Capítulo

2

Blockchains com Hyperledger: conceitos, instalação, configuração e uso

Charles Christian Miers, Guilherme Piêgas Koslovski, Maurício Aronne Pillon, Marco Antonio Marques

PPGCA - LabP2D - UDESC

Resumo

A introdução dos contratos inteligentes possibilitou o desenvolvimento de uma série de inovações, tais como aplicações e organizações autônomas descentralizadas, propriedade inteligente, tokens inteligentes, dentre outros, que permitiram que a blockchain fosse além das soluções financeiras, fato conhecido como Blockchain 3.0. Dentre os diversos modelos de Blockchain 3.0, o Hyperledger é uma plataforma Distributed Ledger Technologies (DLT) permissionada, modular, configurável e de código aberto, desenvolvida pela Linux Foundation. O Hyperledger implementa o conceito de contratos inteligentes, que são regras de negócio implementadas como transações na blockchain e chamadas por transações subsequentes, permitindo o desenvolvimento de contratos de negócio e aplicações descentralizadas. Tais características tornaram o Hyperledeger Fabric uma solução de destaque na implementação de soluções de blockchain privadas ou consorciadas, estando presente nos maiores provedores de computação em nuvem. Neste contexto, este minicurso tem como objetivo a apresentação teórica e prática do procedimento de criação, configuração e uso de uma rede blockchain baseada no modelo Hyperledger Fabric consorciado.

2.1. Conceitos básicos

A blockchain é um livro-razão seguro, compartilhado e distribuído que facilita o processo de gravação e rastreamento de recursos sem a necessidade de confiança em uma autoridade central, permitindo que duas partes comuniquem-se e troquem recursos em uma rede, na qual decisões são tomadas pela maioria, e não por uma única entidade [Salman et al. 2019]. Este livro-razão é composto de transações assinadas criptograficamente e agrupadas em blocos. Cada bloco está ligado criptograficamente com o bloco anterior através de uma validação utilizando um mecanismo de consenso. Conforme novos blocos

são adicionados à cadeia, torna-se mais difícil a alteração dos blocos antigos [Yaga et al. 2019].

Apesar do interesse crescente sobre o tema, a aplicação, implementação e operação das tecnologias relacionadas com blockchain ainda são consideradas tarefas complexas. Esta tecnologia foi apresentada inicialmente em 2008 com o Bitcoin, com o objetivo de permitir a realização de transações financeiras sem a necessidade de um intermediário confiável. Desde então, com o desenvolvimento de aplicações distribuídas e contratos inteligentes (Subseção 2.1.4), evoluiu rapidamente para soluções em diversas áreas. Neste sentido, torna-se necessário revisitar alguns conceitos básicos sobre blockchains, de modo a facilitar sua compreensão e aplicabilidade. De posse destes conhecimentos é possível a abordagem de aspectos práticos e operacionais relacionados à instalação, configuração e usabilidade de uma blockchain consorciada modelo Hyperledger Fabric.

2.1.1. Tipos de blockchain

A evolução da tecnologia blockchain pode ser dividida em três etapas: blockchain 1.0, blockchain 2.0 e blockchain 3.0. A blockchain 1.0 está relacionada com o Bitcoin e criptomoedas de modo geral. O Bitcoin é o primeiro e maior exemplo deste modelo de blockchain e representa, aproximadamente, 60 % do valor total de mercado de criptomoedas, composto por mais de 7 mil criptomoedas e avaliado em mais de 440 bilhões de dólares. Segundo [Swan 2015], enquanto o objetivo da blockchain 1.0 é a descentralização do dinheiro e meios de pagamento, a blockchain 2.0 busca a descentralização de mercados de modo geral, contemplando transações envolvendo diversos tipos de ativos utilizando a blockchain. Os contratos inteligentes são o principal destaque da blockchain 2.0 e consistem em contratos implementados via código na blockchain, que são executados automaticamente quando as condições pré-definidas são atendidas, sem a necessidade de um intermediário confiável. Já a blockchain 3.0 busca a aplicação da tecnologia em diferentes setores, de modo que seu potencial disruptivo implique em evoluções importantes na sociedade.

2.1.2. Permissionamento

Uma blockchain pode ser classificada como permissionada ou não permissionada. No modelo permissionado uma organização ou um consórcio de organizações são responsáveis pela validação e armazenamento das transações realizadas. Neste modelo, os participantes são previamente conhecidos e dependem de autorização para integrar a rede, gerar ou validar transações. De modo geral, as blockchains permissionadas são úteis para empresas, bancos e instituições que necessitam cumprir uma série de regulamentações e demandam controle completo de seus dados [Sharma 2020]. No modelo não permissionado, a validação e armazenamento das transações dependem do trabalho de um grupo de agentes anônimos, conhecidos como mineradores. A atuação dos mineradores é incentivada através da distribuição de *tokens*, como recompensa pelo trabalho realizado. Outra característica deste modelo é a transparência; os participantes devem ter acesso às informações referentes às transações processadas pela rede, incluindo os endereços e a composição dos blocos. Com base no modelo de permissionamento adotado, as blockchains são categorizadas como públicas, consorciadas ou privadas [Miers et al. 2019]:

- Em uma blockchain pública, todos podem ler, enviar ou validar transações, além de participar do processo de consenso distribuído, sendo considerada um modelo totalmente descentralizado.
- Uma blockchain consorciada é composta por duas ou mais instituições parceiras, que podem alterar as regras conforme suas necessidades. O consenso é obtido através de um processo realizado por um grupo específico de participantes sendo, desta forma, parcialmente descentralizado.
- Já uma blockchain privada é utilizada em modelo de organização única, que pode alterar o funcionamento conforme seus interesses e necessidades. É um modelo centralizado com processo de consenso simples de ser obtido.

Cada uma destas categorias possui uma aplicação distinta, baseada em um conjunto de características da blockchain. A Tabela 2.1 apresenta alguns critérios de comparação entre as principais características dos três tipos de blockchain [Miers et al. 2019].

Tabela 2.1: Comparação entre modelos de acesso da blockchain [Miers et al. 2019].

Características	blockchain Pública	blockchain Consórcio	blockchain Privada
Consenso distribuído	Todos os nós	Nós selecionados	Nós selecionados
Permissão de verificação	Pública	Restrita	Restrita
Imutabilidade	Sim	Adulterável	Adulterável
Centralização	Descentralizado	Parcial	Centralizado
Processo de consenso	Todos os nós	Nós selecionados	Nós selecionados

O consenso distribuído define se todos os nós podem participar do processo de consenso ou apenas nós pré-determinados. A permissão de verificação pode variar entre pública ou restrita, com a identidade dos participantes podendo ser anônima (em blockchains públicas) ou conhecida, nos modelos consórcio e privado [Tinu 2018]. Apesar de, por característica, os dados armazenados na blockchain serem imutáveis após a obtenção de consenso, algumas implementações dos modelos consórcio e privada podem permitir alterações. Já quanto ao grau de centralização, os modelos podem variar de centralizado no modelo de blockchain privada a descentralizado, no modelo público. Por fim, o processo de consenso define se qualquer entidade pode participar do processo ou apenas entidades pré-selecionadas [Miers et al. 2019].

2.1.3. Mecanismos de consenso

Devido à sua natureza assíncrona e descentralizada, um aspecto chave da tecnologia blockchain é determinar quem publicará o próximo bloco. Quando um nó ingressa em uma blockchain, este concorda com o estado inicial do sistema, definido no bloco Gênesis. Todo bloco seguinte deve ser válido e também possível de ser validado, de maneira independente, por qualquer nó da rede. Combinando o estado inicial com a capacidade de verificar cada bloco desde então, os usuários podem concordar sobre o estado atual da blockchain [Yaga et al. 2018]. Porém, cada nó pode ter uma visão diferente do estado da blockchain em um dado instante. Para garantir a convergência dessas visões, a blockchain

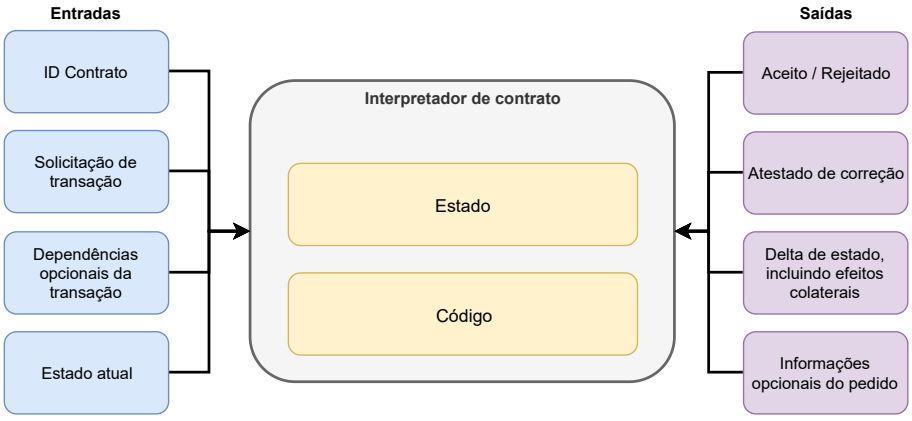
utiliza um mecanismo de consenso que permite que redes distribuídas ou descentralizadas tomem decisões unânimes quando necessário [Sankar et al. 2017]. Este mecanismo cria um sistema consistente, no qual todos os nós concordam com a ordem dos blocos e seus conteúdos.

De modo geral, o consenso pode ser obtido de diferentes maneiras, seja com o uso de algoritmos de loteria, como *Proof of Elapsed Time* (PoET) e *Proof of Work* (PoW), ou através do uso de métodos baseados em votação, tais como *Practical Byzantine Fault Tolerance* (PBFT) e RAFT [Hyperledger.Architecture 2017]. Dentre os mecanismos baseados em votação destacam-se os modelos *Crash Fault Tolerant* (CFT) e *Bizantine Fault Tolerant* (BFT). Enquanto o mecanismo CFT assume que alguns nós responsáveis pelo consenso podem falhar, o mecanismo BFT admite que além de falhar, alguns nós podem agir de maneira maliciosa. Cada um destes mecanismos possui benefícios e limitações. Desta forma, a escolha adequada à solução deve estar ligada ao cenário ao qual esta solução é implementada.

