED);
                require.using("Only the seller is allowed to start this flow.
                    ", command.getValue().getInitiator().getOwningKey().
                    equals(output.getSeller().getOwningKey()));
                return null;
            });
        } else if (command.getValue() instanceof Commands.Sign) {
            //Using Corda DSL function requireThat to replicate conditions-
                checks
            requireThat(require -> {
                require.using("Exactly one input should be consumed when
                    signing an order.", tx.getInputStates().size() == 1);
                require.using("Function cannot be called at this state: " +
                    input.getOrderState(), input.getOrderState() ==
                    OrderState.State.SHIPPED);
                require.using("Only the buyer and freight company are allowed
                     to start this flow.", Arrays.asList(output.getBuyer().
                    getOwningKey(), output.getFreightCompany().getOwningKey()
                    ).contains(command.getValue().getInitiator().getOwningKey
                    ()));
                return null;
            });
        }
    }

    // Used to indicate the transaction's intent.
    public abstract static class Commands implements CommandData {
        private Party initiator;

        public Commands(Party initiator) {
            this.initiator = initiator;
        }

        public Party getInitiator() {
            return initiator;
        }

        public static class Create extends Commands {
            public Create(Party initiator) {
                super(initiator);
            }
        }

        public static class CheckDate extends Commands {
            public CheckDate(Party initiator) {
                super(initiator);
            }
        }
```

```

        public static class Cancel extends Commands {
            public Cancel(Party initiator) {
                super(initiator);
            }
        }

        public static class Confirm extends Commands {
            public Confirm(Party initiator) {
                super(initiator);
            }
        }

        public static class Ship extends Commands {
            public Ship(Party initiator) {
                super(initiator);
            }
        }

        public static class Sign extends Commands {
            public Sign(Party initiator) {
                super(initiator);
            }
        }
    }
}
```

DataUtils.java

```
package com.template.utils;

import com.template.states.OrderState;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.node.ServiceHub;
import net.corda.core.node.services.Vault;
import net.corda.core.node.services.vault.QueryCriteria;

import java.util.Collections;
import java.util.List;

public class DataUtils {

    public static StateAndRef<OrderState> getOrder(ServiceHub serviceHub,
        String orderId) {
        //Check if an order with this ID already exists
        QueryCriteria.LinearStateQueryCriteria queryCriteria = new
            QueryCriteria.LinearStateQueryCriteria()
                .withExternalId(Collections.singletonList(orderId)).
                    withStatus(Vault.StateStatus.UNCONSUMED);
        List<StateAndRef<OrderState>> results = serviceHub.getVaultService().
            queryBy(OrderState.class, queryCriteria).getStates();
        if (results.isEmpty()) {
```

```
            throw new IllegalArgumentException("An order with ID " + orderId
                + " does not exist or is already consumed.");
        }
        return results.get(0);
    }

}
```

CancelOrder.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import com.template.contracts.TradeFinanceContract;
import com.template.states.OrderState;
import com.template.utils.DataUtils;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.flows.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.util.List;
import java.util.stream.Collectors;

// ******************
// * Initiator flow *
// ******************
@InitiatingFlow
@StartableByRPC
public class CancelOrder extends FlowLogic<String> {
    private final ProgressTracker progressTracker = tracker();

    private static final ProgressTracker.Step GENERATING_TRANSACTION = new
        ProgressTracker.Step("Generating a CancelOrder transaction");
    private static final ProgressTracker.Step SIGNING_TRANSACTION = new
        ProgressTracker.Step("Signing transaction with our private key.");
    private static final ProgressTracker.Step COLLECTING_SIGNATURES = new
        ProgressTracker.Step("Collecting the signatures of the other parties.
        ");
    private static final ProgressTracker.Step FINALISING_TRANSACTION = new
        ProgressTracker.Step("Recording transaction") {
        @Override
        public ProgressTracker childProgressTracker() {
            return FinalityFlow.tracker();
        }
    };

    private static ProgressTracker tracker() {
        return new ProgressTracker(
```

```
                GENERATING_TRANSACTION,
                SIGNING_TRANSACTION,
                COLLECTING_SIGNATURES,
                FINALISING_TRANSACTION
        );
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    //private variables
    private final String orderId;

    //public constructor
    public CancelOrder(String orderId) {
        this.orderId = orderId;
    }

    @Suspendable
    @Override
    public String call() throws FlowException {
        // Step 1. Get the order data from the vault
        StateAndRef<OrderState> inputOrderStateAndRef = DataUtils.getOrder(
            getServiceHub(), this.orderId);
        OrderState inputOrderState = inputOrderStateAndRef.getState().getData
            ();

        // Generate State for transfer
        // Step 2. Get a reference to the notary service on our network and
            our key pair.
        final Party notary = getServiceHub().getNetworkMapCache().
            getNotaryIdentities().get(0);

        // Step 3. Compose the State that carries the order data
        progressTracker.setCurrentStep(GENERATING_TRANSACTION);
        OrderState outputOrderState = inputOrderState.copy();
        outputOrderState.setOrderState(OrderState.State.CANCELLED);

        // Step 4. Create a new TransactionBuilder object.
        final TransactionBuilder builder = new TransactionBuilder(notary);

        // Step 5. Add the order as an output state, as well as a command to
            the transaction builder.
        builder.addInputState(inputOrderStateAndRef);
        builder.addOutputState(outputOrderState);
        builder.addCommand(new TradeFinanceContract.Commands.Cancel(
            getOurIdentity()), outputOrderState.getParticipants().stream().
            map(AbstractParty::getOwningKey).collect(Collectors.toList()));

        // Step 6. Verify and sign it with our KeyPair.
        progressTracker.setCurrentStep(SIGNING_TRANSACTION);
```

```
        builder.verify(getServiceHub());
        final SignedTransaction ptx = getServiceHub().signInitialTransaction(
            builder);

        // Step 7. Collect the other party's signature using the
            SignTransactionFlow.
        progressTracker.setCurrentStep(COLLECTING_SIGNATURES);
        List<Party> otherParties = outputOrderState.getParticipants().stream
            ().map(el -> (Party) el).collect(Collectors.toList());
        otherParties.remove(getOurIdentity());
        List<FlowSession> sessions = otherParties.stream().map(this::
            initiateFlow).collect(Collectors.toList());

        SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx,
            sessions));

        // Step 8. Assuming no exceptions, we can now finalise the
            transaction
        progressTracker.setCurrentStep(FINALISING_TRANSACTION);
        subFlow(new FinalityFlow(stx, sessions));

        return "Cancel flow for order with ID '" + this.orderId + "' of buyer
            '" + outputOrderState.getBuyer().getName() + "' executed.";
    }
}
```

