Adriano Manuel Tomé de Almeida Figueira Batista

Licenciado em Engenharia Informática

Ferramenta de Gestão de Projectos suportada pela tecnologia Blockchain

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientador: Carlos Eduardo Dias Coutinho, CEO, Caixa Mágica Software

Co-orientador: Luís Caires, Professor Catedrático, Universidade NOVA de Lisboa

Júri

Presidente: Professor Doutor Nuno Manuel Ribeiro Preguiça, FCT-UNL
Arguente: Professor Doutor Henrique João Lopes Domingos, FCT-UNL
Vogal: Doutor Carlos Eduardo Dias Coutinho, Caixa Mágica Software

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Dezembro, 2021

Ferramenta de Gestão de Projectos suportada pela tecnologia Blockchain

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "novathesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Resumo

O mercado de ferramentas de gestão de projectos está repleto de soluções bastante completas, que se centram, no entanto, apenas do lado dos gestores e desenvolvedores, deixando *stakeholders* externos fora do ciclo de desenvolvimento e facturação. A ligação entre estes *stakeholders* e o projecto é feita paralelamente pelo(s) gestor(es) responsáveis e não permite uma natural fluidez de informação entre os vários níveis de gestão.

Uma solução que permitisse esta transmissão e transparência de informação era uma mais-valia para qualquer empresa e transversal para todos os ramos de actividade.

Esta tese desenvolve e apresenta uma prova de conceito de uma ferramenta de gestão de projectos que, com recurso à tecnologia da *blockchain* - mais especificamente à plataforma Hyperledger Fabric - tenta combater os problemas anteriormente referidos providenciando uma *"unique source of truth"*, transparência entre os vários *stakeholders* envolvidos, redundância e imutabilidade de informação, facilitando assim o processo de auditoria. Adicionalmente, recorrendo a *Smart Contracts*, prevê-se uma optimização do *workflow*, automatizando a componente de gestão de risco de um projecto, documentação automática (*timesheets*, *task progress*, *team assignments*) e registo em tempo real de todos os riscos/problemas com falta de recursos ou datas de tarefas ou projectos.

Desta forma, irá ser descrito em detalhe o processo de construção da rede Hyperledger Fabric, discussão das funcionalidades da API e todos os pormenores relativos ao código do *chaincode* desenvolvido, apresentação dos resultados da estratégia implementada e, por fim, conclusões finais e possíveis futuros desenvolvimentos.

Palavras-chave: blockchain, gestão de projectos, smart contracts, Hyperledger Fabric

Abstract

The project management tools market is filled with complete and robust solutions, which, unfortunately focus only on the manager's and developer's side, leaving external stakeholders outside of the the development and billing cycle. The link between these stakeholders and the project is supported by the managers and doesn't allow a natural flow of information between the existing management levels.

A solution which facilitated this flow and transparency of information would be very useful for any company and transversal to all types of activity.

This thesis develops a proof-of-concept project management tool that, using blockchain technology - Hyperledger Fabric platform - provides a "unique source of truth", transparency between the stakeholders involved, redundancy of information and complete and unchanged data, and also eases possible audit procedures. Additionally, using Smart Contracts, it is expected a workflow optimization, for example automating the risk management component, automatic documentation (timesheets, task progress, team assignments) and real-time log of all risks / problems with lack of resources, project or task dates.

This document will describe in detail the process of building an Hyperledger Fabric network, discussion of API functionalities and all details regarding the developed chaincode code. In a later section, results are organized and discussed and, finally, some ending remarks and possible future developments.

Keywords: blockchain, project management, smart contracts, Hyperledger Fabric

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Glossário

containers	Ambiente virtual isolado com comunicação e alocação de recursos limitada.
enclave	No contexto do Intel SGX, enclaves são regiões de memória privadas, que podem incluir código de utilizador ou de sistema. O conteúdo dessas regiões está protegido contra leitura e escrita por parte de qualquer processo fora do enclave, incluindo processos com níveis de privilégio mais elevado.
postman	Ferramenta de construção e teste de APIs.

Siglas

CRUD	Create, Read, Update, Delete | Criar, Ler, Actualizar, Apagar
HLF	Hyperledger Fabric
KPI	Key Performance Indicator | Indicadores Chave de Performance
PME	Pequena e Média Empresa

Capítulo 1

Introdução

1.1 Enquadramento

1.1.1 Motivação

O mundo empresarial está repleto de desafios muito pouco explorados, e quando se fala em gestão de projecto, encontramos ainda uma grande base para evolução. Grande parte das soluções existentes não tratam todas as partes envolvidas num projecto da mesma forma, no que diz respeito a acesso a certos dados, interacção com o projecto ou outro tipo de apresentação de resultados.

Esta dissertação propõe e desenvolve uma nova ferramenta que, com recurso à tecnologia relativamente recente da Blockchain, prima por transparência e credibilidade de informação, pretendendo desta forma colmatar as falhas que existem no universo de soluções que tentam responder a este problema. Assim, permitirá também à empresa, a Caixa Mágica, que já tem experiência em projectos relacionados com a Blockchain - WalliD - voltar a apostar no desenvolvimento de um projecto nesta área promissora.

1.1.2 Soluções Existentes

Existem actualmente inúmeras ferramentas de gestão de projecto, no entanto, apenas muito recentemente surgiram as primeiras soluções que abordam o problema apresentado.

Uma primeira solução, a Alehub [1], apresenta-se como uma ferramenta com suporte de execução de *smart contracts*, processos intra-organizacionais e entre diferentes organizações. O ponto forte desta solução centra-se na livre definição e execução de contratos sob uma plataforma distribuída, recorrendo a um *token* criado pelos próprios (ALE Token) para reger interacções entre os vários intervenientes. Desta forma, a Alehub tem como

objectivo facilitar negociações, acordos e o grande número de processos que existem no contexto da gestão de projecto.

Noutra perspectiva, a Colony [5] propõe um novo método de gestão de grupos de trabalho descentralizados através da execução de *smart contracts*. Das várias funcionalidades oferecidas pelo sistema destacam-se a criação de *tokens*, mecanismos de recompensas e de reputação. Esta solução aplica os conceitos anteriormente referidos no contexto da organização empresarial, de forma a incentivar uma melhor organização e definição de objectivos dos vários trabalhadores.

1.2 Problema

Actualmente, as soluções de ferramentas de gestão de projecto utilizadas pelas grandes empresas partilham todas grandes falhas a nível dos canais de comunicação cliente - gestor, o que pode levar a desentendimentos relativamente a prazos, progressos e pagamentos. Estas falhas de comunicação devem-se à existência de uma barreira de separação entre os vários níveis de chefia num dado projecto, podendo existir dados que, mesmo sendo declarados e aprovados por um dos lados, não são imediatamente visíveis do outro, ou pecam em fiabilidade. O trabalho de realização de uma auditoria, quer interna, quer externa, também podia ser mais agilizado.

Da mesma forma, os valores dos vários KPI's considerados num dado contexto empresarial, devem ser traduzidos de forma transparente para o cliente, e muitas vezes tal não acontece, levando à diminuição do nível de confiança e, possivelmente à perda ou não renovação do contrato estabelecido.

1.3 Solução Proposta

Considerando os problemas apontados na sub-secção anterior, desenvolve-se uma ferramenta de gestão de projectos *blockchain-enabled* que, através da execução de *smart contracts*, agiliza o processo de gestão e automatiza a componente de gestão de risco, assim colmatando as falhas das restantes plataformas já existentes. Ao registar todas as interacções na *blockchain*, é também um dos objectivos agilizar o processo de auditoria.

Assim, para além das funcionalidades base que serão apresentadas em secções posteriores, é também possível fazer uma gestão de risco de uma macro ou micro tarefa tendo em conta os recursos e prazos associados e incluir via *Smart Contracts* certas funcionalidades de automação como a configuração de alertas de aproximação de prazos ou falta de recursos.

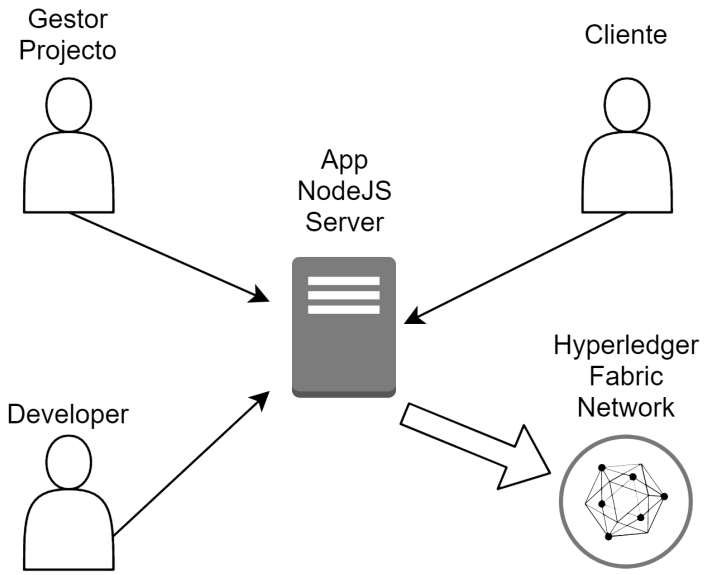

Figura 1.1: Solução proposta

1.4 Estrutura

Este documento irá estar organizado por capítulos na seguinte ordem:

- **Estado da Arte** - Apresentação e discussão de ferramentas e tecnologias relacionadas ao problema em questão.
- **Trabalhos Relacionados** - Exposição de projectos relacionados com o tema central da solução desenvolvida.
- **Desenvolvimento da Solução** - Detalhe e análise do ambiente, arquitectura, implementação e funcionamento da solução desenvolvida.
- **Testes e Avaliação** - Descrição da estratégia de validação adoptada, apresentação e discussão de resultados.
- **Conclusão** - Apresentação e discussão de conclusões finais sobre o trabalho realizado, possíveis futuros desenvolvimentos e outras considerações.

Capítulo 2

Estado da Arte

Este capítulo reúne informação relacionada com ferramentas e tecnologias paralelas à área de desenvolvimento desta dissertação. Assim, serão apresentadas e discutidas estas mesmas tecnologias, ferramentas e ideias que serviram de base ao desenvolvimento da solução proposta.

2.1 Tecnologia Blockchain

A tecnologia da Blockchain, que é mais conhecida por suportar uma das grandes criptomoedas, a Bitcoin [17], é um sistema de registo distribuído público/privado que visa manter a integridade de dados [27] através da descentralização. Adicionalmente, garante confiança (*trust*) em todas as interacções internodais dentro da rede. Podem-se considerar duas grandes categorias de redes blockchain - públicas e privadas.

No seu ambiente público ou "*permissionless*", como não existe um mecanismo de controlo de acesso, a blockchain permite que qualquer entidade independente submeta transacções sem ter uma identidade definida. Neste contexto, a única certeza é a garantia de imutabilidade do estado da blockchain. Dada a natureza deste tipo de rede, pela forma como escala, e de modo a mitigar a falta de confiança (*trust*) na rede, existe, por norma, um *token* de incentivo para os utilizadores com valor variável consoante o estado da *blockchain* a que pertence - uma criptomoeda. Em blockchains públicas, são usados algoritmos de *consensus* tais como o "Proof of Work"(PoW) ou o "Proof of Stake"(PoS) - ver secção 2.1.5.

Do outro lado do espectro, em blockchains privadas ou "*permissioned*", a identidade de cada participante é conhecida dado que é permitido acesso apenas a entidades autorizadas. Por serem redes mantidas por entidades privadas, nestes ambientes não está associado, por norma, um mecanismo de recompensa ou incentivo. Neste tipo de blockchains, são

normalmente aplicados algoritmos de *consensus* de Tolerância a Falhas Bizantinas (pBFT) [2] - ver secção 2.1.5.

Sabah, Mahdi & Majeeed [24] queriam categorizar blockchains para além do espectro público-privado, e assim, propuseram a seguinte classificação:

1. Blockchain orientada para Criptomoedas (C2C): Tipo reservado para pagamentos e descentralização de moeda, tomando como exemplo a Bitcoin Blockchain;

2. Blockchain Negócio - Criptomoeda (B2C): Adição de uma camada lógica no *ledger*. Para além de guardar transacções financeiras, o *ledger* executa programas na blockchain (*smart contracts*). Um exemplo deste tipo é a Ethereum Blockchain;

3. Blockchain orientada para Negócio (B2B): Não existe moeda neste tipo. Suporta a execução de *software* de *business logic*. Exemplo deste tipo é o projecto Hyperledger Fabric;

2.1.1 Estrutura da Blockchain

A blockchain é uma rede, um sistema distribuído, constituída por blocos que representam transacções submetidas. Assim, pode-se dividir a blockchain em 3 componentes: blocos, *ledger* e rede (*network*). As transacções são guardadas no *ledger* dentro de blocos, que variam em tamanho, longevidade e finalidade. Estes blocos, ligados por funções de *hash* criptográficas cujo *output* é directamente influenciado pelos dados do bloco anterior, formam a blockchain. Os nós que compõem a rede têm o papel de replicar o estado da blockchain - o *ledger* - através de mecanismos de *consensus*. A figura 2.1 representa uma sequência destes blocos.