2.1.4. Contratos Inteligentes

Um contrato inteligente é uma regra de negócio contendo a lógica a ser executada em uma blockchain, podendo ser uma simples atualização de dados ou uma complexa execução de um contrato com condições anexas. Existem dois tipos diferentes de contratos inteligentes: os contratos inteligentes instalados e os contratos inteligentes na cadeia. Os contratos inteligentes instalados tem as regras de negócio instaladas nos validadores da rede antes que esta seja lançada. Já os contratos inteligentes na cadeia tem as regras de negócio implementadas como transações na blockchain, e são chamadas por transações subsequentes. Neste modelo, o código que define as regras de negócio torna-se parte da blockchain [Hyperledger.Smartcontracts 2017]. O modelo proposto na Figura 2.1 representa como uma solicitação enviada à um contrato inteligente é processada.

Figura 2.1: Processamento de contrato inteligente - Fonte: Adaptado de [Hyperledger.Smartcontracts 2017].



As saídas adequadas são geradas se a solicitação é válida e aceita. Estas saídas incluem o novo estado e possíveis efeitos colaterais desta alteração. Quando o processamento está concluído, o interpretador empacota o novo estado, um atestado de correção e qualquer outra informação adicional, e o envia para o serviço de consenso. A camada

de contrato inteligente valida cada solicitação, garantindo que está de acordo com as políticas da transação. Solicitações inválidas são rejeitadas e descartadas, sem inclusão na blockchain.

2.2. Introdução ao Hyperledger

O Projeto Hyperledger é uma iniciativa da Linux Foundation para desenvolver um ecossistema de código aberto de desenvolvimento de blockchain. A Linux Foundation visa criar um ambiente no qual comunidades de desenvolvedores de *software* e empresas se reúnam e se coordenem para construir estruturas de blockchain. O modelo Hyperledger é um projeto multiplataforma, modular e de código aberto. Esta abordagem apresenta, como diferenciais, características como extensibilidade, flexibilidade e a capacidade de modificar qualquer componente sem afetar o sistema [Hyperledger.Architecture 2017]. O Hyperledger é um projeto guarda-chuvas, sob o qual estão em desenvolvimento cinco *frameworks* distintos [Dhillon et al. 2017]:

- **Sawtooth**: Desenvolvido pela Intel com o objetivo de criar uma blockchain de código aberto, o Hyperledger Sawtooth é uma plataforma modular para desenvolver, implementar e executar *Distributed Ledger Technologies* (DLT). Adota um mecanismo de consenso conhecido como PoET, que tem como objetivo a obtenção de consenso com mínimo esforço computacional. É um *framework* que permite a implementação de soluções permissionadas e não permissionadas.

- **Fabric**: Este *framework* tem como objetivo permitir o desenvolvimento de aplicações empresariais com arquitetura modular, possibilitando a adoção de diferentes mecanismos de consenso e serviços únicos de filiação.

- **Iroha**: Conjunto de bibliotecas e componentes que permite a implementação de DLT em infraestruturas já existentes.

- **Burrow**: Blockchain permissionada que executa contratos inteligentes de maneira similar à *Ethereum Virtual Machine* (EVM). Permite a execução de contratos inteligentes em múltiplas blockchains compatíveis entre si porém em execução em domínios distintos.

- **Indy**: Trata-se de um *Software Development Kit* (SDK) para o Hyperledger que oferece componentes que permitem adicionar novos recursos e funcionalidades para gestão de identidades de maneira descentralizada.

Desta forma, existem projetos distintos de blockchain Hyperledger, cada qual possuindo características para um modelo específico de solução. Contudo, todos os projetos Hyperledger seguem um modelo de desenvolvimento que inclui uma abordagem modular e extensiva, interoperabilidade e ênfase em soluções seguras, com uma abordagem independente de *token* e sem criptomoeda nativa, além do uso de *Application Programming Interface* (API).

2.2.1. Hyperledger Fabric

O Hyperledger Fabric é uma arquitetura modular de blockchain que permite a conexão de componentes como mecanismo de consenso, serviços de filiação e funções de transação. Esta característica permite uma customização efetiva da plataforma, conforme as necessidades do ambiente. Assim, quando implementada em um ambiente composto por apenas uma empresa, por exemplo, o uso de um mecanismo de consenso CFT pode ser mais adequado que um mecanismo BFT. Por outro lado, em ambientes descentralizados multiorganizacionais, um mecanismo BFT pode ser necessário [Hyperledger 2020a]. Além do mecanismo de consenso, outros componentes do Hyperledger Fabric são modulares e configuráveis:

- *Ordering service*: Responsável por estabelecer o consenso quanto à ordem das transações e enviá-las aos *peers*.
- *Membership service provider*: Responsável pela associação de entidades que compõe a rede à identidades criptográficas.
- *Peer-to-peer gossip service*: Responsável por disseminar os blocos para todos os nós.
- *Chaincode*: No Hyperledger Fabric os contratos inteligentes são conhecidos também como *chaincode*, e podem ser desenvolvidos em Go, Javascript e, eventualmente, outra linguagem como Java. Neste modelo, existem dois tipos de *chaincode*: *system chaincode* e *application chaincode*. O *system chaincode* normalmente lida com transações relativas ao sistema, tais como gerenciamento do ciclo de vida e configurações de políticas. Já o *application chaincode* gerencia o estado do *ledger*, incluindo registros de dados.

Adicionalmente, a plataforma também dá suporte a uma variedade de sistemas de gerenciamento de banco de dados, e permite a configuração das políticas de endosso e validação exigidas. No Hyperledger Fabric, a blockchain é composta por nós pertencentes às organizações que compõe o consórcio, além de um ou mais nós do tipo *orderer*, responsáveis por ordenar as transações. Para poderem se comunicar, estes nós devem participar de um mesmo canal.

2.2.2. Canais e seus componentes

O modelo proposto pelo Hyperledger Fabric permite que as organizações participem de múltiplas redes blockchain independentes, através dos canais. Toda transação deve ser executada em um canal, no qual todos os participantes devem ser autenticados e autorizados a realizar transações. O canal oferece o compartilhamento de uma infraestrutura, mantendo a privacidade dos dados e da comunicação, e é composto pelos nós membros (*peer*) pertencentes às organizações participantes, nós âncora (*anchor peer)*, nós de ordenamento das transações (*order peer)*, *ledger* e *chaincode*.

Todas as organizações que compõe o canal devem possuir ao menos um *anchor peer*. Os *anchor peer* são responsáveis por comunicar-se com os *orderer peer*, obter os

blocos contendo as transações, e disseminá-los para os demais nós de suas organizações. Apesar de poderem pertencer a múltiplos canais, e possuir diversos *ledger*, os dados dos canais são privados, e não podem trafegar entre eles. Já os *orderer peer* são responsáveis por ordenar as transações através de um mecanismo de consenso, montar os blocos e entregá-los aos *anchor peer* [Hyperledger 2020a].

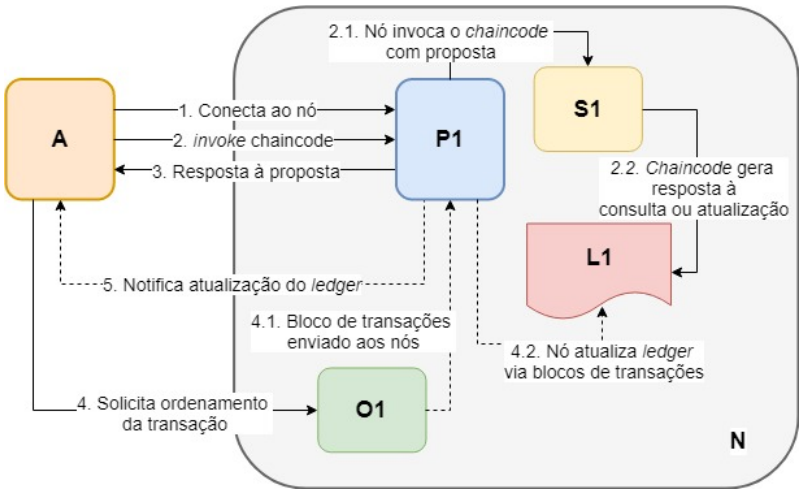
2.2.3. Arquitetura de transações

O Hyperledger Fabric implementa um novo modelo de arquitetura de transações composto de três passos e conhecido como *execute-order-validate*. O primeiro passo executa a transação, verifica sua exatidão e a endossa. O segundo passo ordena as transações utilizando um mecanismo de consenso. O terceiro passo valida as transações por meio de uma política de validação específica da aplicação, antes de adicioná-la ao *ledger*. A política de validação especifica quais e quantos nós devem validar a execução do *chaincode*.

Esta arquitetura agrega características para lidar com desafios referentes à escalabilidade, desempenho e confidencialidade existentes no modelo tradicional *order-validate*. Pode-se imaginar, como exemplo, um *chaincode* cuja execução seja um *loop* infinito. Em uma arquitetura *order-execute*, este *chaincode* teria uma consequência crítica. Já o modelo *execute-order-validate* é capaz de identificar este problema na primeira etapa de execução, quando os nós devem endossar as transações que serão submetidas à validação. Neste modelo, a transação não seria endossada e seria descartada antes de ser repassada para os nós *orderer* e de endosso.

Para realizar a execução e interação com os *chaincodes*, o Hyperledger Fabric faz uso das transações. Conforme apresenta [Dhillon et al. 2017], existem dois tipos de transações: *Invoke* e *Query*. A transação do tipo *invoke* executa um *chaincode* na blockchain, enquanto a transação *query* executa uma consulta. Através de uma transação *invoke* é possível a um cliente executar uma função específica contida em um *chaincode*. A Figura 2.2 ilustra como aplicações interagem com os nós para acessar o *ledger*.