CancelOrderResponder.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.flows.*;
import net.corda.core.transactions.SignedTransaction;

// ******************
// * Responder flow *
// ******************
@InitiatedBy(CancelOrder.class)
public class CancelOrderResponder extends FlowLogic<Void> {

    //private variable
    private FlowSession counterpartySession;

    //Constructor
    public CancelOrderResponder(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
```

```
        SignedTransaction signedTransaction = subFlow(new SignTransactionFlow
            (counterpartySession) {
            @Suspendable
            @Override
            protected void checkTransaction(SignedTransaction stx) throws
                FlowException {
                /*
                 * SignTransactionFlow will automatically verify the
                   transaction and its signatures before signing it.
                 * However, just because a transaction is contractually valid
                    doesn't mean we necessarily want to sign.
                 * What if we don't want to deal with the counterparty in
                   question, or the value is too high,
                 * or we're not happy with the transaction's structure?
                   checkTransaction
                 * allows us to define these additional checks. If any of
                   these conditions are not met,
                 * we will not sign the transaction - even if the transaction
                    and its signatures are contractually valid.
                 * ----------
                 * For this cordapp, we will not implement any additional
                   checks.
                 * */
            }
        });
        //Stored the transaction into data base.
        subFlow(new ReceiveFinalityFlow(counterpartySession,
            signedTransaction.getId()));
        return null;
    }
}
```

CheckDeliveryDate.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import com.template.contracts.TradeFinanceContract;
import com.template.states.OrderState;
import com.template.utils.DataUtils;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.flows.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.util.List;
import java.util.stream.Collectors;

// *****************
```

```java
// * Initiator flow *
// ******************
@InitiatingFlow
@StartableByRPC
public class CheckDeliveryDate extends FlowLogic<String> {
    private final ProgressTracker progressTracker = tracker();

    private static final ProgressTracker.Step GENERATING_TRANSACTION = new
        ProgressTracker.Step("Generating a CheckDeliveryDate transaction");
    private static final ProgressTracker.Step SIGNING_TRANSACTION = new
        ProgressTracker.Step("Signing transaction with our private key.");
    private static final ProgressTracker.Step COLLECTING_SIGNATURES = new
        ProgressTracker.Step("Collecting the signatures of the other parties.
        ");
    private static final ProgressTracker.Step FINALISING_TRANSACTION = new
        ProgressTracker.Step("Recording transaction") {
        @Override
        public ProgressTracker childProgressTracker() {
            return FinalityFlow.tracker();
        }
    };

    private static ProgressTracker tracker() {
        return new ProgressTracker(
                GENERATING_TRANSACTION,
                SIGNING_TRANSACTION,
                COLLECTING_SIGNATURES,
                FINALISING_TRANSACTION
        );
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    //private variables
    private final String orderId;

    //public constructor
    public CheckDeliveryDate(String orderId) {
        this.orderId = orderId;
    }

    @Suspendable
    @Override
    public String call() throws FlowException {
        // Step 1. Check if an order with this ID already exists
        StateAndRef<OrderState> inputOrderStateAndRef = DataUtils.getOrder(
            getServiceHub(), this.orderId);
        OrderState inputOrderState = inputOrderStateAndRef.getState().getData
            ();

```

```java
        // Generate State for transfer
        // Step 2. Get a reference to the notary service on our network and
            our key pair.
        final Party notary = getServiceHub().getNetworkMapCache().
            getNotaryIdentities().get(0);

        // Step 3. Compose the State that carries the order data
        progressTracker.setCurrentStep(GENERATING_TRANSACTION);
        OrderState outputOrderState = inputOrderState.copy();
        outputOrderState.setOrderState(OrderState.State.PASSED);

        // Step 4. Create a new TransactionBuilder object.
        final TransactionBuilder builder = new TransactionBuilder(notary);

        // Step 5. Add the order as an output state, as well as a command to
            the transaction builder.
        builder.addInputState(inputOrderStateAndRef);
        builder.addOutputState(outputOrderState);
        builder.addCommand(new TradeFinanceContract.Commands.CheckDate(
            getOurIdentity()), outputOrderState.getParticipants().stream().
            map(AbstractParty::getOwningKey).collect(Collectors.toList()));

        // Step 6. Verify and sign it with our KeyPair.
        progressTracker.setCurrentStep(SIGNING_TRANSACTION);
        builder.verify(getServiceHub());
        final SignedTransaction ptx = getServiceHub().signInitialTransaction(
            builder);

        // Step 7. Collect the other party’s signature using the
            SignTransactionFlow.
        progressTracker.setCurrentStep(COLLECTING_SIGNATURES);
        List<Party> otherParties = outputOrderState.getParticipants().stream
            ().map(el -> (Party) el).collect(Collectors.toList());
        otherParties.remove(getOurIdentity());
        List<FlowSession> sessions = otherParties.stream().map(this::
            initiateFlow).collect(Collectors.toList());

        SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx,
            sessions));

        // Step 8. Assuming no exceptions, we can now finalise the
            transaction
        progressTracker.setCurrentStep(FINALISING_TRANSACTION);
        subFlow(new FinalityFlow(stx, sessions));

        return "Check delivery date flow for order with ID ’" + this.orderId
            + "’ of buyer ’" + outputOrderState.getBuyer().getName() + "’
            executed.";
    }
}
```