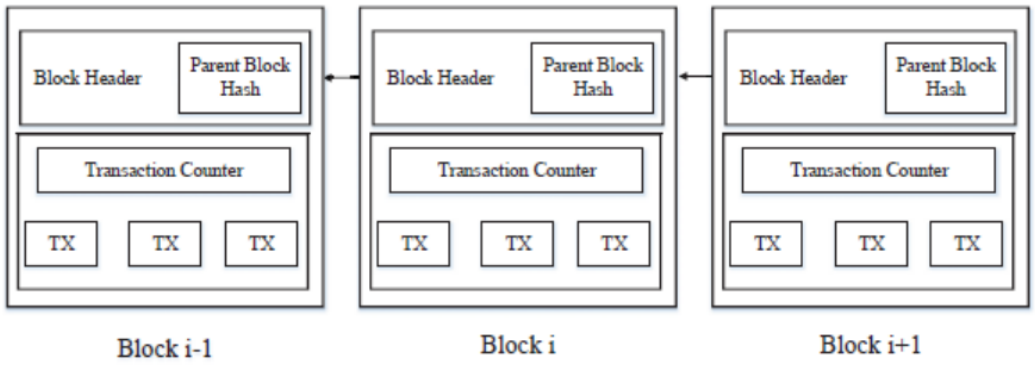

Figura 2.1: Exemplo de uma sequência de blocos [28]

2.1.2 Transacções

Transacções são parte fundamental num sistema blockchain. Uma transacção é a unidade representante de cada alteração de estado. Cada transacção é assinada com a chave

privada do participante. Após esta assinatura, é enviada para os restantes participantes para ser validada. Estas chaves (pública e privada) estão guardadas na *wallet* de cada participante. Depois desta validação, a transacção é incluída num bloco, juntamente com outras transacções. Cada bloco tem na sua estrutura, o *hash* do bloco anterior, formando assim uma corrente de blocos (*block-chain*). A alteração do conteúdo de um bloco implicaria fazer alterações no seu bloco "pai"e assim sucessivamente, o que seria virtualmente impossível dada a complexidade desta cadeia - natureza imutável da blockchain.

2.1.3 *Ledger*

O conceito de *ledger* distribuído (*Distributed Ledger Technology* - DLT) é um dos mais importantes neste contexto. Tipicamente, refere-se a uma base de dados partilhada, sincronizada e replicada por todos os participantes da blockchain. Para uma transacção ser guardada no *ledger*, a maioria dos participantes tem de dar e guardar o seu consentimento. Este registo é validado autonomamente, por processamento de transacções e pelo mecanismo de *consensus*. O primeiro bloco a ser gerado na blockchain é denominado bloco génesis (*genesis block*).

2.1.4 *Hashing*

Como foi referido anteriormente, cada bloco que forma a blockchain contém a *hash* do bloco anterior (*parent block*). Este valor é determinístico e é calculado através de funções criptográficas - por exemplo a Bitcoin recorre ao SHA-256. Alterações num bloco implicam a alteração da sua *hash*, afectando o bloco seguinte, visto que o seu endereço está nesse bloco. Existe, assim, uma dependência unidireccional entre blocos subsequentes. Desta forma, é uma das componentes que definem a natureza imutável, integra e autêntica da blockchain.

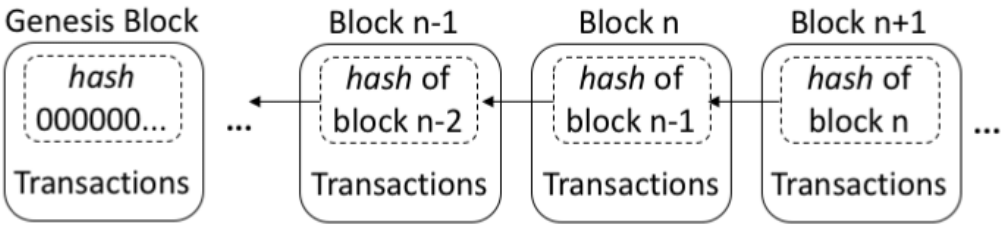

Figura 2.2: Relação de *hashing* entre blocos [19]

2.1.5 *Consensus*

Uma das características mais importantes em qualquer sistema distribuído é ser possível manter um único estado entre todos os nós do sistema. Assim surge o conceito de *consensus*. Se existe um estado uniforme que é mantido durante as interacções com os nós, então existe *consensus* no sistema.

Da mesma forma, no contexto da blockchain o *consensus* é identificado pelo estado do sistema depois da correcta execução de um determinado número de processos sobre o *ledger*. Para este efeito, a ordem pela qual cada bloco é adicionado ao *ledger* é acordada por todos os nós. Existem diversos algoritmos que suportam este mecanismo:

- **Proof of Work (PoW)** - Conceptualizado por Cynthia Dwork e Moni Naor em 1993 [6], é um protocolo que pretende limitar ataques *DDoS* (*Denial of Service*), *spam* de emails e quaisquer abusos que façam pressão na rede. Este protocolo requer que a entidade que submete o pedido realize algum tipo de trabalho (*work*) computacional pesado de forma a provar a sua intenção ao prestador de serviço. Variantes deste protocolo incluem protocolos *challenge-response* e *solution-verification*.

- **Proof of Stake (PoS)** - Em alternativa ao protocolo anterior, o **Proof of Stake (PoS)** confere poder de decisão sobre os participantes activos na rede [25]. Assim, numa rede gerida por PoS, nem todos podem entrar. Possuir moeda ou um tipo de depósito na rede é o factor que permite participar na validação de transacções e na criação de blocos. Contrariamente ao PoW, neste protocolo não é necessário poder computacional para resolver *challenges*. Uma das desvantagens deste mecanismo é que é possível ter o monopólio da rede caso um ou mais participantes reúnam a maioria dos activos. Para combater este problema, é necessário complementar este mecanismo com outro protocolo - por exemplo, o PoW.

- **Proof of Elapsed Time (PoET)** - Desenhado pela *Intel*, o PoET está dependente do set de instruções de CPU chamado *Intel Software Guard Extensions* (SGX) [23]. Assim, o PoET está desenhado como uma *lottery function* [14] que garante igualdade entre os participantes, igualdade de investimento (custo de controlar eleição de líder deve ser proporcional ao valor que é ganho por ser eleito) e facilidade de verificação - deve ser simples verificar que o líder foi legitimamente eleito. O fluxo de operação começa na requisição de um *wait time* aleatório devolvido por um enclave. De seguida, o *validator* com o tempo de espera menor é eleito líder. Ao mesmo tempo, uma função de verificação certifica que o tempo de espera foi efectivamente emitido por um *enclave* e o mesmo foi cumprido antes do inicio da liderança. Dadas estas características, este algoritmo de *consensus* é considerado bastante robusto.

- **Practical Byzantine Fault Tolerance (pBFT)** - Introduzido por Miguel Castro e Barbara Liskov em 1999[21], este protocolo, com base de implementação no BFT introduzido por Robert Shostak, Leslie Lamport e Marshall Pease [15], para além de ser capaz de tolerar falhas no sistema, também lida com dados corrompidos e ataques externos. No contexto do protocolo pBFT, todos os nós do sistema são ordenados, havendo um nó denominado líder. Cada ronda de *consensus* do pBFT dá pelo nome de *view* e está dividida em 4 fases:

  1. Cliente envia transacções para nó líder;

2. Nó líder envia o próximo bloco para os restantes nós;
3. Nós validam o bloco e emitem uma resposta;
4. Nó cliente espera receber $f+1$ respostas com o mesmo resultado (onde f se considera o número máximo de *faulty nodes*)

O nó líder é alterado em cada ronda e pode ser substituído prematuramente caso não transmita mensagens para os restantes nós. Assim, este protocolo garante *consensus* caso o número de nós *faulty* for inferior a 1/3 de todos os nós no sistema.

Em grande parte das blockchains públicas, são maioritariamente usados os algoritmos de *Proof-of-Work* (Bitcoin) e *Proof-of-Stake* (Ethereum) motivado pelo seu modelo cripto-económico. Em blockchains privadas, são normalmente aplicados mecanismos menos intensivos computacionalmente (em comparação ao *Proof-of-Work*, por exemplo), tal como o algoritmo *Practical Byzantine Fault Tolerance* (pBFT). Assim, há espaço para melhor escalabilidade e performance.

2.2 Plataformas de Blockchain

Existe actualmente um espectro bastante alargado de plataformas disponíveis e, assim sendo, iremos focar-nos nos casos de estudo mais relevantes de blockchains privadas e públicas. Dada a finalidade de uso empresarial deste projecto, vão ser estudadas ao detalhe soluções de blockchains privadas, assim permitindo gerir acessos e requisitos.

2.2.1 Blockchains Públicas - Ethereum

Em 2013, Vitalik Buterin publicou um *White Paper* [4] onde expandia o conceito da blockchain, vendo o protocolo da Bitcoin como um sistema de transição de estados. Assim, o Ethereum é uma plataforma descentralizada, construída sob uma rede *peer-to-peer* de máquinas virtuais *Turing*-completas que correm aplicações descentralizadas. Estas aplicações são, por norma, *smart contracts* que são guardados e executados na blockchain Ethereum. O fluxo de funcionamento desta plataforma segue os seguintes princípios:

1. Cada nó (máquina) da rede possui uma *Ethereum Virtual Machine* (EVM) que sincroniza com a blockchain e que fica também responsável por executar *smart contracts*;
2. O estado da rede é constituído por "contas"que efectuam transacções entre si, fazendo assim funcionar o sistema de transição de estados que envolve toda esta plataforma;
3. Os *smart contracts* são considerados contas autónomas, controladas pelo seu código interno que é executado quando existe um *trigger* - uma "mensagem". Uma "mensagem"é tal como uma transacção, mas entre dois *smart contracts*;

4. Transacções são mensagens enviadas entre contas externas (controladas por chaves privadas). No caso especifico da Ethereum, as transacções incluem 2 campos extra: **STARTGAS** e **GASPRICE**. Estes campos são usados para prevenir uso computacional excessivo tanto acidental como por parte de atacantes, atribuindo custos operacionais aos recursos consumidos por cada operação;

O Ethereum foi pensado para uma aplicação Business-to-Consumer (B2C), e assim, sendo uma blockchain pública, não se encaixa nos moldes e requisitos necessários para uma aplicação *enterprise*. No sub-capitulo seguinte iremos estudar uma alternativa de blockchains privadas.

2.2.2 Blockchains Privadas - Hyperledger Fabric

O projecto Hyperledger começou em 2015 e está a ser um esforço conjunto de diversas empresas que têm como objectivo criar uma plataforma de blockchain *open-source* que se pudesse tornar no *standard* da indústria. Actualmente, o projecto está a ser mantido pela Linux Foundation e tem vindo a crescer bastante.

A filosofia de design por detrás do desenvolvimento do Hyperledger é para servir para bastantes casos de uso. Assim, todos os projectos Hyperledger são: [3]

- Modulares;
- Altamente seguros;
- Interoperáveis;
- Agnósticos em relação a criptomoedas;
- Disponíveis via APIs;

Um destes projectos é de grande interesse para o *use-case* da solução desenvolvida nesta dissertação: Hyperledger Fabric.

O Hyperledger Fabric segue a filosofia de design dos restantes projectos, nomeadamente, os vários componentes (mecanismos de *consensus*, serviços de *membership*) são *plug-and-play*. Adicionalmente, corre sobre um ambiente de containers e aloja *smart contracts* sob a forma de *"chaincode"*, actualmente suportando Go, Java e JavaScript. De igual forma, suporta vários algoritmos de *consensus*, não depende de nenhuma cripto-moeda e também suporta canais (*channels*) que permitem a um grupo de participantes criar um *ledger* separado. A rede é constituída por conjuntos de *peers* - denominados organizações.