Figura 2.2: Fluxo de transações no Hyperledger Fabric. Adaptado de [Hyperledger 2020a].



Neste exemplo, a aplicação A conecta ao nó P1 e realiza um *invoke* do *chaincode* S1, para realizar uma atualização do *ledger* L1. O nó P1 invoca S1, solicitando uma

resposta contendo o resultado da solicitação de atualização e retorna a resposta para A. A aplicação A recebe a resposta, monta uma transação contendo todas as respostas e envia ao ordenador O1. Caso a solicitação fosse apenas uma consulta ao *ledger*, o processo estaria completo. Em seguida, o ordenador coleta as transações e as agrupa em blocos, distribuindo-os a todos os nós da rede. P1 valida a transação antes de adicioná-la ao *ledger*. Por fim, P1 gera um evento e envia à aplicação.

2.2.4. Binários do Hyperledger Fabric

O Hyperledger Fabric dispõe de um conjunto de binários responsáveis pelas principais funcionalidades da plataforma:

- *Configtxgen*: Responsável pela geração dos artefatos de rede, tais como o genesis.block e channel.tx;
- *Configtxlator*: Responsável por gerar e configurar o canal de comunicação;
- *Cryptogen*: Responsável pela geração do material criptográfico;
- *Discovery*: Cliente de linha de comando para o serviço de descoberta;
- *Idemixgen*: Utilitário para geração de chaves a serem usadas com o *Membership Service Provider* (MSP);
- *Oderer*: Responsável pela interação com o nó de ordenamento;
- *Peer*: Responsável pela interação com o nó de validação; e
- *Fabric-ca-client*: Cliente para criação e registro de usuários.

2.3. Instalação e configuração do Hyperledger Fabric

Conforme apresentado na documentação oficial da plataforma, para a correta instalação e utilização do Hyperledger Fabric é necessária a instalação de um conjunto de pré-requisitos. Estes pré-requisitos, listados na Subseção 2.3.1, permitem a implementação dos nós que compõe a blockchain, geração das chaves criptográficas, instalação do *chaincode*, dentre outras ações necessárias.

Para apresentar as funcionalidades disponíveis no Hyperledger Fabric e como operá-las, este trabalho utiliza um repositório contendo um modelo de rede, os *scripts* e arquivos de configurações necessários para a implementação proposta. Este repositório é baseado no modelo desenvolvido por [Padvan 2020] que, por sua vez, tem como referência os exemplos disponibilizados pelo Hyperledger, na coletânea *fabric-samples*.

A instalação do Hyperledger Fabric pode ser realizada em ambientes GNU/Linux, MS-Windows e Apple MacOS, sendo que cada uma das instalações tem suas particularidades. Este trabalho adota como plataforma base de desenvolvimento o sistema operacional GNU/Linux, especificamente a distribuição Ubuntu Desktop 16.04.

2.3.1. Instalação dos pré-requisitos

A correta execução do Hyperledger Fabric depende da instalação de uma série de pré-requisitos. Com o objetivo de facilitar a instalação destas dependências, foi elaborado o *script preinstall.sh*, disponibilizado na raíz do repositório clonado. Para que seja possível clonar o repositório é necessário, primeiramente, realizar a instalação do Git, através do comando *sudo apt install git*. Em seguida, deve-se clonar o repositório, de preferência no diretório raiz, através do comando *git clone https://github.com/marco-developer/errc2020-hyperledgerfabric.git*. Por fim, o *script* deve ser executado como root, e instalará os seguintes componentes:

- cUrl: Ferramenta que permite a transferência de dados via linha de comando, utilizada para realizar o *download* de aplicações necessárias;
- Docker: Plataforma de código aberto que permite o desenvolvimento, implementação e execução de aplicações em contêineres. Utilizado para a criação e execução da rede e dos nós que compõe o Hyperledger Fabric. Versão utilizada: Última *stable*;
- Docker Compose: Ferramenta que auxilia a definição e execução de aplicações multi contêineres através de um arquivo YAML. Utilizada para facilitar a configuração e implementação dos nós que compõe a rede Hyperledger. Versão utilizada: 1.27.3;
- Binários do Hyperledger Fabric: Necessários para criação, configuração e interação com a rede Hyperledger;
- Imagens Docker: Imagens utilizadas para criação dos contêineres que compõe a plataforma;
- Coletânea de exemplos do Hyperledger Fabric: Exemplos disponibilizados pelos desenvolvedores que auxiliam na compreensão do funcionamento da plataforma;
- Node: Permite a execução de código Javascript utilizado na implementação e execução da plataforma. Versão utilizada: V12.19;
- Go: Linguagem de programação utilizada no desenvolvimento do *chaincode*. Versão utilizada: 1.15.3;
- NPM: Gerenciador de pacotes Javascript, utilizado na disponibilização da API para comunicação com a blockchain; e
- SSH: Protocolo de rede seguro, utilizado para acesso ao *host* de desenvolvimento.

Após a instalação das dependências acima listadas, o *script* adicionará à variável PATH do *bash* os caminhos referentes aos componentes, de modo que possam ser executados em qualquer diretório. Finalizada a execução do *script* sem erros, será possível iniciar a configuração do modelo de implementação proposto por este trabalho.

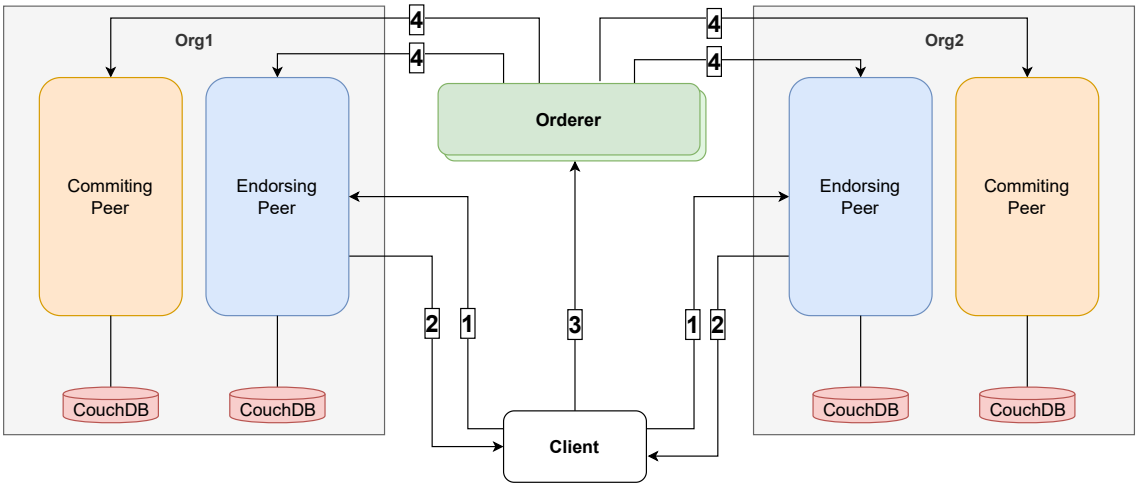
2.3.2. Proposta de implementação

Conforme citado na Seção 2.3, para explorar e apresentar as diversas opções de configuração da rede e seus respectivos arquivos, será utilizado neste minicurso um repositório contendo um modelo básico de rede.

Este repositório agrupa os arquivos de configuração necessários para a implementação e interação com a blockchain, detalhados na Subseção 2.3.3. A implementação proposta neste modelo consiste em uma blockchain composta por duas organizações, que possuem dois nós (*peer*), sendo um deles de endosso (*endorsing peer*) e uma *Certified Authority* (CA) cada. O serviço de ordenação das transações, por sua vez, é composto por três nós, utilizando o mecanismo de consenso RAFT.

O RAFT é um mecanismo que busca o consenso através da eleição de um líder, responsável pela gestão do consenso envolvendo as transações. Este líder recebe as transações e tem como tarefas replicá-las para os outros membros, que devem validá-las, além de informá-los quando devem aplicar as alterações propostas [Ongaro and Ousterhout 2014]. Os nós CouchDB são responsáveis pelo armazenamento do *ledger* contendo as transações validadas. É importante notar que todo nó *peer* demanda um nó CouchDB, que será responsável pelo armazenamento do *ledger*. A Figura 2.3 representa um exemplo de transação no modelo proposto.

Figura 2.3: Modelo de implementação.



Neste modelo (Figura 2.3) está representada uma transação do tipo *invoke*, na qual o cliente envia a proposta aos *endorser peer*(1), que executam e endossam a transação, retornando-a ao cliente(2). Em seguida, o cliente envia a proposta endossada ao *orderer*(3), que a ordena e reencaminha para todos os nós de validação(4). Estes nós validarão a transação e armazenarão o novo estado do *ledger* no CouchDB. Para a implementação do modelo proposto, é necessária a execução de uma série de etapas:

- Edição do arquivo *criptoconfig.yaml*, contendo as configurações referentes aos nós *orderer* e *peer*;
- Geração do material criptográfico relativo aos nós definidos em *criptoconfig.yaml*;

- Edição do arquivo *configtx.yaml*, que contém configurações referentes às organizações, nós *orderer*, canal e políticas de interação;
- Geração do bloco gênesis, da transação de configuração do canal e definição dos *anchor peer*;
- Edição do arquivo *docker-compose.yaml*, contendo as informações necessárias para a criação dos contêineres e operacionalização da rede Hyperledger Fabric;
- Criação do canal e conexão dos nós que o compõe;
- Instalação e inicialização do *chaincode*.

A Subseção 2.3.3 apresenta as principais funções dos arquivos de configuração *criptoconfig.yaml* e *configtx.yaml*, bem como a geração do material criptográfico, geração do bloco gênesis, criação e execução dos nós que compõe a rede. Já a Seção 2.4 apresenta o processo de criação do canal, conexão dos nós membros, instalação e inicialização do *chaincode*.