CheckDeliveryDateResponder.java

```java
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.flows.*;
import net.corda.core.transactions.SignedTransaction;

// ******************
// * Responder flow *
// ******************
@InitiatedBy(CheckDeliveryDate.class)
public class CheckDeliveryDateResponder extends FlowLogic<Void> {

    //private variable
    private FlowSession counterpartySession;

    //Constructor
    public CheckDeliveryDateResponder(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
        SignedTransaction signedTransaction = subFlow(new SignTransactionFlow
            (counterpartySession) {
            @Suspendable
            @Override
            protected void checkTransaction(SignedTransaction stx) throws
                FlowException {
                /*
                 * SignTransactionFlow will automatically verify the
                   transaction and its signatures before signing it.
                 * However, just because a transaction is contractually valid
                    doesn't mean we necessarily want to sign.
                 * What if we don't want to deal with the counterparty in
                   question, or the value is too high,
                 * or we're not happy with the transaction's structure?
                   checkTransaction
                 * allows us to define these additional checks. If any of
                   these conditions are not met,
                 * we will not sign the transaction - even if the transaction
                    and its signatures are contractually valid.
                 * ----------
                 * For this cordapp, we will not implement any additional
                   checks.
                 * */
            }
        });
        //Stored the transaction into data base.
        subFlow(new ReceiveFinalityFlow(counterpartySession,
```

```
            signedTransaction.getId()));
        return null;
    }
}
```

ConfirmOrder.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import com.template.contracts.TradeFinanceContract;
import com.template.states.OrderState;
import com.template.utils.DataUtils;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.flows.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.util.List;
import java.util.stream.Collectors;

// ******************
// * Initiator flow *
// ******************
@InitiatingFlow
@StartableByRPC
public class ConfirmOrder extends FlowLogic<String> {
    private final ProgressTracker progressTracker = tracker();

    private static final ProgressTracker.Step GENERATING_TRANSACTION = new
        ProgressTracker.Step("Generating a ConfirmOrder transaction");
    private static final ProgressTracker.Step SIGNING_TRANSACTION = new
        ProgressTracker.Step("Signing transaction with our private key.");
    private static final ProgressTracker.Step COLLECTING_SIGNATURES = new
        ProgressTracker.Step("Collecting the signatures of the other parties.
        ");
    private static final ProgressTracker.Step FINALISING_TRANSACTION = new
        ProgressTracker.Step("Recording transaction") {
        @Override
        public ProgressTracker childProgressTracker() {
            return FinalityFlow.tracker();
        }
    };

    private static ProgressTracker tracker() {
        return new ProgressTracker(
                GENERATING_TRANSACTION,
                SIGNING_TRANSACTION,
                COLLECTING_SIGNATURES,
```

```java
                FINALISING_TRANSACTION
        );
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    //private variables
    private final String orderId;

    //public constructor
    public ConfirmOrder(String orderId) {
        this.orderId = orderId;
    }

    @Suspendable
    @Override
    public String call() throws FlowException {
        // Step 1. Check if an order with this ID already exists
        StateAndRef<OrderState> inputOrderStateAndRef = DataUtils.getOrder(getServiceHub(), this.orderId);
        OrderState inputOrderState = inputOrderStateAndRef.getState().getData();

        // Generate State for transfer
        // Step 2. Get a reference to the notary service on our network and our key pair.
        final Party notary = getServiceHub().getNetworkMapCache().getNotaryIdentities().get(0);

        // Step 3. Compose the State that carries the order data
        progressTracker.setCurrentStep(GENERATING_TRANSACTION);
        OrderState outputOrderState = inputOrderState.copy();
        outputOrderState.setOrderState(OrderState.State.CONFIRMED);

        // Step 4. Create a new TransactionBuilder object.
        final TransactionBuilder builder = new TransactionBuilder(notary);

        // Step 5. Add the order as an output state, as well as a command to the transaction builder.
        builder.addInputState(inputOrderStateAndRef);
        builder.addOutputState(outputOrderState);
        builder.addCommand(new TradeFinanceContract.Commands.Confirm(getOurIdentity(), outputOrderState.getParticipants().stream().map(AbstractParty::getOwningKey).collect(Collectors.toList()));

        // Step 6. Verify and sign it with our KeyPair.
        progressTracker.setCurrentStep(SIGNING_TRANSACTION);
        builder.verify(getServiceHub());
        final SignedTransaction ptx = getServiceHub().signInitialTransaction(builder);
```

```java

        // Step 7. Collect the other party's signature using the SignTransactionFlow.
        progressTracker.setCurrentStep(COLLECTING_SIGNATURES);
        List<Party> otherParties = outputOrderState.getParticipants().stream().map(el -> (Party) el).collect(Collectors.toList());
        otherParties.remove(getOurIdentity());
        List<FlowSession> sessions = otherParties.stream().map(this::initiateFlow).collect(Collectors.toList());

        SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx, sessions));

        // Step 8. Assuming no exceptions, we can now finalise the transaction
        progressTracker.setCurrentStep(FINALISING_TRANSACTION);
        subFlow(new FinalityFlow(stx, sessions));

        return "Confirm order flow for order with ID '" + this.orderId + "' of buyer '" + outputOrderState.getBuyer().getName() + "' executed.";
    }
}
```

ConfirmOrderResponder.java