Também é caracterizado por disponibilizar uma infraestrutura do mecanismo de *consensus* composta por uma combinação de serviços de *endorsing* e de *ordering*. Um certo conjunto de *peer nodes* [11] é escolhido para um grupo de *endorsers*. Estes nós são os primeiros a receber e a verificar transacções. A cópia do *ledger* disponível localmente nestes nós serve de base para executar e armazenar os resultados destas transacções que serão

posteriormente avaliados e aglomerados em *endorsements*, sendo enviados de volta para o cliente.

De forma complementar, existem políticas de *endorsement* que definem que transacções são válidas, por via de operadores lógicos. É registado se os *endorsements* recebidos de volta no cliente foram validados ou não pelos nós *endorsers* e se esses nós têm identidades válidas. Se as políticas forem cumpridas, as transacções são declaradas válidas e são enviadas para o *ledger*. Adicionalmente, é guardada uma cópia destas transacções no *ledger* dos *peer nodes* antes de ser submetida no *ledger* principal.

Para finalizar, existem *ordering peers* - nós que recebem *endorsements* e verificam a ordem cronológica, as transacções e os certificados dos nós *endorsers* e por fim, se tudo for válido, submetem a transacção ao *ledger*, transmitindo o resultado desta validação a todos os *peers*. Este serviço - *ordering service* - é mantido por um algoritmo de *consensus*, o *Raft*. [10] A figura 2.3 ilustra o fluxo acima descrito.

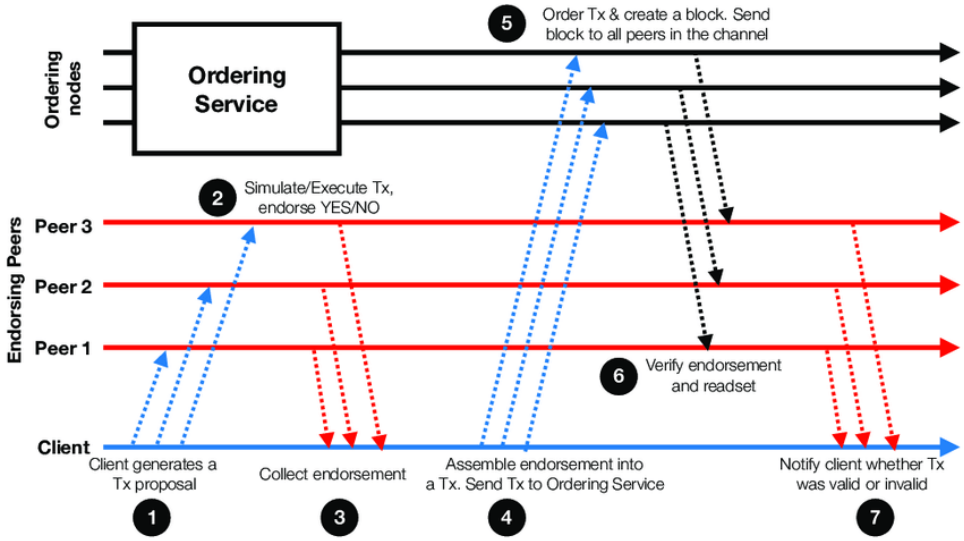

Figura 2.3: Fluxo do *Ordering Service* [22]

Para além de uma cópia do *ledger*, cada *peer* guarda uma cópia do estado - *world state* - do *ledger*, uma base de dados que regista os valores de um conjunto de estados do *ledger*. Estes estados são representados por pares chave-valor. O *world state* sofre modificações com bastante frequência, com adição, alteração e remoção de estados. Esta estrutura permite que *chaincode* tenha acesso ao valor actual do estado ao invés de o calcular percorrendo o *log* de transacções. Actualmente, o Hyperledger Fabric suporta duas opções para a base de dados *world state*: LevelDB e CouchDB. A primeira opção - por defeito - é suficiente nos casos em que cada estado são pares chave-valor. Por outro lado, o CouchDB permite que os mesmos estados estejam estruturados como documentos JSON, possibilitando desta forma fazer *queries* às chaves mas também aos valores dos dados e usar índices para suportar as *queries*, sendo mais flexível e eficaz em grandes volumes de dados.

Por fim, a *Certificate Authority* do Hyperledger Fabric é responsável pela gestão de identidades e certificados dos participantes do sistema necessários para a participação na rede. Esta informação é mantida, por defeito, numa base de dados SQLite partilhada por todos os servidores da CA.

Tendo em conta a informação e todas as características que foram referidas acima, assume-se o projecto Hyperledger Fabric como a plataforma escolhida para o desenvolvimento, encaixando-se perfeitamente na solução desenvolvida.

Capítulo 3

Trabalhos Relacionados

Neste capítulo vão ser abordados alguns projectos relacionados em áreas paralelas à matéria de desenvolvimento desta dissertação, desde a gestão de projectos em diversas indústrias à gestão da *supply-chain*.

3.1 Indústria da Construção

A partilha de informação, automação de processos e confiança entre parceiros fazem parte de um conjunto de desafios que a gestão de projectos na indústria da construção enfrenta actualmente. Jun Wang et al. [26] estudaram o efeito que a introdução da blockchain teria nesse ambiente e possíveis aplicações concretas. Os autores separam as várias aplicações em três tipos de categorias:

1. Verificação de autenticidade de documentos - Com a aplicação desta tecnologia, seria possível armazenar todos os documentos no *ledger* e saber quem e quando se fez cada alteração;

2. Automação de pagamentos e procurações - Pagamentos, aluguer de equipamentos e venda de casas podem ser agilizados, poupando tempo e reduzindo custos;

3. Rastreamento e transparência em redes de fornecimento (*supply-chains*) - Visibilidade de todas as transacções na blockchain facilita o rastreamento de produtos e confirma a autenticidade dos mesmos;

Por fim propõem três exemplos de aplicações concretas - gestão de contratos, gestão de redes de fornecimento e aluguer de equipamento *blockchain-enabled*. Para resolver o primeiro *use-case*, e o mais relevante para a matéria desta dissertação, os autores avançam com uma proposta de implementação que melhora a eficiência dos processos administrativos, assegura confiança entre os vários intervenientes do projecto e extingue problemas

de pagamento e fluidez de dinheiro, integrando esta lógica em *smart contracts* (ver figura 3.1) e assim protegendo todas as partes do negócio. Dão como exemplo o pagamento automático quando se atinge um objectivo de construção, activando também todos os pagamentos entre cada uma das partes envolvidas no projecto, não havendo assim atrasos nem transacções duplicadas.

SOLIDITY CONTRACT SOURCE CODE

```
pragma solidity ^0.4.2;


contract MyContract {
    /* Constructor */
    address public contractor;
    uint256 public allowance;
    uint256 public temperature;

    mapping (address => uint) public balanceOf;
    event Transfer(address _from, address _to, uint value);


    function token(uint supply) {
        balanceOf[msg.sender] = supply;
    }

    function transfer (address contractor, uint256 allowance) {
        if (temperature < 40) throw;
        if (balanceOf[msg.sender] < allowance) throw;
        if (balanceOf[contractor] + allowance < balanceOf[contractor]) throw;

        balanceOf[msg.sender] -= allowance;
        balanceOf[contractor] += allowance;
        Transfer (msg.sender, contractor, allowance);

    }
}
```

Figura 3.1: Exemplo de um *smart contract* [26]

Abordando as duas últimas aplicações, nomeadamente a gestão de redes de fornecimento (*supply-chains*) e o aluguer de equipamento, Jun Wang et al. [26] argumentam que o uso da blockchain também é bastante benéfico. Para o primeiro caso apontam que todos os dados relativos à encomenda e envio destes produtos é carregado para a blockchain, tornando assim este processo mais transparente e alvo de certificação. Em relação ao segundo caso, identificam o elevado custo inerente a actos de aluguer de equipamentos, tanto em relação às máquinas como em relação a reparações relacionadas, e admitem que a disponibilização e registo destes equipamentos na blockchain viria a diminuir o impacto destes problemas. Assim, seria possível seleccionar a modalidade de aluguer - regime temporal ou financeiro, por exemplo - optar por várias opções de seguro e realizar o pagamento automaticamente.

Finalmente, discutem a grande redução de custos e tempo que a automação de processos pode vir a trazer, mas sem antes referir três desafios que se impõem. O primeiro é um desafio técnico e está relacionado com limitações da blockchain, tais como a capacidade de tráfego (*throughput*) que teoricamente está maximizado nas sete transacções por segundo, na latência (tempo de processamento de 10 minutos por bloco) e na largura de

banda disponível para descarregar a blockchain. O segundo desafio está ligado à natureza própria da indústria e aos seus antigos costumes e o quão difícil seria transpor toda a lógica de negócio para a blockchain. Adicionalmente, consideram o elevado investimento inicial que seria necessário caso fosse escolhida uma solução de blockchain privada. O último desafio prende-se a factores humanos. A falta de conhecimento e interesse da grande generalidade das pessoas dificulta a difusão desta tecnologia e devido a isso ainda há muita relutância em partilhar detalhes confidenciais e privados na blockchain.

Para concluir, os autores voltam a frisar que serviços de notariado, processamento e autenticação de documentos, automação de pagamentos e transparência em ambiente industrial são as grandes aplicações que futuramente poderão vir a agilizar este ramo da indústria.

3.2 Blockchain aplicada à Gestão de Projectos

Imanol Pastor et. al [20] são os autores de um estudo que tem como objectivo encontrar onde e como integrar esta tecnologia na área da gestão de projectos.

Começam por dividir a gestão de projectos (PM), de acordo com o PMBOK [13], em duas abordagens diferentes: *process groups* e *knowledge areas*.

Analisando as vantagens de integração da blockchain pelos cinco *process groups* definidos pelo PMBOK [13] apresentam as seguintes vantagens:

1. **Iniciação**: Fase de definição do projecto
    a) Rede de Confiança - A tecnologia da blockchain pode ser usada para estabelecer uma rede de confiança (*trusted network*) entre todos os *stakeholders* envolvidos e onde se partilha toda a informação relacionada com o projecto.
    b) Ferramenta de acordos suportada por *smart contracts* - Vem substituir a necessidade de realização de várias reuniões, simplificando e automatizando este processo.
    c) Base de dados de registo de especificações - Alterações às especificações iniciais são registadas de forma a prever e assumir consequências e planos de actuação.
    d) Ecossistema de suporte financeiro - Tem como alvo membros do projecto que sejam financeiramente frágeis.
    e) Gestão de Risco - Ferramenta de seguro entre os envolvidos sem necessidade de intermediários.
2. **Planeamento**: Fase de desenvolvimento e acordo do plano de gestão
    a) Ferramenta de acordos - Tendo em conta custos e tempo que foram previamente acordados.
    b) Definição de limites - Utilização de *smart contracts* para incluir acções que são disparadas na conclusão de cada tarefa, com incentivos ou penalizações.

   c) Identificação de risco - Inclusão do risco de cada tarefa na blockchain e a influência que terá em outras tarefas, dinamicamente alterando prioridades dessas mesmas tarefas.

3. **Execução**: Fase de actualização do planeamento. Nesta fase a blockchain pode ser aplicada em:

   a) Aumentar flexibilidade em tomada de decisões devido à remoção das burocracias.

   b) Facilitar efectivação de contractos e acordos.

   c) Gestão de fluxo de trabalho (*workflow*), ao calcular a dependência entre tarefas.

   d) Gestão de requisitos e os impactos no projecto em relação a custos, tempo e risco.

4. **Monitorização e Controlo**: Nesta fase, a capacidade da blockchain pode ser utilizada para fazer a validação e teste de cada componente e para ir gerindo em *real time* toda a informação relativa ao projecto.

5. **Conclusão**: Fase de fecho do projecto, e de todas as obrigações contractuais

   a) Ferramenta de cumprimento de contractos suportada por *smart contracts* - Validação do cumprimento de todos os requisitos é chave para o fecho do projecto.

   b) Pode ser uma boa fonte de análise para futuros projectos.

   c) Base de dados de auditoria - Os dados recolhidos e armazenados na blockchain podem ser usados para uma auditoria ou avaliação externa.

Os autores vêem as vantagens do uso da blockchain durante a realização de um projecto bastante valiosas e que se podem estender para além do mesmo, passando pela manutenção e garantia do produto.

Da mesma forma, foram identificadas as vantagens que a introdução desta tecnologia teria nas dez *knowledge areas* usadas frequentemente em projectos:

1. **Gestão de Integração** - Blockchain usada como uma base de dados de informação partilhada entre todos os *stakeholders*, servindo também para monitorização do projecto.

2. **Gestão de Escopo** - Registo e validação de requisitos e funcionalidades, com burocracia e tempo despendido reduzido.

