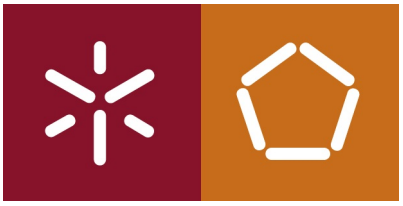

Universidade do Minho
mestrado integrado em engenharia informática

Laboratórios em Engenharia Informática

(2º Semestre / 4º Ano)

Mercado de Dados na Blockchain
BlockDataMarket

Grupo 52:
João Silva (a82005)
Nelson Sousa (a82053)
Pedro Ferreira (a81135)

Resumo

Com a proliferação de novos serviços e sistemas digitais, a quantidade de dados disponíveis tem aberto novas oportunidades para a extração de valor e monetização desses dados. Paralelamente, a tecnologia em torno de *distributed ledgers* potencia uma relação colaborativa e descentralizada com um histórico imutável que oferece garantias de integridade e não-repúdio.

O presente relatório apresenta uma possível solução para uma implementação duma plataforma de mercado de dados em cima duma tecnologia de *Blockchain*. A plataforma visa mapear a dinâmica de mercado num cenário competitivo de procura/oferta na compra e venda de dados.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

No âmbito da unidade curricular de Laboratórios em Engenharia Informática escolhemos o tema *Mercado de dados na blockchain - BlockDataMarket* que se insere na área de Sistemas Distribuídos mas que abrange outras áreas também, tais como, a área da Criptografia e Segurança da Informação e a área dos Sistemas Inteligentes. É, por isso, um projeto multidisciplinar que toca em várias áreas e é realizado por alunos dessas diferentes áreas também.

1.1 Motivações do Projeto

Com a proliferação de novos serviços e sistemas digitais, a quantidade de dados disponíveis tem aberto novas oportunidades para a extração de valor e monetização desses dados. Tipicamente, estes dados alimentam processos de decisão ou previsão cujos possíveis casos de uso cobrem diversas áreas de interesse. No entanto, a utilização destes dados e a monetização dos produtores dos dados dificilmente estão associadas, sendo que é difícil compreender o impacto de cada contribuição individual na sua utilização final.

Paralelamente, a tecnologia em torno de distributed ledgers (e.g., Blockchain) potencia uma relação colaborativa e descentralizada com um histórico imutável que oferece garantias de integridade e não-repúdio. Desta forma, faria sentido a existência de uma plataforma de troca de dados, que incluísse alguma forma de monetização de dados. Uma plataforma que fosse como um mercado, onde as entidades poderiam expor os dados que querem vender, e outras entidades poderiam comprar esses dados, ou na totalidade ou apenas parte deles, devendo por isso existir uma forma de negociação que fosse minimamente programática e contratual entre as entidades.

1.2 Objetivos do Projeto

Com este projeto, o grupo de trabalho tentou avaliar em que medida é possível e viável recorrer a uma tecnologia de *Blockchain* para assegurar o correto funcionamento dum mercado de dados.

Nesse sentido, este projecto tem como objectivo o desenvolvimento de uma plataforma de troca de dados, que inclua monetização, baseada na tecnologia HyperLedger Fabric. A plataforma visa mapear a dinâmica de mercado num cenário competitivo de procura/oferta na compra e venda de dados, devendo permitir aos fornecedores de dados anunciar a sua oferta, aos compradores anunciar o seu interesse e estabelecer de forma programática a relação contratual entre estes.

Com isto, ao longo do relatório será exposto que tipo de decisões foram tomadas para que a lógica do mercado de dados fosse mantida e que tipos de recursos foram necessários para tal.

1.3 Estrutura do Relatório

O presente relatório no capítulo 2 aprofundará o problema em questão assim como trará à tona o domínio do problema e principais requisitos. No capítulo 3 descreverá a arquitetura proposta para albergar os requisitos definidos para o problema em questão.

No capítulo 4 abordará a lógica de mercado presente na rede de *Blockchain* escrita sobre a forma de *Chaincode*. No capítulo 5 descreveremos a aplicação externa criada assim como toda a lógica que permite a aplicação funcionar em conjunto com a rede.

No capítulo 6 será descrita como a negociação dinâmica é feita e porque é útil num sistema como este. No capítulo 7 descrever-se-á como os dados podem passar pela rede de *Blockchain* sem que as entidades que não sejam o vendedor ou o comprador os consigam ver/desencriptar.