```java
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.flows.*;
import net.corda.core.transactions.SignedTransaction;

// ******************
// * Responder flow *
// ******************
@InitiatedBy(ConfirmOrder.class)
public class ConfirmOrderResponder extends FlowLogic<Void> {

    //private variable
    private FlowSession counterpartySession;

    //Constructor
    public ConfirmOrderResponder(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
        SignedTransaction signedTransaction = subFlow(new SignTransactionFlow(counterpartySession) {
            @Suspendable
```

```
            @Override
            protected void checkTransaction(SignedTransaction stx) throws
                FlowException {
                /*
                 * SignTransactionFlow will automatically verify the
                   transaction and its signatures before signing it.
                 * However, just because a transaction is contractually valid
                    doesn't mean we necessarily want to sign.
                 * What if we don't want to deal with the counterparty in
                   question, or the value is too high,
                 * or we're not happy with the transaction's structure?
                   checkTransaction
                 * allows us to define these additional checks. If any of
                   these conditions are not met,
                 * we will not sign the transaction - even if the transaction
                    and its signatures are contractually valid.
                 * ----------
                 * For this cordapp, we will not implement any additional
                   checks.
                 * */
            }
        });
        //Stored the transaction into data base.
        subFlow(new ReceiveFinalityFlow(counterpartySession,
            signedTransaction.getId()));
        return null;
    }
}
```

CreateOrder.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import com.template.contracts.TradeFinanceContract;
import com.template.states.OrderState;
import net.corda.core.contracts.Amount;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.flows.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.node.services.Vault;
import net.corda.core.node.services.vault.QueryCriteria;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.time.Instant;
import java.time.LocalDate;
import java.time.ZoneId;
import java.util.*;
import java.util.stream.Collectors;
```

```

// ******************
// * Initiator flow *
// ******************
@InitiatingFlow
@StartableByRPC
public class CreateOrder extends FlowLogic<String> {
    private final ProgressTracker progressTracker = tracker();

    private static final ProgressTracker.Step GENERATING_TRANSACTION = new
        ProgressTracker.Step("Generating a CreateOrder transaction");
    private static final ProgressTracker.Step SIGNING_TRANSACTION = new
        ProgressTracker.Step("Signing transaction with our private key.");
    private static final ProgressTracker.Step COLLECTING_SIGNATURES = new
        ProgressTracker.Step("Collecting the signatures of the other parties.
        ");
    private static final ProgressTracker.Step FINALISING_TRANSACTION = new
        ProgressTracker.Step("Recording transaction") {
        @Override
        public ProgressTracker childProgressTracker() {
            return FinalityFlow.tracker();
        }
    };

    private static ProgressTracker tracker() {
        return new ProgressTracker(
                GENERATING_TRANSACTION,
                SIGNING_TRANSACTION,
                COLLECTING_SIGNATURES,
                FINALISING_TRANSACTION
        );
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    //private variables
    private Party seller;
    private String buyer;
    private String orderId;
    private int productId;
    private double quantity;
    private Amount<Currency> price;
    private Amount<Currency> shippingCosts;
    private String shippingAddress;
    private Instant latestDeliveryDate;

    //public constructor
    public CreateOrder(String buyer, String orderId, int productId, double
        quantity, String price, String shippingCosts, String shippingAddress,
         String latestDeliveryDate) {
```

```
        this.buyer = buyer;
        this.orderId = orderId;
        this.productId = productId;
        this.quantity = quantity;
        this.price = Amount.parseCurrency(price);
        this.shippingCosts = Amount.parseCurrency(shippingCosts);
        this.shippingAddress = shippingAddress;
        this.latestDeliveryDate = LocalDate.parse(latestDeliveryDate).
            atStartOfDay(ZoneId.systemDefault()).toInstant();
    }

    @Suspendable
    @Override
    public String call() throws FlowException {
        this.seller = getOurIdentity();

        // Step 1. Check if an order with this ID already exists
        QueryCriteria.LinearStateQueryCriteria queryCriteria = new
            QueryCriteria.LinearStateQueryCriteria().withExternalId(
            Collections.singletonList(this.orderId));
        List<StateAndRef<OrderState>> results = getServiceHub().
            getVaultService().queryBy(OrderState.class, queryCriteria).
            getStates();
        if (results.size() != 0) {
            throw new IllegalArgumentException("An order with ID " + this.
                orderId + " already exists.");
        }

        // Step 2. Get a reference to the notary service on our network and
            our key pair.
        final Party notary = getServiceHub().getNetworkMapCache().
            getNotaryIdentities().get(0);

        // Step 3. Compose the State that carries the order data.
        progressTracker.setCurrentStep(GENERATING_TRANSACTION);
        Party buyerParty = getServiceHub().getIdentityService().
            partiesFromName(this.buyer, true).stream().findFirst().get();
        final OrderState output = new OrderState(this.seller, buyerParty,
            this.orderId, this.productId, this.quantity, this.price, this.
            shippingCosts, this.shippingAddress, this.latestDeliveryDate);

        // Step 4. Create a new TransactionBuilder object.
        final TransactionBuilder builder = new TransactionBuilder(notary);

        // Step 5. Add the order as an output state, as well as a command to
            the transaction builder.
        builder.addOutputState(output, TradeFinanceContract.ID);
        builder.addCommand(new TradeFinanceContract.Commands.Create(
            getOurIdentity()), output.getParticipants().stream().map(
            AbstractParty::getOwningKey).collect(Collectors.toList()));

        // Step 6. Verify and sign it with our KeyPair.
        progressTracker.setCurrentStep(SIGNING_TRANSACTION);
```

```java
        builder.verify(getServiceHub());
        final SignedTransaction ptx = getServiceHub().signInitialTransaction(builder);

        // Step 7. Collect the other party's signature using the SignTransactionFlow.
        progressTracker.setCurrentStep(COLLECTING_SIGNATURES);
        List<Party> otherParties = output.getParticipants().stream().map(el -> (Party) el).collect(Collectors.toList());
        otherParties.remove(getOurIdentity());
        List<FlowSession> sessions = otherParties.stream().map(this::initiateFlow).collect(Collectors.toList());

        SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx, sessions));

        // Step 8. Assuming no exceptions, we can now finalise the transaction
        progressTracker.setCurrentStep(FINALISING_TRANSACTION);
        subFlow(new FinalityFlow(stx, sessions));

        return "Order with ID '" + this.orderId + "' of buyer '" + buyerParty.getName() + "' added.";
    }
}
```