3. **Gestão de Tempo** - Registo do mapa temporal na blockchain, tal como alterações ao mesmo. Recorrer a *smart contracts* para orientar e regular *deadlines*.

4. **Gestão de Custo** - Registo das despesas na blockchain, tornando esta informação transparente e disponível para todos os intervenientes.

5. **Gestão de Qualidade** - Registo dos testes e validações necessárias para a conclusão do projecto. Recorrer a *smart contracts* para automatizar execução sequencial de testes interdependentes.

6. **Gestão de Recursos Humanos** - Possíveis aplicações na área de gestão de identidade.

7. **Gestão de Risco** - Registo de todos os riscos inerentes a uma tarefa e as suas implicações. Recorrer a *smart contracts* para automatizar certas acções a tomar caso seja necessário detectado um risco para o projecto.

8. **Gestão de *stakeholders*** - Integração dos *stakeholders* em vários níveis da rede e disponibilizando informação transparente e passível de ser discutida.

Discutidas todas estas vantagens da introdução da blockchain na gestão de projectos, Imanol Pastor et. al [20] revelam os desafios que nos separam desta realidade. Para começar, é imperativo encontrar soluções para o elevado nível de complexidade que esta tecnologia nos apresenta, tanto em relação a poder computacional como em escalabilidade do sistema. Da mesma forma, não será fácil encontrar indivíduos que validem as transacções no sistema, dado que este não tem sistema de recompensas, não promovendo a sua actuação. Em termos culturais, será uma nova ferramenta a utilizar e a integrar no *workflow* existente, o que certamente causará algum receio. Ainda dentro deste espectro, a falta de *standards* nos protocolos a integrar na blockchain e o elevado custo e tempo que será necessário irá dificultar a sua integração no modelo actual da gestão de projectos. Os autores avançam ainda com a possibilidade da adopção de blockchains privadas numa fase inicial, de forma a mitigar estes desafios.

Este estudo revela-se bastante interessante e relevante para o desenvolvimento da solução proposta nesta dissertação, não só pelas ideias nele apresentadas mas também pela discussão dos possíveis desafios a enfrentar, que foram certamente alvo de análise e de melhorias ao longo do desenvolvimento deste trabalho.

Capítulo 4

Desenvolvimento da Solução

Neste capítulo vão ser abordados todos os passos do desenvolvimento da solução, passando pela configuração da rede Hyperledger Fabric, implementação de chaincode e da aplicação cliente, sendo explicadas as decisões de implementação e pormenores relevantes em cada secção.

4.1 Rede Hyperledger Fabric

Para o desenvolvimento desta solução, a primeira dependência prende-se com a existência de uma rede Fabric. Para o efeito, decidiu-se usar uma ferramenta pertencente ao projecto Hyperledger Labs [12] - Minifabric [16]. Esta ferramenta permite criar uma rede Fabric de forma simples, mas completa, com suporte a todas as capacidades do Hyperledger Fabric:

- Configuração costumizada da rede, com possibilidade de extensão (adicionar novas organizações)
- *Channel query*, *create*, *join* e *update*
- Chaincode *install*, *approve*, *instantiation*, *invoke*, *query* e colecções privadas
- *Block queries*

Através de um conjunto de comandos é possível fazer todas as operações necessárias para o desenvolvimento da solução apresentada, incluindo a criação de organizações, criação da *network* e a gestão do lifecycle do chaincode - *install*, *approve* e *instantiate*.

Assim, foi necessário seguir a seguinte ordem de processos:

1. Configurar a rede Fabric

2. Criar um *channel*
3. Adicionar *peers* ao *channel*
4. Instalar o *chaincode* desenvolvido nos *peers*
5. Aprovar o chaincode - acção necessária a partir da versão 2.0 do Hyperledger Fabric
6. *Commit chaincode*

4.1.1 Configuração de Rede

No contexto do desenvolvimento da solução, pretende-se que esta ferramenta seja apenas de uso interno, sendo assim essencial a criação de uma organização que represente a empresa detentora da ferramenta - *org0*.

Tendo em conta que todas as transacções estariam, logicamente, contidas no mesmo contexto, apenas se considerou a criação de um *channel*. Neste *channel* foi instalado o *chaincode* desenvolvido no contexto desta solução.

A organização criada é constituída por dois *peers* e o *ordering service* é constituído por três nós *orderers*, relativos à única organização criada.

O *Minifabric* suporta especificações de rede costumizadas, definidas no ficheiro **spec.yaml**. Durante o desenvolvimento foi usada a seguinte configuração:

Listagem 4.1: Configuração da rede Fabric

```
fabric:
  cas:
  - "ca1.org0.example.com"
  peers:
  - "peer1.org0.example.com"
  - "peer2.org0.example.com"
  orderers:
  - "orderer1.example.com"
  - "orderer2.example.com"
  - "orderer3.example.com"
  settings:
    ca:
      FABRIC_LOGGING_SPEC: DEBUG
    peer:
      FABRIC_LOGGING_SPEC: DEBUG
    orderer:
      FABRIC_LOGGING_SPEC: DEBUG
```

De seguida, a rede Fabric é gerada pela execução do seguinte comando:

Listagem 4.2: Comando de criação inicial da rede Fabric

```
minifab up -e -s couchdb
```

Este comando, para além de especificar a opção de exposição dos *ports* e o tipo de *state dabatase* - CouchDB, neste caso -, executa automaticamente todos os passos que seriam necessariamente executados manualmente. Numa primeira fase, gera todos os certificados necessários para a configuração inicial da rede, com recurso ao *toolkit openssl* - em vez de utilizar o *cryptogen* disponibilizado pelo Hyperledger Fabric, para um maior controlo destes certificados. Nesta fase é possível adaptar o ficheiro YAML **crypto-config.yaml**, ainda que este já venha completo tendo em conta a configuração detalhada em 4.1. Findo este processo, resulta um conjunto de certificados e chaves de cada componente da rede - CA, organizações, *peers* e *orderers*.

O segundo processo está relacionado com a criação de *channel artifacts*, incluindo, entre outros, o bloco *genesis* usado para inicializar o *ordering service*. A definição dos artefactos a gerar é registada num segundo ficheiro YAML - **configtx.yaml**.

Listagem 4.3: Definição inicial da organização

```
- &org0-example-com
    Name: org0-example-com
    ID: org0-example-com
    MSPDir: keyfiles/peerOrganizations/org0.example.com/msp
    Policies:
      Readers:
        Type: Signature
        Rule: "OR('org0-example-com.admin', 'org0-example-com.peer', 'org0-example-com.
            ↪ client')"
      Writers:
        Type: Signature
        Rule: "OR('org0-example-com.admin', 'org0-example-com.client')"
      Admins:
        Type: Signature
        Rule: "OR('org0-example-com.admin')"
      Endorsement:
        Type: Signature
        Rule: "OR('org0-example-com.peer')"

    AnchorPeers:
    - Host: 192.168.1.165
      Port: 7002
    - Host: 192.168.1.165
      Port: 7003
```

Este ficheiro tem informação sobre os vários componentes da rede. Com mais relevância, em 4.3 está configurado os *anchor peers* da organização, e o conjunto de regras de gestão da mesma. Estas regras (*policies*) definem como é que se chega a um acordo em relação a mudanças na rede, *channel*, ou *smart contract*.

Por sua vez, em 4.4 é definido o *ordering service* com a especificação do tipo de serviço - *Raft*, neste caso - e o conjunto de regras do serviço:

Listagem 4.4: Definição inicial do *ordering service*

```
Orderer: &OrdererDefaults
  OrdererType: etcdraft

  BatchTimeout: 2s

  BatchSize:
    MaxMessageCount: 10
    AbsoluteMaxBytes: 99 MB
    PreferredMaxBytes: 512 KB

  Addresses:
  - 192.168.1.165:7010
  - 192.168.1.165:7011
  - 192.168.1.165:7012
  EtcdRaft:
    Consenters:
    - Host: 192.168.1.165
      Port: 7010
      ClientTLSCert: keyfiles/ordererOrganizations/example.com/orderers/orderer1.example.
          ↪ com/tls/server.crt
      ServerTLSCert: keyfiles/ordererOrganizations/example.com/orderers/orderer1.example.
          ↪ com/tls/server.crt
  [...]
  Organizations:
  Policies:
    Readers:
      Type: ImplicitMeta
      Rule: "ANY Readers"
    Writers:
      Type: ImplicitMeta
      Rule: "ANY Writers"
    Admins:
      Type: ImplicitMeta
      Rule: "MAJORITY Admins"
    BlockValidation:
      Type: ImplicitMeta
      Rule: "ANY Writers"
```

Finalmente, em 4.5 é especificado o bloco *genesis* enviado para os *ordering nodes* e são configuradas as capacidades da rede. Durante o desenvolvimento desta solução foi usada a versão 2.2.3 do Hyperledger Fabric, e assim, especifica-se o suporte às capacidades 2.0 do Hyperledger:

- Application V2_0: Activa o novo *lifecycle* do *chaincode*;
- Channel V2_0: Não existem novas capacidades a este nível;
- Orderer V2_0: Muda a forma como as transacções de criação de *channels* são validadas.

Listagem 4.5: Definição inicial do bloco genesis

```
Capabilities:
  Channel: &ChannelCapabilities
    V2_0: true
  Orderer: &OrdererCapabilities
    V2_0: true
  Application: &ApplicationCapabilities
    V2_0: true

   Profiles:
  OrgChannel:
    Consortium: SampleConsortium
    <<: *ChannelDefaults
    Application:
      <<: *ApplicationDefaults
      Organizations:
      - *org0-example-com
      Capabilities:
        <<: *ApplicationCapabilities

  OrdererGenesis:
    <<: *ChannelDefaults
    Capabilities:
      <<: *ChannelCapabilities
    Orderer:
      <<: *OrdererDefaults
      Organizations:
      - *example-com
      Capabilities:
        <<: *OrdererCapabilities
    Application:
      <<: *ApplicationDefaults
      Organizations:
      - <<: *example-com
    Consortiums:
      SampleConsortium:
        Organizations:
        - *org0-example-com
```

Em seguida, é lançada a rede de *docker containers* onde irá estar instalada a rede Fabric. Essencialmente, são lançados *containers* para cada nó constituinte da rede, comunicando entre si através de uma *virtual network*. Assim, são lançados *containers* para cada *peer*, *orderer*, nó da *CA*, e nó do *couchdb*.

Depois de estar configurada e lançada a rede Fabric, é necessário criar o *channel* de modo a permitir comunicação entre todos os *peers*. Primeiramente, é gerada a transação de criação do *channel* (**mychannel.tx**) que irá ser utilizada no comando seguinte - `peer channel create` - e após serem exportadas as variáveis necessárias é finalmente executado o comando de criação do *channel*. A criação do *channel* é executada apenas uma vez,

num *peer* seleccionado aleatoriamente 4.6.

Listagem 4.6: Criação de *channel*

```
# Script para criar o channel block 0 e de seguida criar o channel
cp $FABRIC_CFG_PATH/core.yaml /vars/core.yaml
cd /vars
export FABRIC_CFG_PATH=/vars
configtxgen -profile OrgChannel \
  -outputCreateChannelTx mychannel.tx -channelID mychannel

export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_ID=cli
export CORE_PEER_ADDRESS=192.168.1.165:7003
export CORE_PEER_TLS_ROOTCERT_FILE=/vars/keyfiles/peerOrganizations/org0.example.com/
    ↪ peers/peer2.org0.example.com/tls/ca.crt
export CORE_PEER_LOCALMSPID=org0-example-com
export CORE_PEER_MSPCONFIGPATH=/vars/keyfiles/peerOrganizations/org0.example.com/users/
    ↪ Admin@org0.example.com/msp
export ORDERER_ADDRESS=192.168.1.165:7010
export ORDERER_TLS_CA=/vars/keyfiles/ordererOrganizations/example.com/orderers/orderer1.
    ↪ example.com/tls/ca.crt
peer channel create -c mychannel -f mychannel.tx -o $ORDERER_ADDRESS \
  --cafile $ORDERER_TLS_CA --tls
```

De seguida, cada *peer* é registado no *channel*, ao escrever no *ledger* o bloco resultante do comando anterior. O script seguinte 4.7 é exemplo deste procedimento para um dos *peers* - peer1.