No capítulo 8 mostrar-se-á como se avaliou o funcionamento e a viabilidade da solução encontrada. No capítulo 9 mostrar-se-á como se pode fazer o *deployment* dos componentes necessários para testar a plataforma criada. Os últimos dois capítulos são reservados para o trabalho futuro e para as conclusões.

Capítulo 2

Descrição do Problema

No capítulo anterior relatou-se o âmbito do problema em questão e qual os objetivos que se pretendem atingir. Neste capítulo aprofundar-se-á o problema em questão, concebendo alguns modelos que capturem o domínio do problema em questão, bem como alguns requisitos.

2.1 Modelo de Domínio

Para poder caracterizar o ambiente em que o sistema se insere, podemos utilizar um modelo de domínio. A elaboração deste diagrama é importante para efeitos de confirmação dos requisitos e para a descoberta de novos requisitos.

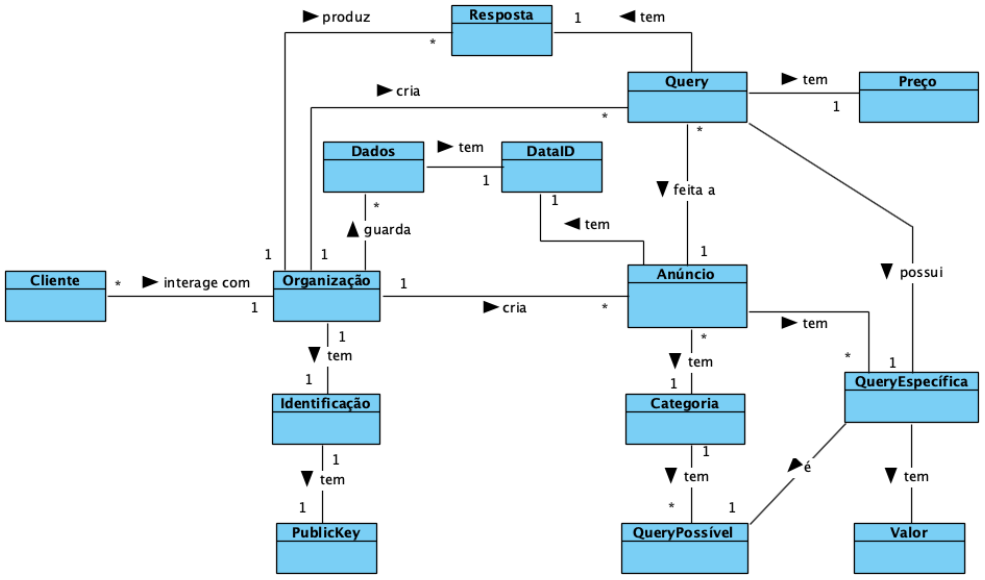

Figura 2.1: Modelo de Domínio

Duma forma muito genérica foi possível através deste modelo de domínio caracterizar o sistema que pretendemos construir capturando assim várias entidades, bem como os seus relacionamentos. Mais tarde adicionaram-se bastantes especificações que não merecem muito destaque num modelo de alto nível como o da figura 2.1.

2.2 Diagrama de Use Cases

Através da análise do domínio do sistema bem como dos objetivos pretendidos começamos por capturar aqueles que parecem ser os requisitos do sistema, ou pelo menos os casos de uso mais frequentes.