CreateOrderResponder.java

```java
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.flows.*;
import net.corda.core.transactions.SignedTransaction;

// ******************
// * Responder flow *
// ******************
@InitiatedBy(CreateOrder.class)
public class CreateOrderResponder extends FlowLogic<Void> {

    //private variable
    private FlowSession counterpartySession;

    //Constructor
    public CreateOrderResponder(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
```

```java
        SignedTransaction signedTransaction = subFlow(new SignTransactionFlow(counterpartySession) {
            @Suspendable
            @Override
            protected void checkTransaction(SignedTransaction stx) throws FlowException {
                /*
                 * SignTransactionFlow will automatically verify the transaction and its signatures before signing it.
                 * However, just because a transaction is contractually valid doesn’t mean we necessarily want to sign.
                 * What if we don’t want to deal with the counterparty in question, or the value is too high,
                 * or we’re not happy with the transaction’s structure? checkTransaction
                 * allows us to define these additional checks. If any of these conditions are not met,
                 * we will not sign the transaction - even if the transaction and its signatures are contractually valid.
                 * ----------
                 * For this cordapp, we will not implement any additional checks.
                 * */
            }
        });
        //Stored the transaction into data base.
        subFlow(new ReceiveFinalityFlow(counterpartySession, signedTransaction.getId()));
        return null;
    }
}
```

ShipOrder.java

```java
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import com.template.contracts.TradeFinanceContract;
import com.template.states.OrderState;
import com.template.utils.DataUtils;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.flows.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.util.List;
import java.util.stream.Collectors;

// ******************
```

```
// * Initiator flow *
// ******************
@InitiatingFlow
@StartableByRPC
public class ShipOrder extends FlowLogic<String> {
    private final ProgressTracker progressTracker = tracker();

    private static final ProgressTracker.Step GENERATING_TRANSACTION = new
        ProgressTracker.Step("Generating a ShipOrder transaction");
    private static final ProgressTracker.Step SIGNING_TRANSACTION = new
        ProgressTracker.Step("Signing transaction with our private key.");
    private static final ProgressTracker.Step COLLECTING_SIGNATURES = new
        ProgressTracker.Step("Collecting the signatures of the other parties.
        ");
    private static final ProgressTracker.Step FINALISING_TRANSACTION = new
        ProgressTracker.Step("Recording transaction") {
        @Override
        public ProgressTracker childProgressTracker() {
            return FinalityFlow.tracker();
        }
    };

    private static ProgressTracker tracker() {
        return new ProgressTracker(
                GENERATING_TRANSACTION,
                SIGNING_TRANSACTION,
                COLLECTING_SIGNATURES,
                FINALISING_TRANSACTION
        );
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    //private variables
    private final String orderId;
    private final String freightCompany;
    private final String trackingCode;

    //public constructor
    public ShipOrder(String orderId, String freightCompany, String
        trackingCode) {
        this.orderId = orderId;
        this.freightCompany = freightCompany;
        this.trackingCode = trackingCode;
    }

    @Suspendable
    @Override
    public String call() throws FlowException {
        // Step 1. Check if an order with this ID already exists
```

```java
        StateAndRef<OrderState> inputOrderStateAndRef = DataUtils.getOrder(
            getServiceHub(), this.orderId);
        OrderState inputOrderState = inputOrderStateAndRef.getState().getData
            ();

        // Generate State for transfer
        // Step 2. Get a reference to the notary service on our network and
            our key pair.
        final Party notary = getServiceHub().getNetworkMapCache().
            getNotaryIdentities().get(0);

        // Step 3. Compose the State that carries the order data
        progressTracker.setCurrentStep(GENERATING_TRANSACTION);
        final Party freightParty = getServiceHub().getIdentityService().
            partiesFromName(this.freightCompany, true).stream().findFirst().
            get();
        OrderState outputOrderState = inputOrderState.copy();
        outputOrderState.setOrderState(OrderState.State.SHIPPED);
        outputOrderState.setFreightCompany(freightParty);
        outputOrderState.setTrackingCode(this.trackingCode);

        // Step 4. Create a new TransactionBuilder object.
        final TransactionBuilder builder = new TransactionBuilder(notary);

        // Step 5. Add the order as an output state, as well as a command to
            the transaction builder.
        builder.addInputState(inputOrderStateAndRef);
        builder.addOutputState(outputOrderState);
        builder.addCommand(new TradeFinanceContract.Commands.Ship(
            getOurIdentity()), outputOrderState.getParticipants().stream().
            map(AbstractParty::getOwningKey).collect(Collectors.toList()));

        // Step 6. Verify and sign it with our KeyPair.
        progressTracker.setCurrentStep(SIGNING_TRANSACTION);
        builder.verify(getServiceHub());
        final SignedTransaction ptx = getServiceHub().signInitialTransaction(
            builder);

        // Step 7. Collect the other party's signature using the
            SignTransactionFlow.
        progressTracker.setCurrentStep(COLLECTING_SIGNATURES);
        List<Party> otherParties = outputOrderState.getParticipants().stream
            ().map(el -> (Party) el).collect(Collectors.toList());
        otherParties.remove(getOurIdentity());
        List<FlowSession> sessions = otherParties.stream().map(this::
            initiateFlow).collect(Collectors.toList());

        SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx,
            sessions));

        // Step 8. Assuming no exceptions, we can now finalise the
            transaction
        progressTracker.setCurrentStep(FINALISING_TRANSACTION);
```

```
        subFlow(new FinalityFlow(stx, sessions));

        return "Ship order flow for order with ID ’" + this.orderId + "’ of buyer ’" + outputOrderState.getBuyer().getName() + "’ executed.";
    }
}
```