Listagem 4.7: Ligação de *peers* ao *channel*

```
# Script para associar um peer a um channel
export CORE_PEER_TLS_ENABLED=true
export CORE_PEER_ID=cli
export CORE_PEER_ADDRESS=192.168.1.165:7002
export CORE_PEER_TLS_ROOTCERT_FILE=/vars/keyfiles/peerOrganizations/org0.example.com/
    ↪ peers/peer1.org0.example.com/tls/ca.crt
export CORE_PEER_LOCALMSPID=org0-example-com
export CORE_PEER_MSPCONFIGPATH=/vars/keyfiles/peerOrganizations/org0.example.com/users/
    ↪ Admin@org0.example.com/msp
export ORDERER_ADDRESS=192.168.1.165:7010
export ORDERER_TLS_CA=/vars/keyfiles/ordererOrganizations/example.com/orderers/orderer1.
    ↪ example.com/tls/ca.crt
if [ ! -f "mychannel.genesis.block" ]; then
  peer channel fetch oldest -o $ORDERER_ADDRESS --cafile $ORDERER_TLS_CA \
  --tls -c mychannel /vars/mychannel.genesis.block
fi
peer channel join -b /vars/mychannel.genesis.block \
  -o $ORDERER_ADDRESS --cafile $ORDERER_TLS_CA --tls
```

4.2 Chaincode

Neste passo, estão configurados e lançados os *containers* da rede Fabric. Assim, resta instalar o *chaincode* desenvolvido no contexto desta solução - **projectManagementCC**. O processo de instalação do *chaincode* segue um fluxo bem definido, denominado *lifecycle* [7]. Considerando a versão 2.2.3 do Hyperledger Fabric que foi escolhida para o desenvolvimento da solução, existem um conjunto de passos a seguir para instalar *chaincode* - ou fazer *upgrade* - num determinado *channel*:

1. Compilar um pacote *.tar* que contenha o *chaincode* e um ficheiro *metadata.json* que inclui informações sobre a linguagem, caminhos e *label* do pacote gerado;
2. Instalar o *chaincode* em todos os *peers* que irão executar e dar *endorse* a transacções;
3. Aprovar a definição do *chaincode* a nível da organização;
4. Fazer *commit* da definição do *chaincode* para o *channel*;
5. Inicializar o *chaincode*, correndo uma função (**Init**), gerando assim um estado inicial;

Listagem 4.8: Instalação do *chaincode*

```
1 minifab install -v 1.0.0 -n projectManagementCC -l node
2 minifab approve
3 minifab commit
4 # Necessário correr o initialize para cada smart contract
5 minifab initialize -p '"projectAssets:InitLedger"'
```

A sequência de comandos em 4.8 é executada na primeira instalação do *chaincode* e após cada actualização do código do mesmo - *chaincode upgrade* - incrementando, obviamente, a versão. Após a execução destes comandos, e aquando da primeira invocação de uma função do *chaincode*, é lançado um novo *container* por cada *peer* relativo a esse *chaincode*.

4.2.1 Desenvolvimento de *smart contracts*

No contexto desta solução, o *chaincode* desenvolvido é composto por vários *smart contracts*. Estes *smart contracts* foram escritos em Node.js, suportados pelo SDK disponibilizado pelo Hyperledger Fabric para este efeito.

Tendo em conta as diversas componentes da ferramenta desenvolvida, o *chaincode* está dividido em cinco *smart contracts*, unicamente identificados pelo seu *namespace*:

- **projectAssets** - Contém funções de lógica de negócio relacionadas com a gestão de objectos do tipo *project*.
- **userAssets** - Contém funções relacionadas com a gestão de objectos do tipo *user*.

- **taskAssets** - Contém funções de lógica de negócio relacionadas com a gestão de objectos do tipo *task*.
- **timesheetAssets** - Contém funções de lógica de negócio relacionadas com a gestão de objectos do tipo *timesheet*.
- **riskAssets** - Contém funções de lógica de negócio relacionadas com a gestão de objectos do tipo *risk*.

Apesar desta divisão, não implica isolamento completo da lógica. Por exemplo, certas funções do *smart contract* **projectAssets** dependem de funções definidas no contrato **riskAssets**. Em todos os *smart contracts*, foi implementado suporte a lógica *CRUD*, permitindo assim criar, listar, actualizar e apagar objectos de cada um dos tipos. Ao mesmo tempo é aplicada lógica relativa a gestão de risco - através de objectos do tipo *risk* - na maioria das operações críticas. Assim, tomando como exemplo a actualização das datas de conclusão de um projecto, é registado o evento de risco e são notificados todos os utilizadores do tipo *manager* e *developer* - dois dos quatro tipos de utilizadores que podem existir, como iremos ver em secção posterior.

Relembrando a execução de 4.8, no último comando foi inicializado o *smart contract* **projectAssets** através da função **InitLedger**.

Nesta função observamos a definição de dois objectos - *assets* - do tipo *project* que são adicionados ao *ledger* durante a fase de *bootstrap* do *chaincode*. Todos os tipos de objectos têm uma estrutura bem definida, sendo sempre identificados pelo campo *ID*. De realçar a existência do atributo *docType*, usado para diferenciar *queries* aos vários tipos de *assets* na *state database*.

Listagem 4.9: Função InitLedger

```
async InitLedger(ctx) {
       const assets = [
          {
             ID: 'project01',
             Name: 'Example Project 1',
             Manager: 'Manager1',
             Created: '2020-12-21T10:09:08Z',
             DueDate: '2021-03-21T10:00:00Z',
             Budget: '30000',
             Cost: '2500',
             Status: 'ongoing'
          },
          {
             ID: 'project02',
             Name: 'Example Project 2',
             Manager: 'Manager2',
             Created: '2020-12-21T10:09:08Z',
             DueDate: '2021-06-12T10:00:00Z',
             Budget: '150000',
```

```
20              Cost: '145000',
21              Status: 'maintenance'
22          }
23      ];
24      for (const asset of assets) {
25          asset.docType = 'project';
26          await ctx.stub.putState(asset.ID, Buffer.from(JSON.stringify(asset)));
27          console.info(`Asset ${asset.ID} initialized`);
28      }
29  }
```

Devido ao elevado volume de código do *chaincode*, irão ser apontados os detalhes mais relevantes:

1. Objectos do tipo *user* são gerados no momento do registo de um novo utilizador na aplicação. Assim, cada objecto segue a seguinte estrutura:

Listagem 4.10: Estrutura de objecto do tipo *user*

```
1 ID: id,
2 UserName: username,
3 # Nivel do utilizador (relacionado com as permissoes na aplicaçao)
4 UserLvl: userlvl,
5 Role: role,
6 # Hash da password definida pelo utilizador durante o registo
7 Passhash: passhash,
8 Created: created
```

2. As *tasks* são definidas por projecto e têm sempre um *assignee* e um *reporter* - habitualmente, um utilizador *Developer* e um *Manager*, respectivamente. Para além dos campos habituais de nome, descrição e estado da tarefa, é possível também definir se existe alguma dependência sobre outra tarefa. Estes dados são importantes e considerados na lógica de gestão de risco.

3. Cada objecto *timesheet* representa uma folha de horas submetida por um utilizador do tipo *developer* num determinado dia e regista a informação de que projectos foram abordados, que tarefas - *tasks* - foram desenvolvidas e durante quantas horas. Por fim, existe um campo *Approved* que indica se essa *timesheet* foi aprovada por um *Manager* e se pode, assim, ser contabilizada no cálculo de horas do projecto.

   Foi implementada lógica com suporte para a apresentação de métricas relativas a *timesheets*, reportando o tempo despendido, por tarefa, em cada projecto e *timesheets* de cada utilizador do tipo *developer* por projecto.

4. No *smart contract riskAssets*, está contida a lógica de negócio relativa à análise de risco de cada projecto. Assim, estão definidas uma série de funções que, entre outros, permitem verificar se existem projectos atrasados, tarefas que possivelmente atrasarão o projecto e tarefas atrasadas e aproximação do limite definido de *budget* para o

projecto. Ainda é possível receber alertas caso existam utilizadores *developers* sem tarefas associadas - ver 4.11 - e também, pelo contrário, caso estejam sobrecarregados e exista falta de recursos humanos no projecto. Ainda é possível correr uma *suite* de análise a todos os projectos existentes através da função `RunRiskAssessmentScan`.

Listagem 4.11: Função de verificação de tasks associadas

```
async IssueDevNoTasksWarning(ctx){
       let taskAssets = new TaskAssets();
       let userAssets = new UserAssets();

       let users = await userAssets.GetAllUsers(ctx);
       users = JSON.parse(users);
       for (let index = 0; index < users.length; index++) {
          const user = users[index];
          let tasks = await taskAssets.GetAllAssigneeTasks(ctx, user.Record.ID);
          tasks = JSON.parse(tasks);
          let now = ctx.stub.getDateTimestamp();
          if (tasks.length <= 0 && user.Record.Role === 'Developer') {
             const riskAsset = {
                ID: 'warn_notasks_'+user.Record.ID,
                Name: user.Record.UserName+'␣has␣no␣assigned␣tasks',
                Assignee: user.Record.UserName,
                Issued: now.toISOString(),
                Type: 'warning',
                Dismissed: false
             };
             riskAsset.docType = 'risk';
             await ctx.stub.putState(riskAsset.ID, Buffer.from(JSON.stringify(
                 ↪ riskAsset)));
          }
       }
   }
```

4.2.2 CouchDB - Desenvolvimento de Índices

Como foi referido anteriormente, foi escolhido o CouchDB como *state database*. A principal vantagem prende-se com a maior flexibilidade e robustez na construção das *queries* (*rich queries*), sendo recomendado, no entanto, a definição de índices - *indexes* - para tomar partido de uma maior performance. Assim, paralelamente ao desenvolvimento dos *smart contracts*, foram sendo desenvolvidos índices de suporte às *queries* mais recorrentes no *chaincode*. Para a definição de um índice é necessário três tipos de informação: *fields*, *name* e *type*. O atributo *fields* define os campos que se pretende indexar. Os atributos *name* e *type*, representam, respectivamente, o nome e o tipo de dados do índice.

Listagem 4.12: Definição do índice indexProjectOwner

```
{
   "index":{
```

```
        "fields":["docType","Manager"]
    },
    "ddoc":"indexProjectOwnerDoc",
    "name":"indexProjectOwner",
    "type":"json"
}
```

A título de exemplo, em 4.12, é definido um índice com os campos *docType* e *Manager*, o que irá permitir optimizar *queries* ao gestor de cada projecto. Ao invés de ser realizada uma pesquisa global ao *world state* a todos os documentos em busca de campos que igualem o valor do *Manager*, a existência do índice facilita este processo, disponibilizando uma série de referências de documentos organizadas por *Manager*. Cada vez que um novo documento é adicionado, os índices têm que ser actualizados. O comportamento *standard* do CouchDB é re-indexar quando é recebida uma *query*, no entanto o Hyperledger Fabric lida com a indexação de documentos cada vez que um bloco é *commited* para o *ledger*, possibilitando velocidades de pesquisa muito mais elevadas. Este processo tem por nome *index warming*.

As definições dos índices são, posteriormente, adicionadas ao *chaincode* antes do passo de *packaging* [7], devendo ser colocadas na mesma diretoria do *chaincode* em `META-INF/statedb/couchdb/indexes`.

Listagem 4.13: Exemplo de uma query parametrizada

```
async GetAllManagerProjects(ctx, manager) {
        let queryString = {};
        queryString.selector = {};
        queryString.selector.docType = 'project';
        queryString.selector.Manager = manager;
        return await this.GetQueryResultForQueryString(ctx, JSON.stringify(queryString));
    }

async GetQueryResultForQueryString(ctx, queryString) {
        let resultsIterator = await ctx.stub.getQueryResult(queryString);
        let results = await this.GetAllResults(resultsIterator);
        return JSON.stringify(results);
    }
```

Em 4.13 apresenta-se um exemplo de uma *query* parametrizada. Neste exemplo em específico, a *query* está construída para encontrar documentos do tipo *project* em que o atributo Manager tenha o valor passado. A *selector query* construída é passada para a função *GetQueryResultForQueryString*, que por sua vez invoca a função *getQueryResult* pertencente à *shim API* do Fabric que executa a *query* na *state database*. O CouchDB selecciona automaticamente o melhor índice para servir de suporte à *query* pretendida, sendo que neste caso a decisão óbvia passaria pelo recurso ao índice apresentado em 4.12.

4.3 Aplicação Cliente - API

De forma a interagir com o *chaincode* desenvolvido, foi implementada uma aplicação - API - que invoca directamente as operações dos *smart contracts*. Este componente foi desenvolvido em Node.js, e assim sendo, foi possível utilizar o SDK disponibilizado pelo Hyperledger Fabric. O SDK é composto pelos seguintes módulos: *fabric-ca-client*, *fabric-network*, *fabric-common e fabric-protos*. Realça-se a relevância dos dois primeiros; o primeiro por ser o componente que permite fazer *register* e *enroll* de um novo utilizador, gerando assim novas *trusted identities* na blockchain; o segundo por ser a principal API usada pela aplicação para interagir com a rede Fabric, fazer *queries* e submeter transacções.