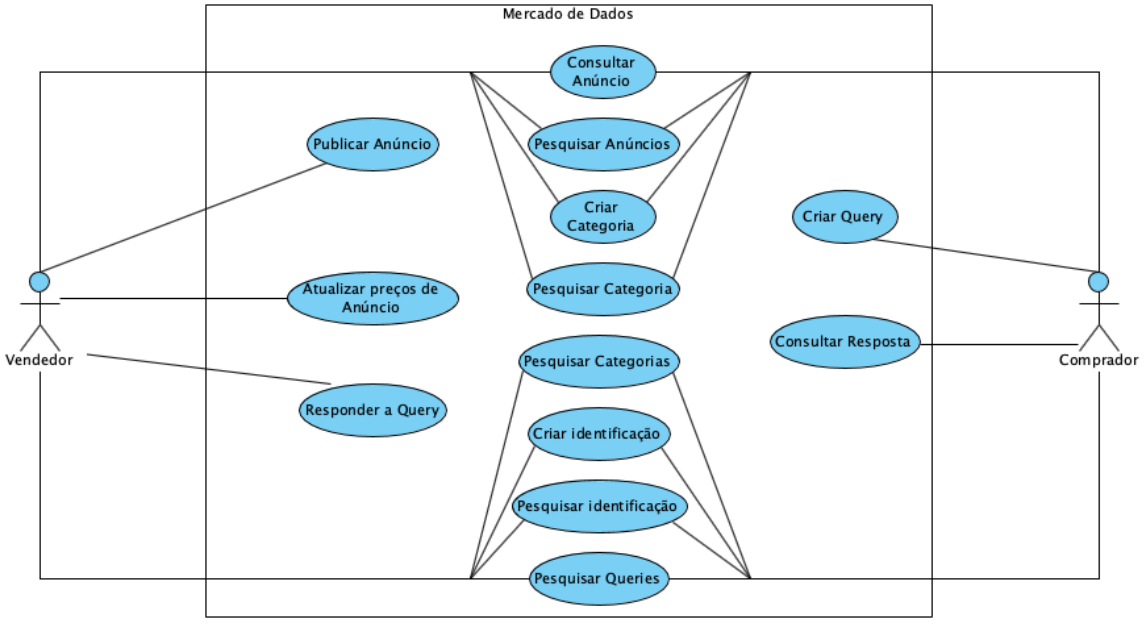

Figura 2.2: Diagrama de Use Cases

Facilmente retiramos dois atores que participarão no processo de compra e venda de dados. O **vendedor** e o **comprador**. Em termos reais uma entidade pode ser vendedor e comprador ao mesmo tempo mas em negociações diferentes, porque quando se trata duma negociação uma entidade ou é o vendedor ou o comprador.

Posta a existência destes dois atores, fica clara a separação dos papéis, bem como as ações que cada ator pode executar no sistema.

No contexto duma negociação, o **vendedor** pode:

- Publicar um anúncio;

- Atualizar preços desse anúncio;
- Responder a Queries feitas a esse Anúncio;

No contexto duma negociação, o **comprador** pode:

- Criar uma Query direcionada a um anúncio;
- Consultar a resposta a essa Query;

Apesar destas ações que são específicas a cada ator, existem ações que podem ser executadas por qualquer ator e que não constam na totalidade no diagrama acima apresentado. Apenas algumas dessas ações aparecem que são as que se encontram no centro do sistema representado no diagrama.

2.3 HyperLedger Fabric

O Hyperledger Fabric é uma plataforma *open-source* de *distributed ledger technology* (DLT), projetada para uso em contextos empresariais, que fornece alguns recursos diferenciadores importantes em relação a outras plataformas populares de *distributed ledgers* ou *blockchain*.

Vale a pena mencionar esta tecnologia assim como alguns termos que dela advêm, pois foi tendo-a em conta que construímos este mercado de dados, e sem ela não poderíamos ter certas garantias que são essenciais para o correto funcionamento do sistema.

Ledger

No Hyperledger Fabric, um *ledger* consiste em duas partes distintas, embora relacionadas - um *World State* e uma *Blockchain*. Cada um deles representa um conjunto de factos sobre um conjunto de objetos de negócio.

World State

O *World State* mantém o valor atual dos atributos de um objeto de negócio como um estado único do *ledger*. Isso é útil porque os programas geralmente exigem o valor atual de um objeto, pelo que seria complicado atravessar todo o *Blockchain* para calcular o valor atual de um objeto. Assim, basta apenas questionar diretamente o *World State*.

Blockchain

Enquanto que o *World State* contém um conjunto de factos relacionados com o estado atual de um conjunto de objetos de negócio, o *Blockchain* é um registo histórico dos factos que dita como os objetos chegaram aos seus estados atuais. O *Blockchain* regista todas as versões anteriores de cada estado do *ledger* e como este foi alterado.

HyperLedger Fabric e Sistemas Distribuídos

Do ponto de vista de Sistemas Distribuídos esta ferramenta parece ser bastante completa, uma vez que para manter as bases de dados das diferentes organizações replicadas, necessita que todo o sistema transacional assente em pressupostos que garantam critérios de coerência adequados.