ShipOrderResponder.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.flows.*;
import net.corda.core.transactions.SignedTransaction;

// ******************
// * Responder flow *
// ******************
@InitiatedBy(ShipOrder.class)
public class ShipOrderResponder extends FlowLogic<Void> {

    //private variable
    private FlowSession counterpartySession;

    //Constructor
    public ShipOrderResponder(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
        SignedTransaction signedTransaction = subFlow(new SignTransactionFlow(counterpartySession) {
            @Suspendable
            @Override
            protected void checkTransaction(SignedTransaction stx) throws FlowException {
                /*
                 * SignTransactionFlow will automatically verify the transaction and its signatures before signing it.
                 * However, just because a transaction is contractually valid doesn’t mean we necessarily want to sign.
                 * What if we don’t want to deal with the counterparty in question, or the value is too high,
                 * or we’re not happy with the transaction’s structure? checkTransaction
                 * allows us to define these additional checks. If any of these conditions are not met,
                 * we will not sign the transaction - even if the transaction and its signatures are contractually valid.
                 * ----------
```

```
                 * For this cordapp, we will not implement any additional checks.
                 * */
            }
        });
        //Stored the transaction into data base.
        subFlow(new ReceiveFinalityFlow(counterpartySession, signedTransaction.getId()));
        return null;
    }
}
```

SignArrival.java

```
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import com.template.contracts.TradeFinanceContract;
import com.template.states.OrderState;
import com.template.utils.DataUtils;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.flows.*;
import net.corda.core.identity.AbstractParty;
import net.corda.core.identity.Party;
import net.corda.core.transactions.SignedTransaction;
import net.corda.core.transactions.TransactionBuilder;
import net.corda.core.utilities.ProgressTracker;

import java.util.List;
import java.util.stream.Collectors;

// ******************
// * Initiator flow *
// ******************
@InitiatingFlow
@StartableByRPC
public class SignArrival extends FlowLogic<String> {
    private final ProgressTracker progressTracker = tracker();

    private static final ProgressTracker.Step GENERATING_TRANSACTION = new ProgressTracker.Step("Generating a SignArrival transaction");
    private static final ProgressTracker.Step SIGNING_TRANSACTION = new ProgressTracker.Step("Signing transaction with our private key.");
    private static final ProgressTracker.Step COLLECTING_SIGNATURES = new ProgressTracker.Step("Collecting the signatures of the other parties.");
    private static final ProgressTracker.Step FINALISING_TRANSACTION = new ProgressTracker.Step("Recording transaction") {
        @Override
        public ProgressTracker childProgressTracker() {
            return FinalityFlow.tracker();
        }
```

```
    };

    private static ProgressTracker tracker() {
        return new ProgressTracker(
                GENERATING_TRANSACTION,
                SIGNING_TRANSACTION,
                COLLECTING_SIGNATURES,
                FINALISING_TRANSACTION
        );
    }

    @Override
    public ProgressTracker getProgressTracker() {
        return progressTracker;
    }

    //private variables
    private final String orderId;

    //public constructor
    public SignArrival(String orderId) {
        this.orderId = orderId;
    }

    @Suspendable
    @Override
    public String call() throws FlowException {
        String signer = "";
        // Step 1. Check if an order with this ID already exists
        StateAndRef<OrderState> inputOrderStateAndRef = DataUtils.getOrder(
            getServiceHub(), this.orderId);
        OrderState inputOrderState = inputOrderStateAndRef.getState().getData
            ();

        // Generate State for transfer
        // Step 2. Get a reference to the notary service on our network and
            our key pair.
        final Party notary = getServiceHub().getNetworkMapCache().
            getNotaryIdentities().get(0);

        // Step 3. Compose the State that carries the order data
        progressTracker.setCurrentStep(GENERATING_TRANSACTION);
        OrderState outputOrderState = inputOrderState.copy();
        if (getOurIdentity().getOwningKey().equals(outputOrderState.getBuyer
            ().getOwningKey())) {
            outputOrderState.setBuyerSigned(true);
            signer = outputOrderState.getBuyer().getName().toString();
        } else if (getOurIdentity().getOwningKey().equals(outputOrderState.
            getFreightCompany().getOwningKey())) {
            outputOrderState.setFreightSigned(true);
            signer = outputOrderState.getFreightCompany().getName().toString
                ();
        }
```

```java

        if (outputOrderState.isBuyerSigned() && outputOrderState.isFreightSigned()) {
            outputOrderState.setOrderState(OrderState.State.DELIVERED);
        }

        // Step 4. Create a new TransactionBuilder object.
        final TransactionBuilder builder = new TransactionBuilder(notary);

        // Step 5. Add the order as an output state, as well as a command to the transaction builder.
        builder.addInputState(inputOrderStateAndRef);
        builder.addOutputState(outputOrderState);
        builder.addCommand(new TradeFinanceContract.Commands.Sign(getOurIdentity()), outputOrderState.getParticipants().stream().map(AbstractParty::getOwningKey).collect(Collectors.toList()));