4.3.1 Configuração do *Connection Profile* e *Gateway*

Para ser possível estabelecer uma ligação entre a aplicação e a camada da rede Fabric, é necessário uma configuração inicial que consiste em construir e disponibilizar um ficheiro JSON contendo informação sobre todos os componentes pertencentes à rede - habitualmente denominado *connection profile* [9].

Assim, este ficheiro descreve detalhadamente endereços e certificados dos diversos *peers*, *orderers* e *CAs* da *blockchain*, como também informação acerca dos *channels* e organizações existentes. Este ficheiro é então utilizado na configuração do *network gateway* que forma o nível de abstracção de comunicação com a rede da *blockchain*.

De forma a modularizar a aplicação desenvolvida, foi implementado um módulo 4.14 especificamente para este fluxo de configuração - considerando que a ferramenta é multi-utilizador este fluxo terá de se repetir aquando de um novo pedido. A partir de um dado utilizador (identificado através de um JWT, como iremos abordar em subsecção posterior), é carregado o *connection profile*, criada - caso não exista - a *wallet* no sistema de ficheiros, verificada a existência desse utilizador e de seguida, caso o fluxo prossiga, é feita a ligação à rede e obtido o objecto "*network*"que permitirá interagir com a *blockchain*.

Listagem 4.14: Módulo de interacção com a *gateway*

```
1  const { Gateway, Wallets } = require('fabric-network');
2
3  let gateways = [];
4
5  module.exports =
6  (async function(user) {
7      if(gateways[user]) {
8          return gateways[user];
9      }
10     const gateway = new Gateway();
11     const ccpPath = path.resolve(__dirname, '.', 'connection.json');
12     const ccp = JSON.parse(fs.readFileSync(ccpPath, 'utf8'));
13     const walletPath = path.join(__dirname+'/wallets', 'org0.example.com');
14     const wallet = await Wallets.newFileSystemWallet(walletPath);
15
```

```
16      const identity = await wallet.get(user);
17      if (! identity) {
18          return;
19      }
20
21      await gateway.connect(ccp, { wallet, identity: user, discovery: { enabled: true,
            ↪ asLocalhost: true } });
22
23      const network = await gateway.getNetwork('mychannel');
24      gateways[user] = network;
25      if (network != {}) {
26          return network;
27      }
28  });
```

4.3.2 Autenticação e *Wallet* de identidades

Como foi detalhado em 4.3.1, para interagir com a rede Fabric - configurando a *gateway* - é necessário, para além de um ficheiro de configuração - *connection profile* - a definição de identidades para cada utilizador. Estas identidades constituem parte do sistema de controlo de acessos do Hyperledger Fabric. No caso da rede criada no desenvolvimento desta dissertação, a única *Certificate Authority* existente é responsável por gerar os certificados necessários, registando os utilizadores e finalizando o fluxo com o *enroll*.

Numa primeira fase, é feito o "registo"do novo utilizador pelo administrador da CA. O nome de utilizador e a palavra-passe escolhida pelo utilizador são associadas a uma nova identidade com o *role* de "cliente"e esta informação é registada da base de dados da CA. De seguida, segue-se o processo de "*enrollment*", em que são gerados os certificados associados à identidade anteriormente definida. Após serem geradas as chaves públicas e privadas, estas são passadas ao utilizador e a identidade é importada para a *wallet*. Um excerto simplificado do fluxo de registo de um novo utilizador é apresentado em 4.15.

Listagem 4.15: Registo de um novo utilizador na CA

```
1  const walletPath = path.join(__dirname+'/../blockchain-network/wallets', affiliation);
2  const wallet = await Wallets.newFileSystemWallet(walletPath);
3
4  const adminIdentity = await wallet.get(adminUserId);
5
6  const provider = wallet.getProviderRegistry().getProvider(adminIdentity.type);
7  const adminUser = await provider.getUserContext(adminIdentity, adminUserId);
8
9  // userId & password definidos pelo utilizador
10 await caClient.register({
11   enrollmentID: userId,
12   enrollmentSecret: password,
13   role: 'client'
14 }, adminUser);
```

```
15 const enrollment = await caClient.enroll({
16   enrollmentID: userId,
17   enrollmentSecret: password
18 });
19 const x509Identity = {
20   credentials: {
21     certificate: enrollment.certificate,
22     privateKey: enrollment.key.toBytes(),
23   },
24   mspId: orgMspId,
25   type: 'X.509'
26 };
27
28 await wallet.put(userId, x509Identity);
```

4.3.3 Interacção com Chaincode

Todas as operações disponibilizadas na API - desde a autenticação, às listagens e gestão das várias componentes da ferramenta - interagem, de alguma forma, com a *blockchain* - diga-se, o *chaincode* desenvolvido. O processo de interacção é semelhante em todas as operações e passa pelos seguintes passos:

1. Inicializar uma nova instância do objecto "*network*", que na prática resulta numa nova configuração da *gateway* para o utilizador que realizou o pedido, ou reutilizar uma *gateway* já configurada para esse utilizador específico;

2. Invocar a função *getContract* do objecto *network*, especificando o chaincode (projectManagementCC) e o nome do *smart contract* dentro do contexto da operação - ver subsecção 4.2.1 - que retorna uma instância do *smart contract*;

3. Consoante o tipo de transacção, invocar a função *evaluateTransaction* ou *submitTransaction*;

   Caso seja um contexto de *query* ao *ledger* - operações de leitura - a função *evaluateTransaction* permite avaliar uma transacção nos nós *endorsers* sem enviar as respostas para o *ordering service*, e assim, sem registar a transacção no *ledger*. Desta forma, o resultado da *query* é enviado muito mais rapidamente.

   Por outro lado, no caso de ser uma operação de escrita, é invocada a função *submitTransaction* que, tal como acontece na função anterior, avalia a transacção nos nós *endorsers* enviando, no entanto, a transacção para o *ordering service* para ser registada no *ledger*.

4. Tratar o resultado da transacção. No contexto da solução desenvolvida, este passo traduz-se na avaliação da *string* JSON recebida e a sua transformação em objecto - *JSON.parse(resultado)*;

Listagem 4.16: Submissão de transacções

```
const network = await require('../blockchain-network/connection')(req.user.id ? req.user.
    ↪ id : 'Admin');

const contract = network.getContract('projectManagementCC');

// Exemplo EvaluateTransaction
let exists = await contract.evaluateTransaction('ProjectExists', params.id).catch(err =>
    ↪ res.status(500).send(errorResponse));
if (JSON.parse(exists)) {
    res.status(409).send(errorResponse);
    return;
}

// Exemplo SubmitTransaction.
let result = await contract.submitTransaction('CreateProject', params.id, params.name,
    ↪ created, params.due_date, params.manager, params.status, params.budget).catch(err
    ↪ => res.status(500).send(errorResponse));
successResponse.data = JSON.parse(result.toString());
res.status(200).send(successResponse);
```

4.3.4 Funcionalidades da API

Tendo em conta as operações de interacção desenvolvidas no *chaincode*, disponibiliza-se o mesmo leque de funcionalidades via a API implementada em Node.js. Encontram-se disponíveis os seguintes recursos:

- /api/projects
- /api/tasks
- /api/timesheets
- /api/users
- /api/risks

Nos três primeiros recursos listados, são suportadas operações do tipo *get*, *set*, *delete* e *update*, que permitem respectivamente listar, criar, apagar e actualizar dados de projectos, tarefas e *timesheets*. Consoante o tipo de utilizador que utilize a ferramenta - diga-se nível de permissão - este poderá, ou não, ter acesso a certas funcionalidades. Apresenta-se na tabela 4.1 uma matriz de *user roles* que ilustra que tipo de permissões cada tipo de utilizador possui.

Ainda no contexto de permissões, mesmo tendo acesso à funcionalidade, tal não significa que o comportamento será semelhante para todos os tipos de utilizadores. Tomando como exemplo a listagem de notificações de risco, dependendo do tipo de utilizador, este terá acesso a diferentes notificações - caso seja *developer* apenas terá acesso às notificações

Tabela 4.1: Matriz de permissões

	Admin	Manager	Developer	Cliente
Listar Projectos	✓	✓	✓	✓
Gerir Projectos	✓	✓		
Listar tarefas	✓	✓	✓	
Gerir Tarefas	✓	✓		
Listar timesheets	✓	✓	✓	
Listar timesheets globais	✓	✓		
Criar timesheets	✓	✓	✓	
Apagar timesheet	✓	✓		
Actualizar timesheet	✓	✓	✓	
Aprovar timesheet	✓	✓		
Gestão de risco	✓	✓	✓	
Gestão de Permissões	✓			

dirigidas ao seu utilizador; caso seja *manager* terá acesso a todas as notificações dirigidas ao seu utilizador, como também notificações específicas de projectos.

De seguida, apresenta-se a listagem de todos os *endpoints* desenvolvidos assim como uma breve descrição. Para evitar repetições, aplica-se um asterisco (*) sempre que se considerou redundante uma nova descrição:

1. /api/users

    a) /register

    Permite a criação de um novo utilizador, com uma combinação *username*/*password* e com as permissões mínimas da aplicação (*Client*).

    b) /login

    Permite fazer *login* na aplicação a partir de um par *username*/*password*, retornando um JWT com duração de vinte e quatro horas, que será a forma de autenticação nos restantes endpoints da API - integração com *Passport.js*.

    c) /changeLevel

    Permite que um utilizador do tipo *Admin* faça alterações aos *roles* e permissões de cada utilizador.

2. /api/projects

    a) /get

    Retorna uma lista de todos os projectos, sendo possível filtrar por *manager* ou *id* do projecto.

    b) /set

    Permite criar um novo projecto a partir de um conjunto de parâmetros.

    c) /delete

    Permite apagar um projecto existente.

d) /update

Permite actualizar dados de um determinado projecto.

e) /transfer

Permite transferir a gestão de um projecto entre utilizadores do tipo *manager*.

3. /api/tasks

a) /get *

b) /set *

c) /delete *

d) /update *

e) /transfer

Permite transferir o *assignee* ou o *reporter* uma tarefa entre utilizadores.

4. /api/timesheets

a) /get *

b) /getProjectHours

Retorna uma estimativa de horas aplicadas num determinado projecto, tendo em conta as tarefas concluídas e as *timesheets* submetidas e já aprovadas.

c) /set *

d) /delete *

e) /update *

f) /approve

Permite a um utilizador do tipo *manager* aprovar uma determinada *timesheet*.

5. /api/risks

a) /get

Permite obter uma listagem de todas as notificações de risco associadas a um dado utilizador e/ou projecto. Tal como foi abordado acima, dependendo do tipo de utilizador - *developer* ou *manager* - serão listadas notificações de âmbitos diferentes.

b) /dismiss

Permite que um utilizador registe a sua notificação como "lida"de forma a não ser listada novamente.

Realça-se, novamente, o facto de toda esta solução depender única e exclusivamente da rede Fabric e de todas as operações descritas na listagem acima interagirem, de alguma forma, com a *blockchain*, representando assim a camada de armazenamento de dados e de lógica de negócio desta solução.

Capítulo 5

Testes e Avaliação

Este capítulo aborda, em detalhe, os testes realizados sobre a ferramenta e retira algumas conclusões dos resultados obtidos. Primeiramente, são descritos os vários testes realizados e o ambiente em que se realizam, de seguida são apresentadas as métricas e os resultados dos diversos testes e, por último, apresenta-se uma breve conclusão sobre os resultados obtidos.

5.1 Ambiente de Desenvolvimento

Todo o desenvolvimento desta ferramenta foi realizado em máquinas portáteis, não tendo sido considerado outro tipo de ambiente, dado a natureza PoC do projecto. O *setup* de desenvolvimento passou por duas máquinas; inicialmente num computador com o sistema operativo Linux e mais recentemente num portátil MacBook Pro. Os testes descritos foram corridos no referido MacBook Pro com as seguintes especificações:

- CPU - Intel Core i7@ 4,1 GHz, 10.ª geração
- RAM - 16GB
- Armazenamento - 500GB SSD

5.2 Planeamento

Durante o processo de teste, foi rapidamente evidente que a principal fraqueza desta ferramenta seria a robustez e capacidade de lidar com uma elevada carga de pedidos - também dependente, obviamente, da especificações da máquina de testes. Foi concluído que, dependendo do tipo de operação - escrita ou leitura - registam-se diferentes limites

de pedidos concorrentes, justificado pelo facto de operações de leitura não submeterem transacções para o *ledger*. O que acontece é que números consideráveis de pedidos concorrentes causam falhas de comunicação com os *peers* e/ou *orderers*. Estas falhas de comunicação são colmatadas pelos vários *retries* que o SDK realiza, mas na maior parte das vezes resulta na perda da transacção.