2.3.3. Configuração

O repositório utilizado para armazenar o exemplo em questão contempla os arquivos de configuração que definem a composição das organizações, as informações necessárias para a criação dos contêineres contendo os nós da rede, bem como a API utilizada para interação com a blockchain. Desta forma, é necessário compreender o conteúdo de cada um dos arquivos de configuração para que seja possível a implementação do modelo proposto.

Cryptoconfig.yaml

O arquivo de configuração *crytoconfig.yaml* define características dos nós *orderer* e *peer*, e é utilizado como entrada pelo binário *cryptogen* para a criação do material criptográfico. O Código 1 mostra a configuração do modelo de implementação proposto.

Código 1: criptoconfig.yaml - Definição dos nós

```
1  OrdererOrgs:
2      - Name: Orderer
3      Domain: example.com
4      EnableNodeOUs: true
5      - Specs:
6          - Hostname: orderer
7          SANS:
8              - "localhost"
9              - "127.0.0.1"
10         - Hostname: orderer2
11         SANS:
12             - "localhost"
13             - "127.0.0.1"
14         - Hostname: orderer3
15         SANS:
16             - "localhost"
17             - "127.0.0.1"
18 PeerOrgs:
19     - Name: Org1
20     Domain: org1.example.com
21     EnableNodeOUs: true
22     - Template:
23         Count: 2
24         SANS:
25             - "localhost"
26     - Users
27         Count: 1
28     - Name: Org2
29     Domain: org2.example.com
30     EnableNodeOUs: true
31     - Template:
32         Count: 2
33         SANS:
34             - "localhost"
35     - Users
36         Count: 1
```

Inicialmente, estão definidas as características genéricas dos nós *orderer*, como nome e o domínio, bem como as características específicas de cada nó, como *hostname* e endereço *Internet Protocol* (IP) (linhas 1 a 17). Também estão definidas as configurações das organizações, com seus respectivos nomes e domínio, dois nós *peer* (linhas 23 e 32) e um usuário (linhas 27 e 36). Após as definições destas configurações é possível prosseguir para a geração do material criptográfico referente aos nós, por meio da execução do comando:

cryptogen generate –config=./crypto-config.yaml –output=./crypto-config/.

O material criptográfico gerado com a execução deste comando será armazenado no caminho definido em "*–output*", no caso em *./crypto-config/.*

Configtx.yaml

O arquivo *configtx.yaml* inclui as configurações do canal de comunicação. Esta configuração é armazenada no *ledger*, e especifica quais organizações são membros do canal, quem são os nós responsáveis pelo ordenamento das transações e geração dos blocos, bem como a política de atualização do canal [Hyperledger 2020a]. As configurações definidas neste arquivo são utilizadas pelo binário *configtxgen* para geração do bloco gênesis. Este arquivo é dividido nos seguintes subgrupos de configuração:

- *Organizations*: Define quais organizações são membros do canal. Cada organização é identificada através de um MSP ID e um MSP de canal, que define direitos administrativos e de participação. O MSP de canal é armazenado na configuração do canal e contém os certificados utilizados para identificar os nós, aplicações e administradores da organização. O modelo de implementação proposto é composto de três organizações: Org1, Org2 e OrdererOrg, responsável pela administração do serviço de ordenamento. O serviço de ordenamento é implementado como organização pois as melhores práticas recomendam que os certificados de nós *peer* sejam emitidos por CA diferente dos certificados de nós *orderer*.

- *Capabilities*: Esta seção permite que organizações que executam versões diferentes dos binários do Hyperledger Fabric participem do mesmo canal. Existem três subgrupos distintos: *Application capabilities*, que define quais os recursos utilizados e a versão mínima do binário *peer*, *orderer capabilities*, que define os recursos utilizados pelos nós de ordenamento, tais como mecanismo de consenso, e a versão mínima do binário *orderer* e, por fim,*channel capabilities*, que define a versão mínima do Hyperledger Fabric permitida.

- *Application*: Define as políticas que controlam a interação entre organizações através do canal. Essas políticas controlam o número de nós de organizações necessários para aprovar uma definição de *chaincode* ou atualização da configuração do canal, além de definir as permissões para escrita ou consulta ao *ledger*.

- *Orderer*: Os nós de ordenamento definidos na configuração devem ser incluídos no *consenter set*, que é o grupo de nós habilitados a criar novos blocos e distribuí-los aos nós *peer* que compõe o canal. O Código 2 lista características de um nó *orderer*.

O campo *OrdererType* define o mecanismo de consenso adotado, no caso, o RAFT. O campo *Consenters* define a lista de nós *orderer* que podem participar do processo de consenso. Neste exemplo há apenas um nó *orderer*, definido como orderer.example.com, e utilizando a porta 7050. Já os campos *ClientTLSCert* e *ServerTLSCert* definem o caminho dos certificados TLS utilizados pelo nó. Por fim, o campo *Addresses* define o endereço de acesso ao nó.

Além destas configurações, esta seção traz também duas configurações importantes, referentes à configuração dos blocos: *BatchTimeout* e *BatchSize*. O campo *BatchTimeout* define com que frequência um bloco é criado. Já o campo *BatchSize* define o tamanho máximo do bloco, em quantidade de transações e em *bytes*.

Código 2: configtx.yaml - Definição de nó *orderer*

```
1  OrdererType: etcdraft
2
3  EtcdRaft:
4          Consenters:
5          - Host: orderer.example.com
6          Port: 7050
7          ClientTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
8          ServerTLSCert: crypto-config/ordererOrganizations/example.com/orderers/orderer.example.com/tls/server.crt
9  Addresses:
10         - orderer.example.com:7050
11
12 BatchTimeout: 2s
13 BatchSize:
14         MaxMessageCount: 10
15         AbsoluteMaxBytes: 99 MB
16         PreferredMaxBytes: 512 KB
```

- *Channel*: Define as políticas de mais alto nível da configuração, tais como algoritmo de *hash* e estrutura dos dados utilizada na criação de novos blocos. Para a maioria das implementações, as configurações padrão definidas nesta seção não requerem modificações.
- *Profiles*: Auxilia a ferramenta *configtxgen* a construir a configuração do canal. Esta reúne informações das seções anteriores utilizadas para criar a transação de criação do canal e para escrever o bloco gênesis.

Com o arquivo *configtx.yaml* adequadamente configurado é possível realizar a criação do bloco gênesis, da transação contendo as configurações do canal e definição dos nós âncora das organizações participantes. Para isso, são executados os comandos listados no Código 3, extraídos do *script create-artifacts.sh*.

Docker-compose.yaml

O arquivo *docker-compose.yaml* contém as configurações aplicadas a todos os contêineres que compõe a rede Hyperledger Fabric, e é utilizado como entrada para o docker-compose, durante o processo de criação e execução da rede. Para o modelo proposto, é necessário acessar o subdiretório */artifacts/* e executar o comando *docker-compose up -d*, que irá criar os contêineres referentes aos seguintes nós:

Código 3: create-artifacts.sh - Bloco gênesis e configuração do canal

```
# Define os nomes dos canais de sistema e comunicacao
SYS_CHANNEL="sys-channel"
CHANNEL_NAME=<nome do canal>

# Gera bloco genesis
configtxgen -profile OrdererGenesis -configPath . -channelID $SYS_CHANNEL -outputBlock ./genesis.block

# Gera transacao de configuracao do canal
configtxgen -profile BasicChannel -configPath . -outputCreateChannelTx ./<nome do canal>.tx -channelID
  $CHANNEL_NAME

# Define os nos ancora de cada organizacao
configtxgen -profile BasicChannel -configPath . -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID
  $CHANNEL_NAME -asOrg Org1MSP

configtxgen -profile BasicChannel -configPath . -outputAnchorPeersUpdate ./Org2MSPanchors.tx -channelID
  $CHANNEL_NAME -asOrg Org2MSP
```

- *Orderer*:
  - orderer.example.com
  - orderer2.example.com
  - orderer3.example.com
- Org1:
  - ca.org1.example.com
  - peer0.org1.example.com
  - peer1.org1.example.com
- Org2:
  - ca.org2.example.com
  - peer0.org2.example.com
  - peer1.org2.example.com
- Armazenamento de estado:
  - couchdb0.example.com
  - couchdb1.example.com
  - couchdb2.example.com
  - couchdb3.example.com

As configurações dos nós estabelecidas no *docker-compose.yaml* são divididas em quatro subgrupos, conforme o tipo de nó: CA, *peer*, *orderer* e CouchDB. Em cada subgrupo é definida a imagem a ser utilizada para montar o contêiner, o nome do *host*, as variáveis de ambiente do contêiner, os volumes mapeados, as portas a serem expostas, o comando a ser executado após a inicialização, as dependências e outras configurações necessárias. Neste arquivo, deve ser dada especial atenção às configurações das variáveis de ambiente e ao mapeamento dos volumes, pois envolvem caminhos de acesso à certificados e chaves, necessários para o correto funcionamento dos nós e, consequentemente, do ambiente.

2.4. Criação do canal e instalação do chaincode

Nas seções anteriores foram instalados os pré-requisitos, criados os materiais criptográficos referentes aos nós que compõe as organizações envolvidas, gerada a transação contendo as configurações do canal de comunicação, o bloco gênesis e criados e executados os contêineres que compõe a rede blockchain. Desta forma, com a rede blockchain operacional, é possível prosseguir com a criação do canal e posterior implementação do *chaincode*. Os comandos executados nesta seção dependem da definição prévia de um conjunto de variáveis globais. O Código 4 faz parte de *script createChannel.sh*, e apresenta as funções que definem as variáveis globais usadas durante a criação do canal e implementação do *chaincode*.