        // Step 6. Verify and sign it with our KeyPair.
        progressTracker.setCurrentStep(SIGNING_TRANSACTION);
        builder.verify(getServiceHub());
        final SignedTransaction ptx = getServiceHub().signInitialTransaction(builder);

        // Step 7. Collect the other party's signature using the SignTransactionFlow.
        progressTracker.setCurrentStep(COLLECTING_SIGNATURES);
        List<Party> otherParties = outputOrderState.getParticipants().stream().map(el -> (Party) el).collect(Collectors.toList());
        otherParties.remove(getOurIdentity());
        List<FlowSession> sessions = otherParties.stream().map(this::initiateFlow).collect(Collectors.toList());

        SignedTransaction stx = subFlow(new CollectSignaturesFlow(ptx, sessions));

        // Step 8. Assuming no exceptions, we can now finalise the transaction
        progressTracker.setCurrentStep(FINALISING_TRANSACTION);
        subFlow(new FinalityFlow(stx, sessions));

        return "The arrival of the order with ID '" + this.orderId + "' has been signed by '" + signer + "'";
    }
}
```

SignArrivalResponder.java

```java
package com.template.flows;

import co.paralleluniverse.fibers.Suspendable;
import net.corda.core.flows.*;
import net.corda.core.transactions.SignedTransaction;
```

```java

// ******************
// * Responder flow *
// ******************
@InitiatedBy(SignArrival.class)
public class SignArrivalResponder extends FlowLogic<Void> {

    //private variable
    private FlowSession counterpartySession;

    //Constructor
    public SignArrivalResponder(FlowSession counterpartySession) {
        this.counterpartySession = counterpartySession;
    }

    @Suspendable
    @Override
    public Void call() throws FlowException {
        SignedTransaction signedTransaction = subFlow(new SignTransactionFlow
            (counterpartySession) {
            @Suspendable
            @Override
            protected void checkTransaction(SignedTransaction stx) throws
                FlowException {
                /*
                 * SignTransactionFlow will automatically verify the
                    transaction and its signatures before signing it.
                 * However, just because a transaction is contractually valid
                     doesn't mean we necessarily want to sign.
                 * What if we don't want to deal with the counterparty in
                    question, or the value is too high,
                 * or we're not happy with the transaction's structure?
                    checkTransaction
                 * allows us to define these additional checks. If any of
                    these conditions are not met,
                 * we will not sign the transaction - even if the transaction
                     and its signatures are contractually valid.
                 * ----------
                 * For this cordapp, we will not implement any additional
                    checks.
                 * */
            }
        });
        //Stored the transaction into data base.
        subFlow(new ReceiveFinalityFlow(counterpartySession,
            signedTransaction.getId()));
        return null;
    }
}
```

FlowTests.java (Tests)

```java
package com.template;

import com.google.common.collect.ImmutableList;
import com.template.flows.*;
import com.template.states.OrderState;
import net.corda.core.concurrent.CordaFuture;
import net.corda.core.contracts.StateAndRef;
import net.corda.core.contracts.TransactionVerificationException;
import net.corda.core.flows.FlowLogic;
import net.corda.core.identity.CordaX500Name;
import net.corda.testing.node.MockNetwork;
import net.corda.testing.node.MockNetworkParameters;
import net.corda.testing.node.StartedMockNode;
import net.corda.testing.node.TestCordapp;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import java.util.List;
import java.util.concurrent.ExecutionException;

import static org.junit.Assert.assertEquals;

public class FlowTests {
    private MockNetwork network;
    private StartedMockNode sellerNode;
    private StartedMockNode buyerNode;
    private StartedMockNode freightNode;

    @Before
    public void setup() {
        network = new MockNetwork(new MockNetworkParameters().
            withCordappsForAllNodes(ImmutableList.of(
                TestCordapp.findCordapp("com.template.contracts"),
                TestCordapp.findCordapp("com.template.flows"))));
        sellerNode = network.createPartyNode(new CordaX500Name("Seller", "
            Berlin", "DE"));
        buyerNode = network.createPartyNode(new CordaX500Name("Buyer", "
            Vienna", "AT"));
        freightNode = network.createPartyNode(new CordaX500Name("Freight
            Company", "New York", "US"));
        // For real nodes this happens automatically, but we have to manually
             register the flow for tests.
        for (StartedMockNode node : ImmutableList.of(sellerNode, buyerNode,
            freightNode)) {
            node.registerInitiatedFlow(CancelOrderResponder.class);
            node.registerInitiatedFlow(CheckDeliveryDateResponder.class);
            node.registerInitiatedFlow(ConfirmOrderResponder.class);
            node.registerInitiatedFlow(CreateOrderResponder.class);
            node.registerInitiatedFlow(ShipOrderResponder.class);
            node.registerInitiatedFlow(SignArrivalResponder.class);
        }
        network.runNetwork();
```

```
    }

    @After
    public void tearDown() {
        network.stopNodes();
    }

    @Test
    public void createOrderTest() throws ExecutionException,
        InterruptedException {
        CreateOrder flow = new CreateOrder("Buyer", "1", 100, 2.0, "10 EUR",
            "2 EUR", "Karlsplatz 13, 1040 Wien", "2020-09-30");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID ’1’ of buyer ’" +
            buyerNode.getInfo().getLegalIdentities().get(0).getName() + "’
            added.");
    }

    @Test
    public void cancelOrderTest() throws ExecutionException,
        InterruptedException {
        FlowLogic<String> flow = new CreateOrder("Buyer", "1", 100, 2.0, "10
            EUR", "2 EUR", "Karlsplatz 13, 1040 Wien", "2020-09-30");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID ’1’ of buyer ’" +
            buyerNode.getInfo().getLegalIdentities().get(0).getName() + "’
            added.");

        flow = new CancelOrder("1");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Cancel flow for order with ID ’1’ of
            buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0).
            getName() + "’ executed.");
    }