Assim, decidiu-se que a melhor abordagem de teste para avaliar a ferramenta seria correr uma série de testes de carga sobre a API de forma a estudar o seu comportamento. Os testes dividem-se entre rondas de operações de leitura e escrita, de modo a serem comparadas, e por fim ainda foi aplicada a ferramenta de *benchmark* Hyperledger Caliper [8]. O Caliper é uma ferramenta de medição de performance da *blockchain* usada para medir a performance directamente pela invocação de uma função de um *smart contract*. A performance é medida em transacções-por-segundo - TPS. Desta forma, descarta-se a componente da aplicação e apenas se testa a performance da blockchain.

5.3 Implementação

Desenvolveu-se inicialmente uma colecção postman contendo todos os *endpoints* 5.1 existentes para facilitar o desenvolvimento. Esta colecção foi posteriormente aproveitada para optimizar o fluxo de testes. Assim, recorreu-se à ferramenta CLI Newman [18], que é um *runner* CLI para colecções Postman, na construção do *script* de testes. Para cada um dos tipos de rondas - escrita e leitura - foi seleccionado um *endpoint* para ser alvo dos testes. Para a ronda de leitura foi escolhido o `/projects/get` e para a ronda de escrita optou-se pelo `/projects/set`. Para cada uma das rondas, foi necessário adaptar o documento da colecção - exportado do Postman - de modo a executar pedidos apenas aos *endpoints* necessários.

Por fim, foi desenvolvido o *script* em Node.js apresentado em 5.1, que executa em paralelo um determinado número de instâncias - intervalo de valores decidido tendo em conta resultados de testes iniciais - do Newman sobre a colecção e ambiente definidos e apresenta, no fim de cada ronda de execução, métricas relacionadas com o número de transacções falhadas e com sucesso como também um tempo médio de resposta de cada pedido, em milissegundos. Foram corridas cinco iterações deste *script* para cada valor de pedidos, sendo estes resultados posteriormente analisados.

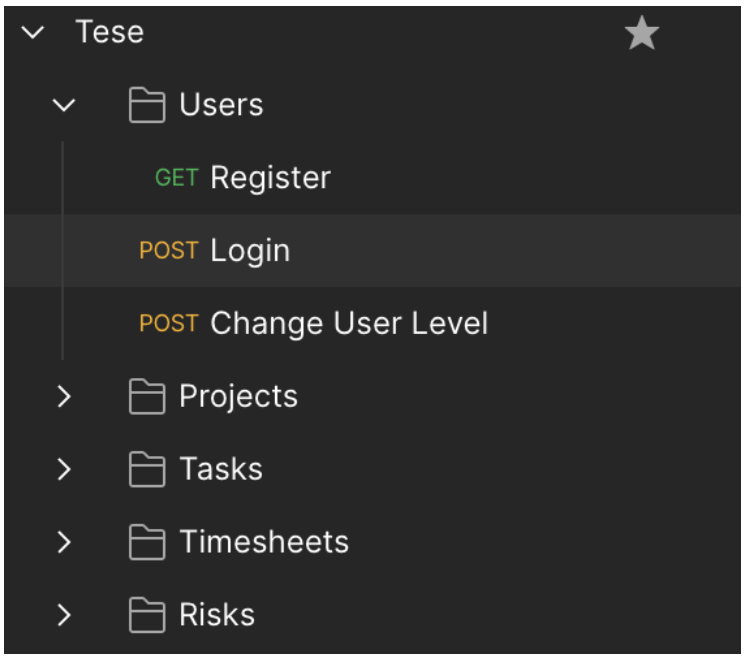

Figura 5.1: Colecção Postman

Listagem 5.1: Script de teste

```
const PARALLEL_RUN_COUNT = 100
const REQUEST_COUNT = 1

const parametersForTestRun = {
    collection: path.join(__dirname, 'postman/Tese.postman_collection.json'),
    environment: path.join(__dirname, 'postman/Tese.postman_environment.json'),
    reporters: 'cli'
};

parallelCollectionRun = function (done) {
    newman.run(parametersForTestRun, done);
};

let commands = []
for (let index = 0; index < PARALLEL_RUN_COUNT; index++) {
    commands.push(parallelCollectionRun);
}

let failed = 0;
let averageResponseTime = 0;
async.parallel(
    commands,
    (err, results) => {
        err && console.error(err);

        results.forEach(function (result) {
```

```
27          failed += result.run.stats.assertions.failed ? result.run.stats.assertions.
                ↪ failed : 0;
28          averageResponseTime += result.run.timings.responseAverage ? result.run.timings.
                ↪ responseAverage : 0;
29      });
30      console.info("Failures:␣"+ failed);
31      let success = PARALLEL_RUN_COUNT * REQUEST_COUNT - failed;
32      console.info("Success:␣"+ success);
33      let percentage = ((PARALLEL_RUN_COUNT * REQUEST_COUNT) * success) / 100;
34      let success_percentage = success > 0 ? (100 * success) / (PARALLEL_RUN_COUNT *
            ↪ REQUEST_COUNT) : 0;
35      console.info("Transaction␣Success␣%:␣"+ success_percentage +"%");
36      let runAverageResponseTime = averageResponseTime / PARALLEL_RUN_COUNT;
37      console.info("Average␣Response␣Time:␣"+runAverageResponseTime);
38  });
```

Tal como foi detalhado anteriormente, uma outra componente de teste foi o recurso à ferramenta Caliper [8]. Esta ferramenta permite invocar directamente uma função de um *smart contract* e estudar apenas a performance da *blockchain*, destacando-se assim da aplicação cliente e podendo agregar resultados para comparação. A ferramenta de *bootstrap* da rede Fabric utilizada - Minifab - vem com o Caliper pré-instalado e de fácil utilização. É necessário realizar alguns passos iniciais que envolvem implementar uma aplicação Node.js que estenda o módulo de *workload* compatível com o Caliper e configurar o ambiente de testes.

Deste modo, foram implementadas duas variações da mesma aplicação que submete uma transacção. Uma primeira invocando uma função de leitura - `ReadProject` - e outra que invoca uma função de escrita - `CreateProject`. De igual forma, foram corridas cinco iterações deste teste de modo a despistar possíveis perturbações de rede. Uma listagem de uma das variações desta aplicação pode ser consultada em 5.2

Listagem 5.2: Aplicação de *Workload* de leitura Caliper

```
1  const { WorkloadModuleBase } = require('@hyperledger/caliper-core');
2
3  class TestWorkload extends WorkloadModuleBase {
4      constructor() {
5          super();
6      }
7      async initializeWorkloadModule(workerIndex, totalWorkers, roundIndex, roundArguments,
           ↪  sutAdapter, sutContext) {
8          await super.initializeWorkloadModule(workerIndex, totalWorkers, roundIndex,
               ↪ roundArguments, sutAdapter, sutContext);
9      }
10     async submitTransaction() {
11         const myArgs = {
12             contractId: this.roundArguments.contractId,
13             contractFunction: 'ProjectAssets:ReadProject',
14             invokerIdentity: this.roundArguments.userID,
```

```
15            contractArguments: ['09c39722-2ebe-48d7-bb98-1efc80d6d144'],
16            readOnly: true
17        };
18        await this.sutAdapter.sendRequests(myArgs);
19    }
20    async cleanupWorkloadModule() {
21    }
22 }
23 function createWorkloadModule() {
24    return new TestWorkload();
25 }
26
27 module.exports.createWorkloadModule = createWorkloadModule;
```

De forma semelhante, é necessário configurar o ambiente de teste do Caliper. Como pode ser visto em 5.3, é configurado, entre outros:

- O nome da *benchmark*;
- Número de *workers* que irão gerar a *workload*;
- Descrição das rondas de teste;
- *txNumber*, o número de transacções que o Caliper submete durante a ronda;
- Descrição do *rate controller*, que controla a velocidade a que as transacções são submetidas - neste caso foi configurado a *fixed-rate* que mantém as TPS (transacções por segundo) estáveis num máximo *threshold*;
- Descrição do módulo de *workload* - ver 5.2;

Os valores escolhidos de *sendRate* tiveram como objectivo sobrecarregar a plataforma, nunca sendo no entanto atingidos. O valor máximo atingido de *sendRate* foi de 314.2 TPS numa das iterações de teste de escrita.

Listagem 5.3: Configuração do ambiente Caliper

```
1  test:
2     name: projectManagementCC-benchmark
3     description: test benchmark
4     workers:
5       type: local
6       number: 2
7     rounds:
8       - label: projectManagementCC test
9         description: projectManagementCC benchmark
10        txNumber: 200
11        rateControl:
12          type: fixed-rate
13          opts:
```

```
14          tps: 1000
15      workload:
16        module: /hyperledger/caliper/workspace/app/callback/app.js
17        arguments:
18          contractId: projectManagementCC
19          randomSeed: 5000000
20          userID: Admin@org0.example.com
```

5.4 Avaliação

5.4.1 Resultados

O fluxo de testes apresentado anteriormente foi corrido e os diversos resultados foram agregados numa folha de cálculo para estudo posterior. As rondas de testes do *script* Newman foram divididas em leitura e escrita, sendo calculados valores médios para cada uma das métricas analisadas, que são apresentadas no gráfico 5.2 e tabela 5.1. No gráfico 5.2 são visíveis duas linhas, uma vermelha representativa da relação entre o número de operações de leitura concorrentes e o tempo de execução médio de cada pedido; e uma azul representando o mesmo para operações de escrita. O eixo horizontal está numerado por número de pedidos concorrentes e o eixo vertical representa, em milissegundos, a duração média de cada pedido efectuado.

De forma semelhante, no caso dos testes via Caliper, foram reunidos os valores de TPS obtidos nas várias rondas de teste realizadas, com diferentes valores de *workload* seleccionados tendo por base os resultados do teste anterior, que estão agrupados no gráfico 5.3. Neste gráfico de barras, apresenta-se a diferença entre os TPS atingidos em cada tipo de operação - leitura e escrita. Representado, novamente, operações de escrita a azul e operações de leitura a vermelho.

Tabela 5.1: Percentagem de sucesso de pedidos por cada tipo de operação.

	% de Sucesso dos Pedidos	
Nº Pedidos	Escrita	Leitura
25	100	100
50	100	100
100	100	100
150	100	100
200	100	100
250	99	100
300	85	100
350	72	100
400	65	100

Os valores apresentados em 5.2 indicam o que se esperava de um sistema dependente da *blockchain*: números elevados de pedidos concorrentes fazem aumentar o tempo de processamento de cada pedido e causam mesmo a falha de um certo número de transacções.

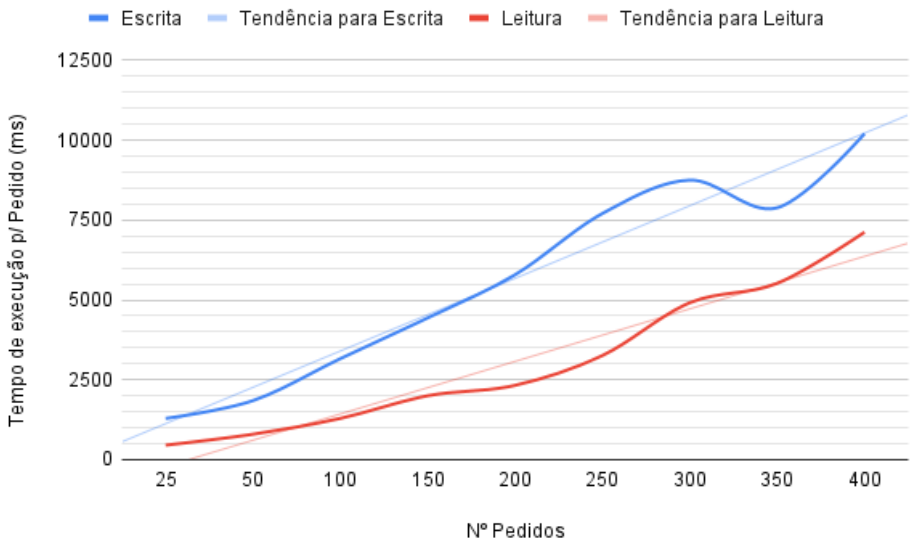

Figura 5.2: Resultados de tempo de execução por pedido

Em relação aos testes a operações de escrita, realça-se o facto de ter sido praticamente impossível a conclusão do último nível de pedidos, devido ao facto da sobrecarga de rede interromper totalmente a comunicação gRPC entre os nós *peers* e *orderers*, obrigando a um "*restart*"da rede em si.