É uma ferramenta que utiliza protocolos de *consensus* para o ordenar os resultados das múltiplas transações a decorrer, assim como para verificar algumas políticas definidas. O algoritmo de *consensus* a usar pode ter em conta diferentes modelos de faltas, podendo-se escolher desde um algoritmo tolerante a faltas de crash, até um algoritmo tolerante a faltas bizantinas.

Utiliza ainda soluções para problemas bastante conhecidos dentro de sistemas distribuídos, tal como, a eleição de um líder, neste caso para escolher um *peer* na organização que mantenha contacto com o serviço de ordenação.

Utiliza ainda protocolos de disseminação de dados baseados em *Gossip* para passagem de mensagens entre *peers* e o *channel* e para assegurar um bom desempenho e escalabilidade no que toca ao tempo.

Desta forma, e tendo uma ferramenta deste calibre, basta-nos apenas configurar parâmetros consoante o que necessitamos para o mercado de dados, não sendo preciso ter em conta aspetos mais profundos de sistemas distribuídos uma vez que o *Hyperledger Fabric* trata da maior parte deles e os abstrai de quem constrói o *chaincode*.

Capítulo 3

Arquitetura Proposta

Sendo o objetivo deste trabalho explorar a tecnologia blockchain aplicada a um mercado de compra e venda de dados, a peça central da nossa arquitetura não podia deixar de ser uma rede blockchain.

Nesta mesma rede vão estar presentes as informações de cada utilizador, os anúncios de dados para venda, e as *queries* aos dados feitas por quem os comprou.

Decidimos então estipular que cada utilizador, seja este uma empresa ou pessoa comum, possui a sua organização. Esta organização é composta por um **data layer**, uma **API** e um **peer**.

O **data layer** é responsavel por guardar os dados que um determinado utilizador quer vender. Este poderá ser implementado como uma base de dados, ou simplesmente usando o filesystem do utilizador.

A **API** é uma camada de *middleware* responsável por facilitar a interação do utilizador com a plataforma. É a partir desta que o utilizador anuncia e compra dados não sendo assim necessária uma interação direta com a rede que acabaria por ser mais complexa para um utilizador menos experiente. A implementação da **API** respeita um estilo arquitetural *REST* permitindo uma fácil interação e possibilitando a criação futura de uma interface gráfica.

O **peer** é o único ponto de acesso à rede, que permite a cada utilizador questionar e fazer alterações aos dados que lá circulam.

Neste diagrama está presente a proposta de arquitetura descrita neste capitulo.