Código 4: createChannel.sh - Variáveis globais

```
# Define o caminho dos certificados das CAs das organizacoes:
export PEER0_ORG1_CA=<Caminho do cert. CA Org1 >
export PEER0_ORG2_CA=<Caminho do cert. CA Org2 >
export ORDERER_CA=<Caminho do cert. CA Orderer >

# Define as config. do MSP, caminho cert. TLS,
caminho config. do MSP e endereco de cada no:
setGlobalsPeerFor0Org1(){
        export CORE_PEER_LOCALMSPID="Org1MSP"
        export CORE_PEER_TLS_ROOTCERT_FILE=
        $PEER0_ORG1_CA
        export CORE_PEER_MSPCONFIGPATH=
        <Caminho do MSP>
        export CORE_PEER_ADDRESS=localhost:7051
}
setGlobalsPeer1ForOrg1(){
        export CORE_PEER_LOCALMSPID="Org1MSP"
        export CORE_PEER_TLS_ROOTCERT_FILE=
        $PEER0_ORG1_CA
        export CORE_PEER_MSPCONFIGPATH=
        <Caminho do MSP>
        export CORE_PEER_ADDRESS=localhost:8051
}
setGlobalsForPeer0Org2(){
        export CORE_PEER_LOCALMSPID="Org2MSP"
        export CORE_PEER_TLS_ROOTCERT_FILE=
        $PEER0_ORG2_CA
        export CORE_PEER_MSPCONFIGPATH=
        <Caminho do MSP>
        export CORE_PEER_ADDRESS=localhost:9051
}
setGlobalsForPeer1Org2(){
        export CORE_PEER_LOCALMSPID="Org2MSP"
        export CORE_PEER_TLS_ROOTCERT_FILE=
        $PEER0_ORG2_CA
        export CORE_PEER_MSPCONFIGPATH=
        <Caminho do MSP>
        export CORE_PEER_ADDRESS=localhost:10051
}
```

2.4.1. Criação do canal

Após a criação dos blocos de configuração, através do comando *configtxgen*, é possível proceder com a criação do canal de comunicação. Este processo envolve três etapas: criação do canal, conexão dos nós ao canal e definição dos *anchor peer* das organizações. O Código 5 apresenta a criação do canal, utilizando como entrada o bloco gerado anteriormente:

Código 5: createChannel.sh - Criação do canal

```
# Define variaveis globais
setGlobalsForPeer0Org1

# Cria canal
peer channel create
        -o localhost:7050
        -c "$CHANNEL_NAME"
        –ordererTLSHostnameOverride orderer.example.com
        -f ./artifacts/channel/$CHANNEL_NAME.tx
        –outputBlock ./channel-artifacts/$CHANNEL_NAME.block
        –tls true
        –cafile $ORDERER_CA
```

Este comando realiza a criação do canal por meio do nó *orderer* (*-o localhost:7050*), utilizando o arquivo de transação contendo as configurações do canal (*-f ./artifacts/channel/$CHANNEL_NAME.tx*) gerado pelo binário *configtxgen*. A saída deste comando será armazenada no bloco gênesis, por meio do argumento *–outputBlock <caminho>\ $CHANNEL_NAME.block*. A execução deste comando com sucesso apresentará no terminal uma saída indicando que o bloco zero (bloco gênesis) foi adicionado à blockchain.

2.4.2. Conectando os nós ao canal

Com o bloco gênesis adicionado, é possível realizar a conexão dos nós das organizações que irão integrar o canal. Para isso, é necessário executar o comando *peer channel join* em todos os nós das organizações que compõe o consórcio. O Código 6 apresenta o trecho do *script* que realiza a conexão dos nós ao canal.

Código 6: createChannel.sh - *join channel*

```
1  joinChannel(){
2  setGlobalsForPeer0Org1
3  peer channel join -b
     ./channel-artifacts/$CHANNEL_NAME.block
4
5  setGlobalsForPeer1Org1
6  peer channel join -b
     ./channel-artifacts/$CHANNEL_NAME.block
12 }
7  setGlobalsForPeer0Org2
8  peer channel join -b
     ./channel-artifacts/$CHANNEL_NAME.block
9
10 setGlobalsForPeer1Org2
11 peer channel join -b
     ./channel-artifacts/$CHANNEL_NAME.block
```

A execução deste comando com sucesso apresentará no terminal uma saída informando que a proposta de conexão do nó ao canal foi enviada com sucesso. Para realizar a conexão dos demais nós, basta definir as variáveis globais dos demais nós e repetir o procedimento.

2.4.3. Definição dos anchor peer

Este tipo de nó é um *peer* que comunica-se e pode ser descoberto por todos os outros nós da rede. Todas as organizações que compõe o consórcio devem possuir um ou mais *anchor peers*. Para o modelo proposto, vamos definir o nó *peer0* das organizações como *anchor peer*. O Código 7 traz a definição dos *anchor peers* de Org1 e Org2.

Código 7: createChannel.sh - Define *anchor peer*

```
1  # Define variaveis globais para Org1
2  setGlobalsForPeer0Org1
3
4  # Define anchor peer para Org1
5  peer channel update
6        -o localhost:7050
7        –ordererTLSHostnameOverride orderer.example.com
8        -c "$CHANNEL_NAME"
9        -f ./artifacts/chan-
           nel/${CORE_PEER_LOCALMSPID}anchors.tx
10       –tls true
11       –cafile $ORDERER_CA
12
13 # Define variaveis globais para Org2
14 setGlobalsPeer0Org2
15
16 # Define anchor peer para Org2
17 peer channel update
18       -o localhost:7050
19       –ordererTLSHostnameOverride orderer.example.com
20       -c "$CHANNEL_NAME"
21       -f ./artifacts/chan-
           nel/${CORE_PEER_LOCALMSPID}anchors.tx
22       –tls true
23       –cafile $ORDERER_CA
```

Com a execução destes comandos os nós *peer0.org1.example.com* e

peer0.org2.example.com serão definidos como os *anchor peers* das organizações, estando acessíveis a todos os participantes da rede.

2.4.4. Instalação do *chaincode*

Após a criação do canal e conexão dos nós participantes, é necessário realizar a instalação do *chaincode* naqueles nós que terão permissão para alterar valores no *ledger*. Seguindo o modelo proposto, cada organização terá um nó com o *chaincode* instalado, que será o *peer0*. O processo de instalação do *chaincode* é realizado pelo *script deploychaincode.sh*, e é composto pelas seguintes etapas:

- *Geração do pacote contendo chaincode*: O primeiro passo antes da instalação do *chaincode* é gerar seu pacote. Nesta etapa devem ser informados o caminho do *chaincode*, a linguagem utilizada e um *label* para identificação. O Código 8 apresenta o processo de geração do pacote contendo o *chaincode*.

Código 8: deployChaincode.sh - Empacota *chaincode*

```
1  # Define variaveis globais
2  setGlobalsForPeer0Org1
3
4  CHANNEL_NAME="$CHANNEL_NAME"
5  CC_RUNTIME_LANGUAGE="golang"
6  VERSION="1"
7  CC_SR_PATH="./artifacts/src/github.com/fabcar/go"
8  CC_NAME="fabcar"
9
10 # Gera pacote contendo chaincode
11 peer lifecycle chaincode package $CC_NAME.tar.gz –path $CC_SRC_PATH –lang $CC_RUNTIME_LANGUAGE
     –label $CC_NAME_$VERSION
```

Para geração do pacote, são definidas as variáveis globais do *peer0.org1*, o nome do canal de comunicação, nome e versão do *chaincode* e linguagem utilizada em seu desenvolvimento.

- *Instalação do Chaincode*: Após a geração do pacote contendo o *chaincode*, ele está apto a ser instalado nos devidos nós. O processo de instalação do *chaincode* é simples, conforme Código 9, extraído do *script* de instalação:

Código 9: deployChaincode.sh - Instalação do *chaincode*

```
1  # Define variaveis globais para Peer0 Org1
2  setGlobalsPeer0Org1
3
4  # Instala chaincode em Peer0 Org1
5  peer lifecycle chaincode install $CC_NAME.tar.gz
6
7  # Define variaveis globais para Peer0 Org2
8  setGlobalsPeer0Org2
9
10 # Instala chaincode em Peer0 Org1
11 peer lifecycle chaincode install $CC_NAME.tar.gz
```

- *Aprovação do chaincode*: Após a instalação do *chaincode*, o próximo passo deve ser a aprovação por parte das organizações que compõe o canal. O comando para

aprovação do *chaincode* demanda diversas informações, dentre elas o ID do *chaincode*. Desta forma, antes de emitir o comando para aprovação do *chaincode*, será executado o comando *queryinstalled*, para obter o ID, direcionando sua saída para a variável *PACKAGE_ID*, que será utilizada no comando de aprovação. Após a aprovação, é executado o comando *checkcommitreadiness*, que verifica o resultado da aprovação. O Código 10 exemplifica estes passos.

Código 10: deployChaincode.sh - Aprovação do *chaincode*

```
1  # Obtem ID do chaincode instalado
2  queryInstalled() {
3  setGlobalsPeer0Org1
4  peer lifecycle chaincode queryinstalled >&log.txt
5  cat log.txt
6  PACKAGE_ID=$(sed -n "/${CC_NAME}_${VERSION}/{s/^Package ID: //; s/, Label:.*$//; p;}"log.txt)
7  }
8
9  # Aprova chaincode na Org1
10 approveForMyOrg1() {
11 peer lifecycle chaincode approveformyorg -o localhost:7050
12       –ordererTLSHostnameOverride orderer.example.com –tls
13       –collections-config $PRIVATE_DATA_CONFIG
14       –cafile $ORDERER_CA –channelID $CHANNEL_NAME –name
15       ${CC_NAME} –version ${VERSION}
16       –init-required –package-id ${PACKAGE_ID}
17       –sequence ${VERSION}
18 }
19
20 # Verifica aprovacao do chaincode
21 checkCommitReadyness() {
22   peer lifecycle chaincode checkcommitreadiness –collections-config $PRIVATE_DATA_CONFIG
23       –channelID $CHANNEL_NAME –name ${CC_NAME} –version ${VERSION}
24       –sequence ${VERSION} –output json –init-required
25 }
```

A primeira função obterá o ID do pacote, a segunda função utilizará este ID para aprovar o *chaincode* instalado no nó *peer0.org1*, e a terceira função irá verificar se o *chaincode* foi instalado corretamente, retornando o resultado desta consulta em formato *JavaScript Object Notation* (JSON). Após a aprovação para a organização 1, o mesmo procedimento deve ser realizado para a organização 2, devendo alterar apenas a definição das variáveis globais de *SetGlobalsForPeer0Org1* para *SetGlobalsForPeer0Org2*.