Por outro lado, os testes às operações de leitura revelaram o oposto. Apesar do tempo de resposta obviamente aumentar, não se registam falhas em nenhum dos pedidos realizados. Este facto leva a crer que o principal culpado deste comportamento, para além de problemas de rede causadas pela saturação das comunicações gRPC do SDK - e que leva mesmo à falha de qualquer pedido subsequente, em efeito "bola de neve- é muito provavelmente os mecanismos de *consensus*, devido à elevada taxa de comunicação entre os vários intervenientes do processo.

Noutra perspectiva, nos testes via Caliper, registaram-se valores muito interessantes. O que se pretendia era testar a performance da *blockchain*, ignorando uma das componentes com mais influência nos testes anteriores: a aplicação cliente e a subjacente comunicação entre o SDK e a rede Fabric. Assim, o gráfico 5.3 demonstra que a congestão de rede era causada, na sua grande maioria, nas comunicações gRPC entre o cliente e a rede Hyperledger Fabric. Não se descarta, no entanto o papel dos mecanismos de *consensus* como uma das causas da falha de transacções. Em todas as iterações das rondas de teste de escrita, foram registadas transacções falhadas. Não se estabelece uma relação entre o número de transacções e o número de falhas, visto que nunca ultrapassavam a ordem de uma dúzia. O que se pode retirar dos resultados deste teste é a capacidade do sistema de superar os limites impostos nos testes anteriores, como também estabelecer uma relação entre o tipo de operação e o *throughput* máximo. Em operações de escrita o valor máximo de TPS registado foi na ronda das 750 transacções - 41.6 TPS e em operações de leitura

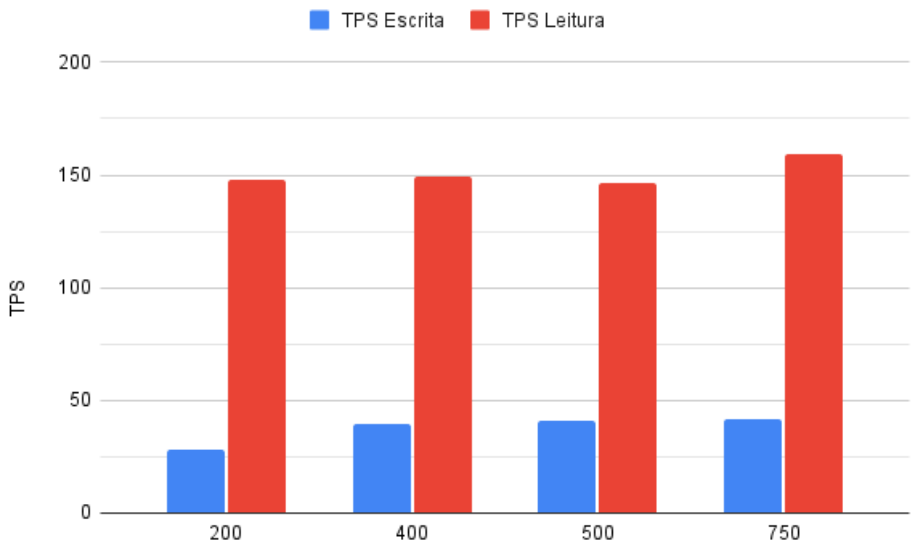

Figura 5.3: Resultados em transacções-por-segundo de cada tipo de operação

foi registado um máximo de 159.1 TPS igualmente na mesma ronda de testes. Em relação aos valores de latência, que o Caliper também reporta, registaram-se, em média, valores na ordem dos 60 milissegundos para operações de leitura - em todas as rondas - e um valor crescente, desde os 3.52 segundos a um máximo de 12.36 segundos na última ronda das operações de escrita. Os resultados demonstram, claramente, um tecto máximo no *throughput* de transacções - quer de leitura, quer de escrita - o que pode ser justificado, não pela plataforma em si, mas sim pela limitação de *hardware* do ambiente de testes.

5.4.2 Conclusão

Tomando em consideração os diversos resultados obtidos, conclui-se que a ferramenta, no seu estado actual de prova-de-conceito, tem condições suficientes para ser integrada e testada no fluxo de trabalho de uma Pequena e Média Empresa (PME). Isto porque numa situação real de uso num destes tipos de organismos, não se espera que existam mais de duzentos e cinquenta gestores de projecto a submeter operações para a rede - limite de congestão da rede. Numa visão ainda mais positiva, tendo em conta os resultados, nenhum dos trabalhadores que realize operações de consulta regularmente na ferramenta terá perturbações de serviço.

Considera-se então, de um ponto de vista puramente técnico, perfeitamente aceitável a performance da ferramenta num ambiente onde o número de utilizadores é mais limitado, tal como é uma PME.

Capítulo 6

Conclusão

Este capítulo final reúne conclusões sobre a prova de conceito desenvolvida e os resultados obtidos, assim como algumas propostas de melhoria e trabalho futuro.

6.1 Objectivos e Resultados

Esta dissertação definiu como objectivo construir uma ferramenta de gestão de projectos que recorresse à *blockchain* para optimizar e automatizar alguns dos processos inerentes à gestão de projecto. O trabalho desenvolvido, que passou pela implementação de cinco *smart contracts* - totalizando sessenta e duas funções - e em 5 rotas - totalizando 21 *endpoints*, culminou numa API com as funcionalidades inicialmente planeadas e com o comportamento desejado. Para além da gestão de cada projecto, de utilizadores, tarefas e *timesheets*, aponta-se o foco para a componente de gestão de risco que permite fazer um "*full scan*"aos projectos e criar notificações com os vários riscos associados. Assim, pode dizer-se que os objectivos foram cumpridos. Os testes de performance realizados apontam ainda que a performance da ferramenta é mais do que adequada para o público para que foi desenvolvida - uma PME.

6.2 Trabalho Futuro

A prova-de-conceito desenvolvida pode ser melhorada em diversos aspectos. De forma a facilitar e melhorar a experiência de utilização desta ferramenta, uma proposta para desenvolvimento futuro será a integração com uma interface gráfica que permita uma experiência costumizada para cada tipo de utilizador. Adicionalmente, a redefinição de alguma da lógica de negócio poderá ser benéfica de um ponto de vista de optimização. A lógica de risco definida nesta prova-de-conceito poderá não ser a mais indicada para uma

empresa ou organismo de outro ramo de actividade, que não o IT. Para diferentes tipos de projectos, exige-se diferentes definições.

6.3 Notas Finais

A aplicação da tecnologia da *blockchain* e *smart contracts* no mundo da gestão de projecto ainda tem muito por onde crescer. Esta prova-de-conceito mostra que tem aplicabilidade e que, com mais investimento, poderá fazer sentido em alguns *use-cases* em que seja necessário bastante automatização de processos e rastreamento de *assets* ou *deliverables* ou em casos de simples adopção de novas ideias e abordagens.

Bibliografia

[1] *Alehub Whitepaper*. URL: https://alehub.io/ALEHUB_WP_eng.pdf.

[2] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco e J. Yellick. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". Em: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. ISBN: 9781450355841. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145/3190508.3190538.

[3] T. Blummer, S. Bohan, M. Bowman, C. Cachin, N. Gaski, N. George, G. Graham, D. Hardman, R. Jagadeesan, T. Keith, R. Khasanshyn, M. Krishna, T. Kuhrt, A. Le Hors, J. Levi, S. Liberman, E. Mendez, D. Middleton, H. Montgomery, D. O'Prey, D. Reed, S. Teis, D. Voell, G. Wallace e B. Yang. *An Introduction to Hyperledger*. 2018. URL: https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf.

[4] V. Buterin. *Ethereum: A next-generation smart contract and decentralized application platform*. 2014. URL: https://github.com/ethereum/wiki/wiki/White-Paper.

[5] *Colony Whitepaper*. URL: https://colony.io/whitepaper.pdf.

[6] C. Dwork e M. Naor. "Pricing via Processing or Combatting Junk Mail". Em: *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '92. Berlin, Heidelberg: Springer-Verlag, 1992, 139–147. ISBN: 3540573402.

[7] *Hyperledger Chaincode Lifecycle*. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.2/chaincode_lifecycle.html.

[8] *Hyperledger Fabric - Caliper*. URL: https://github.com/hyperledger/caliper.

[9] *Hyperledger Fabric - Connection Profile*. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.2/developapps/connectionprofile.html.

[10] *Hyperledger Fabric - Ordering Service*. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.0/orderer/ordering_service.html.

[11] *Hyperledger Fabric - Peers and Orderers*. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.0/peers/peers.html#peers-and-orderers.

[12] *Hyperledger Labs*. URL: https://labs.hyperledger.org/.

[13] P. M. Institute. *A Guide to the Project Management Body of Knowledge (PMBOK Guide) – 2000 Edition*. Project Management Institute, dez. de 2000. ISBN: 9781880410233. URL: https://www.xarg.org/ref/a/1880410230/.

[14] *Introduction — Sawtooth latest documentation*. https://sawtooth.hyperledger.org/docs/core/nightly/0-8/introduction.html. (Accessed on 06/06/2020).

[15] L. Lamport, R. Shostak e M. Pease. "The Byzantine Generals Problem". Em: *ACM Trans. Program. Lang. Syst.* 4.3 (jul. de 1982), 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176.

[16] *Minifabric*. URL: https://github.com/hyperledger-labs/minifabric.

[17] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Dez. de 2008. URL: https://bitcoin.org/bitcoin.pdf.

[18] *Newman CLI*. URL: https://www.npmjs.com/package/newman.

[19] M. Tuler de Oliveira, G. Carrara, N. Fernandes, C. Albuquerque, R. Carrano, D. Medeiros e D. Menezes. "Towards a Performance Evaluation of Private Blockchain Frameworks using a Realistic Workload". Em: fev. de 2019, pp. 180–187. DOI: 10.1109/ICIN.2019.8685888.

[20] I. Pastor, J. Olaso e F. Fuente. "Unveiling the Opportunities of Using Blockchain in Project Management". Em: *1st International Conference on Research and Education in Project Management – REPM 2018* (2018), pp. 22–25. URL: http://dspace.aeipro.com/xmlui/bitstream/handle/123456789/1131/REPM2018_proceedings.pdf?sequence=5&isAllowed=y#page=26.

[21] *Practical Byzantine Fault Tolerance Paper*. 1999. URL: http://pmg.csail.mit.edu/papers/osdi99.pdf.

[22] A. Rao, E. Dimogerontakis, M. Selimi, A. Ali, L. Navarro e A. Sathiaseelan. "Blockchain for Economically Sustainable Wireless Mesh Networks". Em: (nov. de 2018).

[23] K. Rilee. *Understanding Hyperledger Sawtooth — Proof of Elapsed Time*. https://medium.com/kokster/understanding-hyperledger-sawtooth-proof-of-elapsed-time-e0c303577ec1. (Accessed on 06/06/2020).

[24] S. Sabah, N. Mahdi e I. Majeed. "The road to the blockchain technology: Concept and types". Em: 7 (dez. de 2019). DOI: 10.21533/pen.v7i4.935.g450.

[25] S. Seang e D. Torre. *Proof of Work and Proof of Stake Consensus Protocols: A Blockchain Application for Local Complementary Currencies*. GREDEG Working Papers 2019-24. Groupe de REcherche en Droit, Université Côte d'Azur, France, set. de 2019. URL: https://ideas.repec.org/p/gre/wpaper/2019-24.html.

[26] J. Wang, P. Wu, X. Wang e W. Shou. "The outlook of blockchain technology for construction engineering management". Em: *Frontiers of Engineering Management* 4.1, 67 (2017), p. 67. DOI: 10.15302/J-FEM-2017006. URL: http://journal.hep.com.cn/fem/EN/abstract/article_19503.shtml.

[27] J. Yli-Huumo, D. Ko, S. Choi, S. Park e K. Smolander. "Where Is Current Research on Blockchain Technology?—A Systematic Review". Em: *PLOS ONE* 11 (out. de 2016). DOI: 10.1371/journal.pone.0163477.

[28] Z. Zheng, S. Xie, H.-N. Dai, X. Chen e H. Wang. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". Em: jun. de 2017. DOI: 10.1109/BigDataCongress.2017.85.