    @Test
    public void confirmOrderTest() throws ExecutionException,
        InterruptedException {
        FlowLogic<String> flow = new CreateOrder("Buyer", "2", 123587, 5.0, "
            750 EUR", "4 EUR", "Ballhausplatz 2, 1010 Wien", "2020-12-01");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID ’2’ of buyer ’" +
            buyerNode.getInfo().getLegalIdentities().get(0).getName() + "’
            added.");

        flow = new ConfirmOrder("2");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Confirm order flow for order with ID
```

```
            ’2’ of buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0)
            .getName() + "’ executed.");
    }

    @Test
    public void shipOrderTest() throws ExecutionException,
        InterruptedException {
        FlowLogic<String> flow = new CreateOrder("Buyer", "2", 123587, 5.0, "
            750 EUR", "4 EUR", "Ballhausplatz 2, 1010 Wien", "2020-12-01");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID ’2’ of buyer ’" +
            buyerNode.getInfo().getLegalIdentities().get(0).getName() + "’
            added.");

        flow = new ConfirmOrder("2");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Confirm order flow for order with ID
            ’2’ of buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0)
            .getName() + "’ executed.");

        flow = new ShipOrder("2", "Freight Company", "XAFDWEQ");
        future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Ship order flow for order with ID ’2’
            of buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0).
            getName() + "’ executed.");
    }

    @Test
    public void signArrivalTest() throws ExecutionException,
        InterruptedException {
        FlowLogic<String> flow = new CreateOrder("Buyer", "2", 123587, 5.0, "
            750 EUR", "4 EUR", "Ballhausplatz 2, 1010 Wien", "2020-12-01");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID ’2’ of buyer ’" +
            buyerNode.getInfo().getLegalIdentities().get(0).getName() + "’
            added.");

        flow = new ConfirmOrder("2");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Confirm order flow for order with ID
            ’2’ of buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0)
            .getName() + "’ executed.");

        flow = new ShipOrder("2", "Freight Company", "XAFDWEQ");
        future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Ship order flow for order with ID ’2’
            of buyer ’" + buyerNode.getInfo().getLegalIdentities().get(0).
```

```
            getName() + "’ executed.");

        flow = new SignArrival("2");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("The arrival of the order with ID ’2’
            has been signed by ’" + buyerNode.getInfo().getLegalIdentities().
            get(0).getName() + "’");

        flow = new SignArrival("2");
        future = freightNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("The arrival of the order with ID ’2’
            has been signed by ’" + freightNode.getInfo().getLegalIdentities
            ().get(0).getName() + "’");

        // We check the recorded order in all three vaults.
        for (StartedMockNode node : ImmutableList.of(sellerNode, buyerNode,
            freightNode)) {
            node.transaction(() -> {
                List<StateAndRef<OrderState>> orders = node.getServices().
                    getVaultService().queryBy(OrderState.class).getStates();
                assertEquals(1, orders.size());
                OrderState recordedState = orders.get(0).getState().getData()
                    ;
                assertEquals(recordedState.getOrderState(), OrderState.State.
                    DELIVERED);
                return null;
            });
        }
    }

    @Test
    public void checkDeliveryDateTest() throws ExecutionException,
        InterruptedException {
        FlowLogic<String> flow = new CreateOrder("Buyer", "3", 68754, 1.0, "
            1337 EUR", "2 EUR", "Michaelerkuppel, 1010 Wien", "2020-08-15");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID ’3’ of buyer ’" +
            buyerNode.getInfo().getLegalIdentities().get(0).getName() + "’
            added.");

        flow = new CheckDeliveryDate("3");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Check delivery date flow for order with
             ID ’3’ of buyer ’" + buyerNode.getInfo().getLegalIdentities().
            get(0).getName() + "’ executed.");

        // We check the recorded order in all three vaults.
        for (StartedMockNode node : ImmutableList.of(sellerNode, buyerNode))
            {
```

```
            node.transaction(() -> {
                List<StateAndRef<OrderState>> orders = node.getServices().getVaultService().queryBy(OrderState.class).getStates();
                assertEquals(1, orders.size());
                OrderState recordedState = orders.get(0).getState().getData();
                assertEquals(OrderState.State.PASSED, recordedState.getOrderState());
                return null;
            });
        }
    }

    @Test(expected = Exception.class)
    public void confirmCancelledOrderTest() throws ExecutionException, InterruptedException {
        FlowLogic<String> flow = new CreateOrder("Buyer", "1", 100, 2.0, "10 EUR", "2 EUR", "Karlsplatz 13, 1040 Wien", "2020-09-30");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Order with ID '1' of buyer '" + buyerNode.getInfo().getLegalIdentities().get(0).getName() + "' added.");

        flow = new CancelOrder("1");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        assert future.get().contains("Cancel flow for order with ID '1' of buyer '" + buyerNode.getInfo().getLegalIdentities().get(0).getName() + "' executed.");

        flow = new ConfirmOrder("1");
        future = buyerNode.startFlow(flow);
        network.runNetwork();
        future.get();
    }

    @Test(expected = Exception.class)
    public void createOrderHighShippingTest() throws ExecutionException, InterruptedException {
        CreateOrder flow = new CreateOrder("Buyer", "1", 100, 2.0, "10 EUR", "2 EUR0", "Karlsplatz 13, 1040 Wien", "2020-09-30");
        CordaFuture<String> future = sellerNode.startFlow(flow);
        network.runNetwork();
        future.get();
    }
}
```