- *Registro do chaincode*: Uma vez que o *chaincode* foi aprovado por um número suficiente de organizações (por padrão a maioria delas), ele pode ser registrado através do comando *commit*, conforme apresentado no Código 11.

  A execução da função *commitChaincodeDefinition* irá realizar o *commit* do *chaincode* nos nós *peer0* das organizações que compõe o canal. Após sua execução, a função *queryCommited* irá verificar quais *chaincodes* estão instalados no canal especificado. Desta forma, a correta instalação e registro do *chaincode* deve apresentar o nome do *chaincode* como retorno da função *queryCommited*.

- *Inicialização do chaincode*: Por fim, o *chaincode* aprovado deve ser inicializado nos *peers*. Para isso, deve ser executado o comando *invoke*, utilizando o argumento *–isInit*, conforme Código 12.

Código 11: deployChaincode.sh - Registro do *chaincode*

```
# Registra o chaincode
commitChaincodeDefinition() {
setGlobalsPeer0Org1

peer lifecycle chaincode commit -o localhost:7050
        –ordererTLSHostnameOverride orderer.example.com
        –tls $CORE_PEER_TLS_ENABLED –cafile $ORDERER_CA
        –channelID $CHANNEL_NAME –name ${CC_NAME}
        –collections-config $PRIVATE_DATA_CONFIG
        –peerAddresses localhost:7051 –tlsRootCertFiles $PEER0_ORG1_CA
        –peerAddresses localhost:9051 –tlsRootCertFiles $PEER0_ORG2_CA
        –version ${VERSION} –sequence ${VERSION} –init-required
}
# Verifica chaincode instalado
queryCommitted() {
setGlobalsPeer0Org1

peer lifecycle chaincode querycommitted –channelID $CHANNEL_NAME –name ${CC_NAME}
}
```

Código 12: deployChaincode.sh - Inicia *chaincode*

```
# Funcao de inicializacao do chaincode
chaincodeInvokeInit() {
setGlobalsPeer0Org1

peer chaincode invoke -o localhost:7050
        –ordererTLSHostnameOverride orderer.example.com
        –tls $CORE_PEER_TLS_ENABLED
        –cafile $ORDERER_CA
        -C $CHANNEL_NAME -n ${CC_NAME}
        –peerAddresses localhost:7051 –tlsRootCertFiles $PEER0_ORG1_CA
        –peerAddresses localhost:9051 –tlsRootCertFiles $PEER0_ORG2_CA
        –isInit -c '{"Args":[]}'
}
```

A execução da função *chaincodeInvokeInit* irá inicializar o *chaincode* por meio de uma transação do tipo *invoke*, que não requer argumentos adicionais. Com isso, o *chaincode* está instalado e funcional, sendo possível enviar transações de atualização ou consulta ao *ledger*.

O *script deployChaincode.sh* possui, ainda, a função *chaincodeInvoke*, executada após instalação do *chaincode*. Esta função, presente no *chaincode*, não demanda argumentos, e tem como objetivo inicializar o *chaincode* adicionando um conjunto pré-definido de dados ao *ledger*, conforme apresentado no Código 13.

2.5. Interagindo com a blockchain

Após a inicialização com sucesso do *chaincode*, a blockchain está operacional, sendo possível interagir por meio dos diferentes tipos de transação disponíveis. Esta interação com a blockchain pode ser realizada via linha de comando ou através de API.

2.5.1. Interação via linha de comando

A interação por meio de linha de comando dá-se através da emissão de comandos utilizando o binário *peer*, de modo semelhante aos comandos emitidos nos procedimentos de

Código 13: fabcar.go - Função *initLedger*

```
func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface) sc.Response {

cars := []Car{
Car{Make: "Toyota", Model: "Prius", Colour: "blue", Owner: "Tomoko"},
Car{Make: "Ford", Model: "Mustang", Colour: "red", Owner: "Brad"},
Car{Make: "Hyundai", Model: "Tucson", Colour: "green", Owner: "Jin Soo"},
Car{Make: "Volkswagen", Model: "Passat", Colour: "yellow", Owner: "Max"},
Car{Make: "Tesla", Model: "S", Colour: "black", Owner: "Adriana"},
}

i := 0
for i < len(cars) {
        carAsBytes, _ := json.Marshal(cars[i])
        APIstub.PutState("CAR"+strconv.Itoa(i), carAsBytes)
        i = i + 1
}
return shim.Success(nil)
}
```

criação do canal e instalação do *chaincode*. O repositório compartilhado inclui o *script* interact.sh, desenvolvido para demonstrar exemplos de interação com a blockchain por meio da linha de comando. Neste *script* estão contidas algumas funções de exemplo, dentre as quais três funções básicas: *addCar*, *queryCar* e *queryAllCars*. A função *addCar* recebe como entrada os valores referentes aos campos definidos na função homônima, presente no *chaincode*, e está representada no código 14.

Código 14: interact.sh - Função addCar

```
addCar() {
setGlobalsForPeer0Org1

# Solicita dados para transacao
echo -e '\nDigite os dados da transacao: \n'

echo -e '\n Chave: '
read chave
echo -e '\n Make: '
read make
echo -e '\n Model: '
read model
echo -e '\n Color: '
read color
echo -e '\n Owner: '
read owner
echo -e '\n'

# Envia transacao
peer chaincode invoke -o localhost:7050
        –ordererTLSHostnameOverride orderer.example.com
        –tls $CORE_PEER_TLS_ENABLED
        –cafile $ORDERER_CA
        -C $CHANNEL_NAME -n $CC_NAME
        –peerAddresses localhost:7051 –tlsRootCertFiles
          $PEER0_ORG1_CA
        –peerAddresses localhost:9051 –tlsRootCertFiles
          $PEER0_ORG2_CA
        -c '"Args":["createCar", "'$chave'", "'$make'",
          "'$model'", "'$color'", "'$owner'"]'
}
```

Após receber e armazenar os dados necessários, a função *addCar* utiliza uma transação *invoke*, informando os *peers* de endosso e *orderer*, passando como argumentos a função do *chaincode* a ser executada, no caso *createCar*, e os argumentos necessários.

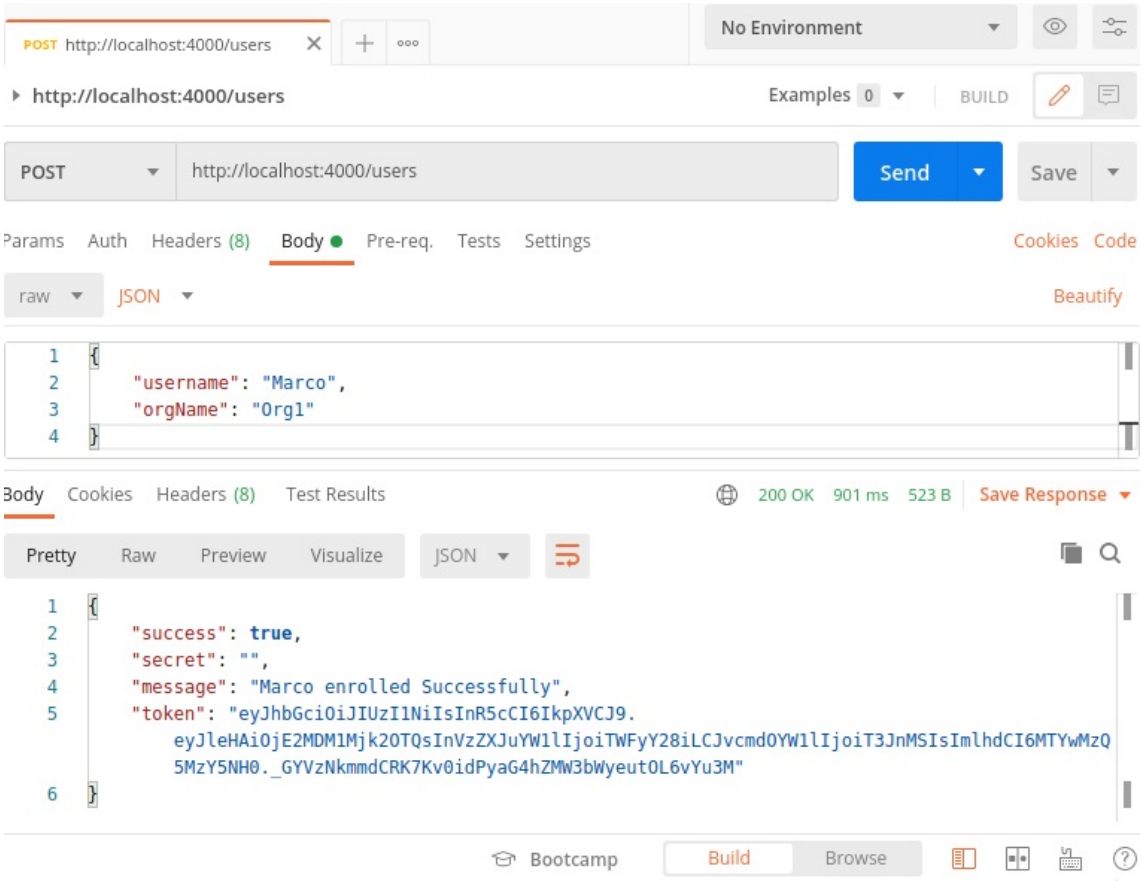
A função *queryCar*, por sua vez, recebe como entrada uma chave e retorna o registro equivalente, ou nada, caso o registro não seja encontrado. Importante notar que esta busca é *case sensitive*. Para realizar esta busca a função utiliza o comando *peer chaincode query* passando como argumentos o canal, nome do *chaincode*, e nos argumentos a função a ser executada, no caso *queryCars* e a chave de busca. Por fim, a função *queryAllCars* não demanda argumentos. Esta função, também contida no *chaincode*, retorna todos os registros contidos na blockchain, por meio do comando *peer chaincode query*, pas-

sando como argumentos o canal, nome do *chaincode* e a função a ser executada, no caso *queryAllCars*.

2.5.2. Interação via API

O repositório possui, no subdiretório ./api-1.4/, uma estrutura de arquivos contendo um conjunto de funções javascript que permitem a interação com a blockchain via API. Para que a interação seja possível é necessária a instalação de um conjunto de módulos do node.js. O *script pre_API.sh* realiza a instalação dos módulos necessários para o correto funcionamento da API. Após a execução do *script*, deve ser executado o comando *node-mon app*, que irá iniciar o servidor e deixar a API acessível via porta 4000. Para interação com a API, será utilizado o aplicativo Postman, um cliente API gratuito que permite a criação e envio de solicitações HTTP e HTTPS, bem como o recebimento das respectivas respostas. A interação segura com a API demanda a geração de um conjunto de usuário e *token*, através do endpoint */users*. A Figura 2.4 apresenta uma solicitação do tipo POST, para geração de usuário e *token*.

Figura 2.4: Solicitação de geração de *token* para usuário



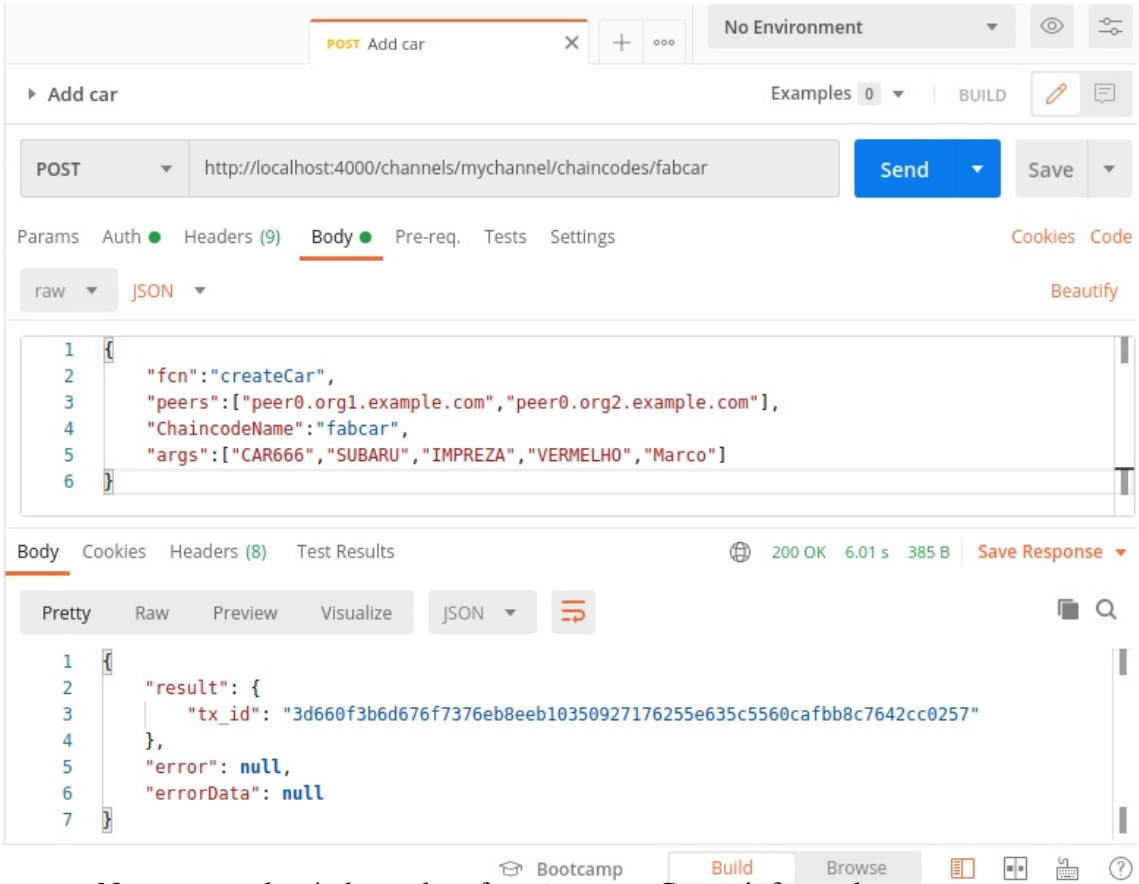
Neste exemplo, como o Postman está instalado no mesmo *host* que a blockchain, a solicitação é direcionada para *http://localhost:4000/users*, contendo no corpo da solicitação o nome de usuário e a organização ao qual pertence. A solicitação é executada e sua resposta apresentada na parte inferior. Na imagem temos uma solicitação executada com sucesso para geração do *token* referente ao usuário Marco, pertencente à organiza-

ção Org1. O conteúdo do campo *token* será, então, utilizado para envio de transações por parte deste usuário.

Transações do tipo *invoke* via API

Para transações do tipo *invoke*, que irão realizar alterações no *ledger*, o *token*, gerado conforme ilustrado na Figura 2.4, deve ser inserido no cabeçalho da solicitação POST. Esta solicitação deve ser encaminhada para */channels/mychannel/chaincodes/fabcar*, informando em seu corpo os parâmetros necessários, como a função a ser executada, os *peers* que irão validar a transação, o nome do *chaincode* e os argumentos da função chamada. Este *endpoint* especifica os respectivos canais e *chaincode* para os quais será enviada a transação, no caso o canal "*mychannel*"e *chaincode fabcar*. A Figura 2.5 apresenta uma solicitação contendo uma transação do tipo *invoke*.

Figura 2.5: Transação do tipo *invoke*



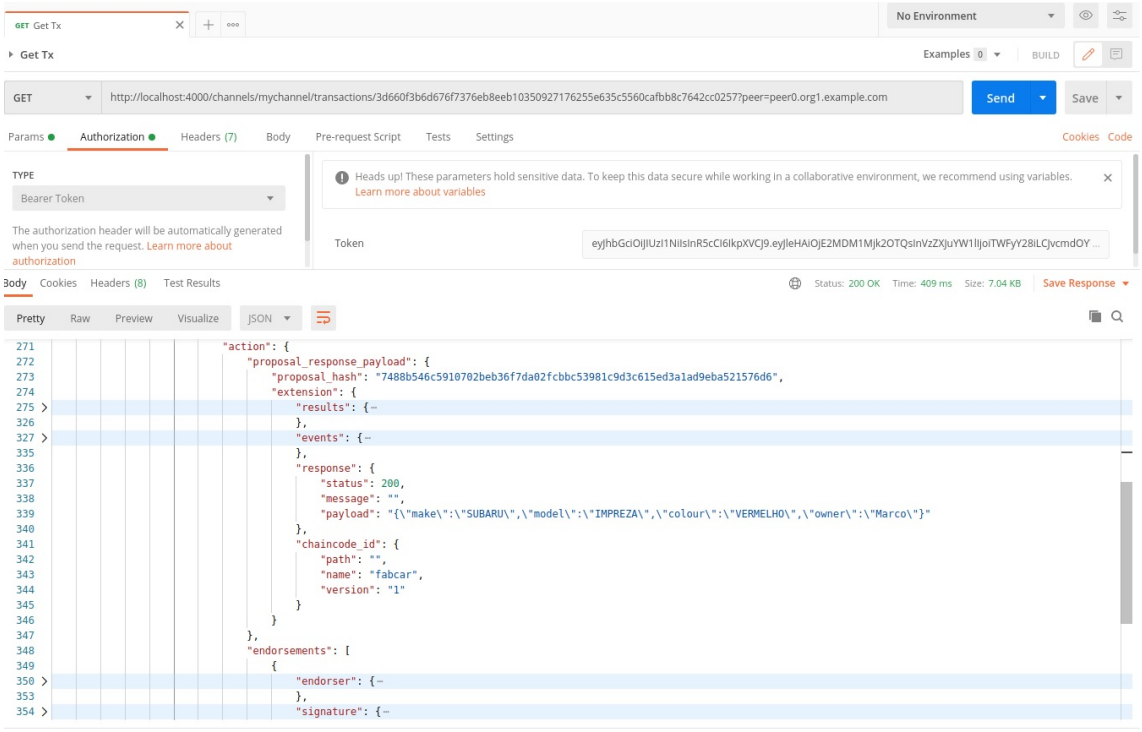
Neste exemplo, é chamada a função *createCar*, e informados os argumentos necessários para sua execução. A execução com sucesso retorna o campo *tx_id*, contendo o identificador da transação realizada.

Transações do tipo *query* via API

As transações do tipo *query* também podem ser executadas via API, com o uso do *token*. O procedimento é similar ao envio de transações do tipo *invoke*, porém mais simples, pois demanda menos argumentos. Para realizar uma consulta por meio de uma transação *query*, é necessário realizar uma solicitação do tipo GET, direcionada para o

endpoint /channels/mychannel/transactions/, informando o *token*, o identificador da transação e o *peer* que será utilizado para consulta. Desta forma, para realizar uma consulta à blockchain quanto à transação realizada na Figura 2.5, deve ser enviada uma solicitação GET, conforme apresentada na Figura 2.6.

Figura 2.6: Transação do tipo *query*



Esta solicitação irá realizar a consulta por meio do *peer0.org1.example.com*, utilizando o *token* informado. Como resposta, obtém-se um conjunto completo de informações sobre a transação, contendo seu conteúdo, os nós que endossaram a transação, seu *hash*, dentre outros.

2.5.3. Hyperledger Explorer

O Hyperledger Explorer é uma ferramenta que oferece uma interface para visualização das transações em uma blockchain Hyperledger Fabric. Dentre suas principais funcionalidades destacam-se sua interface web amigável, métricas referentes aos blocos e transações, ferramentas de busca e filtros para blocos e transações, descoberta dinâmica de novos canais e notificação em tempo real de novos blocos. Esta ferramenta está disponibilizada no repositório clonado, especificamente no subdiretório *./blockchainexplorer*. Sua execução depende, primeiramente, que a blockchain Hyperledger Fabric esteja configurada e em execução. Em seguida, é necessária a configuração correta dos arquivos de configuração *docker-compose.yaml*, *config.json* e *first-network.json*, estando os dois primeiros localizados no diretório *./blockchainexplorer* e o último no subdiretório *./blockchainexplorer/connection-profile*.

Esta aplicação pode ser executada de maneira local ou em contêineres Docker. Neste trabalho o Hyperledger Explorer é executado em ambiente Docker, sendo composto por dois contêineres: explorer e explorer-db. O contêiner explorer abrange o *front-end* da

aplicação, enquanto o explorer-db abriga o banco de dados PostgreSQL, responsável por persistir os dados do Hyperledger Fabric.

O primeiro passo para execução do Hyperledger Explorer envolve a configuração do arquivo *docker-compose.yaml*. Neste arquivo serão definidas as configurações dos contêineres que compõe a aplicação, tais como variáveis de ambiente, nome do *host*, nome do contêiner, rede, portas, volumes e dependências. O mapeamento dos volumes é essencial para integração com a blockchain, e já estão pré-configurados conforme a estrutura de diretórios definida no repositório, sendo necessário verificar apenas o volume referente à pasta contendo o material criptográfico (*/cripto-config/*). O arquivo *docker-compose.yaml* presente no repositório já considera o Hyperledger Explorer como uma subpasta do projeto. Desta forma, o volume referente à pasta cripto-config aponta para *../artifacts/channel/crypto-config/*.

O arquivo *config.json* informa qual o perfil a ser utilizado para as configurações de rede. No caso, o perfil definido aponta para o subdiretório *./connetion-profile/*, onde está localizado o arquivo *first-network.json*, que contém as configurações referentes à blockchain e ao nó com o qual o Hyperledger Explorer se comunicará para obter as informações. Este arquivo é composto por subgrupos contendo configurações específicas, tais como *client, channels, organizations e peers*. Em *client* estão as configurações quanto ao uso de TLS, usuário e senha de administrador, organizações participantes da rede, e *timeout* de conexão de nós *peer* e *orderer*. Em *channels* estão listadas as configurações dos canais, tais como os nós, e *timeout de conexão*. Em *organizations* constam o caminho referente à chave privada e ao certificado do administrador do MSP das organizações. Por fim, em *peers* estão listadas configurações e caminho do certificado CA dos nós *peer* e respectiva URL.

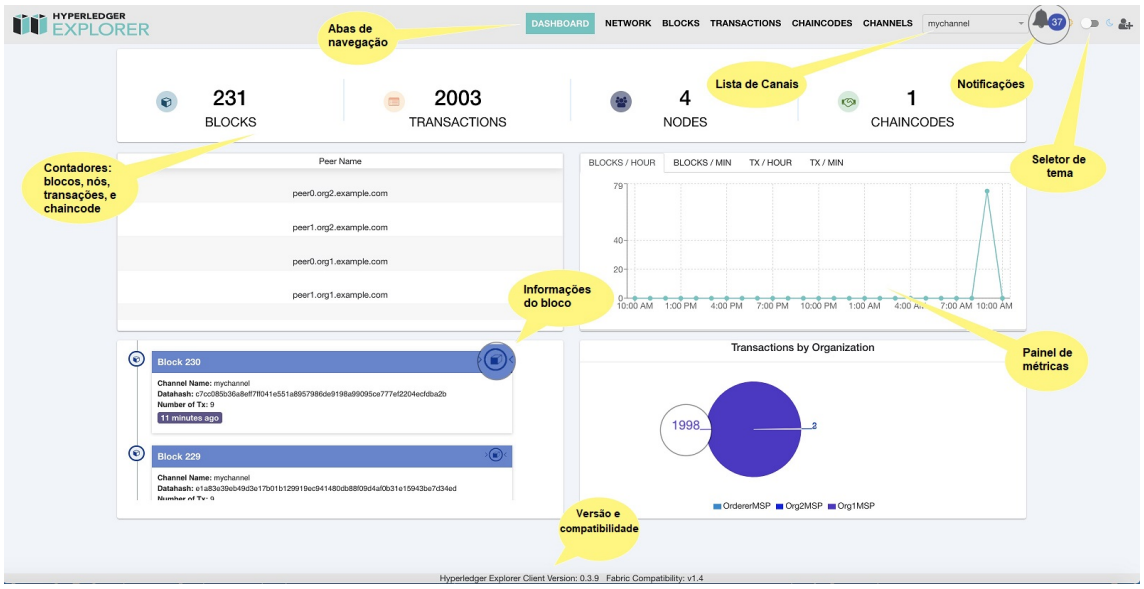
Com a configuração adequada destes arquivos, é possível iniciar a aplicação Hyperledger Explorer por meio do comando *docker-compose up -d*, a ser executado dentro do subdiretório *./blockchainexplorer*. A execução deste comando irá baixar as imagens necessárias, definidas em *docker-compose.yaml* e montar os contêineres. A interface da aplicação pode ser acessada via navegador, do próprio *host*, apontando para *http://localhost:8080* ou de outra máquina da rede, apontando para *http://<IP do host>:8080*. A Figura 2.7 ilustra a interface principal da aplicação.

A interface da aplicação apresenta um conjunto de painéis com as principais informações da blockchain, incluindo os últimos blocos e métricas referentes à blocos e transações. O menu principal é composto de um conjunto de abas de navegação, com acesso à informações sobre a rede, blocos, transações, *chaincodes* e canais. Desta forma, é possível navegar e explorar o conteúdo dos blocos, verificar seu *hash* e as transações que contém. É possível também verificar detalhes das transações, incluindo data e hora, os nós que a endossaram, seu *hash* e conteúdo.

2.6. Considerações

O modelo de implementação proposto e executado no presente trabalho busca detalhar as etapas de configuração e instalação de uma blockchain Hyperledger Fabric permissionada. Esta proposta tem enfoque educacional, para implementação em ambiente de testes. Para implementação em ambiente de produção, os códigos e configurações apre-

Figura 2.7: Hyperledger Explorer *dashboard* - Fonte: Adaptado de [Hyperledger 2020b]



sentados devem ser revistos, levando em conta as melhores práticas de desenvolvimento e segurança da informação. Neste sentido, o objetivo deste trabalho é apresentar os principais conceitos, configurações e arquitetura da solução de modo que, de posse destes conhecimentos, seja possível modelar e implementar soluções baseadas em blockchain Hyperledger Fabric em diversos cenários.

A tecnologia blockchain vem sendo amplamente estudada e implementada nos mais variados setores. Porém, apesar de seu potencial disruptivo, esta nem sempre é a melhor alternativa, sendo necessária uma análise criteriosa, considerando questões como integração com sistemas legado, capacidade e velocidade de processamento, escalabilidade e maturidade da tecnologia, afim de comprovar sua viabilidade e benefícios perante as soluções já existentes.

Agradecimentos

Os autores agradecem o apoio do Laboratório de Processamento Paralelo Distribuído (LabP2D) no Centro de Ciências Tecnológicas (CCT) da Universidade do Estado de Santa Catarina (UDESC).

Os autores agradecem o apoio da Fundação de Amparo à Pesquisa e Inovação do Estado de Santa Catarina (FAPESC).

This work was supported by Ripple's University Blockchain Research Initiative.

Referências

[Dhillon et al. 2017] Dhillon, V., Metcalf, D., and Hooper, M. (2017). The hyperledger project. In *Blockchain enabled applications*, pages 139–149. Springer.

[Hyperledger 2020a] Hyperledger (2020a). A blockchain platform for the enterprise. https://hyperledger-fabric.readthedocs.io/en/release-2.2/index.html.

[Hyperledger 2020b] Hyperledger (2020b). Hyperledger explorer docs. https://blockchain-explorer.readthedocs.io.

[Hyperledger.Architecture 2017] Hyperledger.Architecture (2017). Hyperledger architecture paper 1 consensus. https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

[Hyperledger.Smartcontracts 2017] Hyperledger.Smartcontracts (2017). Hyperledger architecture volume 2 - smart contracts. https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf.

[Miers et al. 2019] Miers, C., Koslovski, G., Pillon, M., Simplicio, M., Carvalho, T., Rodrigues, B., and Battisti, J. (2019). *Análise de Mecanismos para Consenso Distribuído Aplicados a Blockchain*.

[Ongaro and Ousterhout 2014] Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA. USENIX Association.

[Padvan 2020] Padvan, A. (2020). Basic network 2.0. https://github.com/adhavpavan/BasicNetwork-2.0.

[Salman et al. 2019] Salman, T., Zolanvari, M., Erbad, A., Jain, R., and Samaka, M. (2019). Security services using blockchains: A state of the art survey. *IEEE Communications Surveys Tutorials*, 21(1):858–880.

[Sankar et al. 2017] Sankar, L. S., Sindhu, M., and Sethumadhavan, M. (2017). Survey of consensus protocols on blockchain applications. In *2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)*, pages 1–5. IEEE.

[Sharma 2020] Sharma, T. K. (2020). Permissioned and permissionless blockchains: A comprehensive guide. https://www.blockchain-council.org/blockchain/permissioned-and-permissionless-blockchains-a-comprehensive-guide/.

[Swan 2015] Swan, M. (2015). *Blockchain: Blueprint for a new economy*. "O'Reilly Media, Inc.".

[Tinu 2018] Tinu, N. (2018). A survey on blockchain technology- taxonomy, consensus algorithms and applications. *International Journal of Computer Sciences and Engineering O*, 6.

[Yaga et al. 2018] Yaga, D., Mell, P., Roby, N., and Scarfone, K. (2018). Nistir 8202 - blockchain technology overview. Technical report.

[Yaga et al. 2019] Yaga, D., Mell, P., Roby, N., and Scarfone, K. (2019). Blockchain technology overview. *CoRR*, abs/1906.11078.