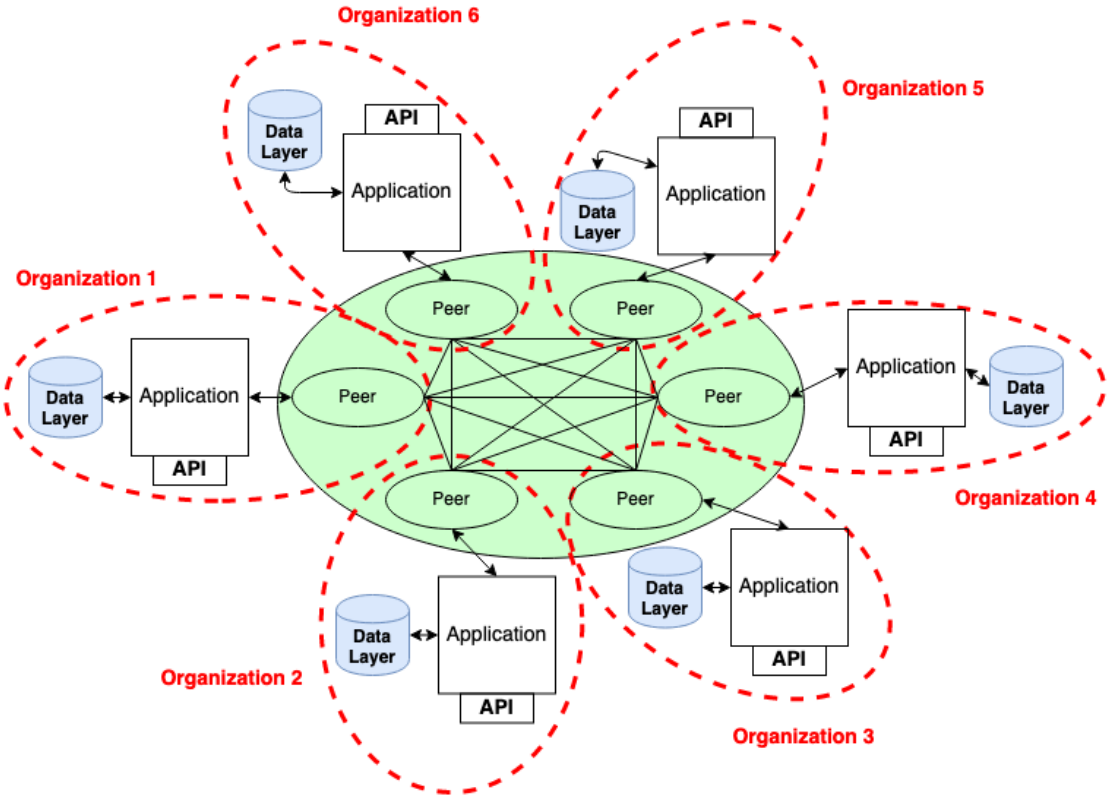

Figura 3.1: Arquitetura Proposta

O objetivo é então, no final, conseguir que uma organização seja constituída por uma camada de dados, uma camada aplicacional e um *peer*. Cada cliente que instalasse um "pacote"com estes três componentes está preparado a interagir com o mercado de dados subsequente.

Capítulo 4

Chaincode

A lógica do mercado de dados é constituída por *Smart Contracts* que juntamente com o *Ledger*, formam o coração do sistema de *Blockchain*. Enquanto que o **Ledger** contém factos sobre o estado atual e o histórico de um conjunto de objetos de negócio, um **Smart Contract** define a lógica de execução que gera novos factos que são adicionados ao *Ledger*. Um **Chaincode** é normalmente usado para agrupar *Smart Contracts* que se relacionam de alguma forma e necessitam de ser *deployed* na rede juntos.

Antes que os nós da rede possam negociar entre si, devem definir um conjunto comum de contratos, cobrindo termos, dados, regras, definições de conceitos e processos comuns. Tomados em conjunto, esses contratos estabelecem o **modelo de negócio** que governa todas as interações entre as partes envolvidas numa transação.

Geralmente, um *Smart Contract* define a lógica da transação que controla o ciclo de vida de um objeto de negócio contido no *World State*. De seguida, esses contratos são empacotados num único *Chaincode* que é *deployed* numa rede *Blockchain*. Fundamentalmente, os *Smart Contracts* podem ser vistos como os controladores das transações, enquanto o *Chaincode* gere como os *Smart Contracts* são empacotados para *deployment*.

Todas as soluções apresentadas a seguir em cada tópico são o culminar de decisões tomadas de forma incremental ao logo de todo o projeto. Apenas nos ficaremos por apresentar as soluções finais ao invés do processo.

Todo o código desenvolvido referente aos conteúdos abordados neste capítulo encontra-se na pasta `BlockDataMarket/chaincode`. De todas as linguagens em que era possível desenvolver *chaincode*, optamos pela linguagem **Go**.

4.1 Estruturas de Dados

As estruturas de dados são os objetos de negócio que são guardados no *World State*, e que representam toda a informação que o mercado de dados precisa de persistir pelas várias Bases de Dados distribuídas para um correto funcionamento do sistema.

Desta forma, e atendendo ao propósito do sistema em questão consideramos as seguintes estruturas:

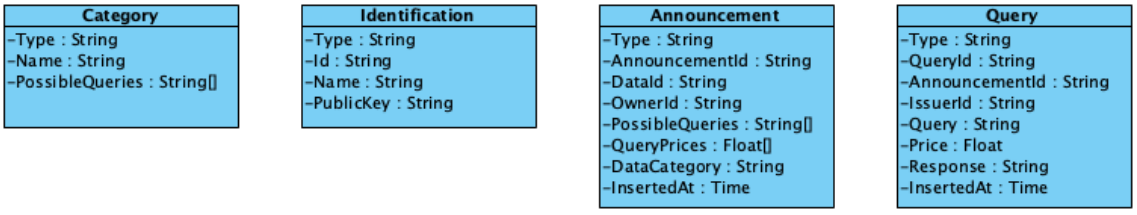

Figura 4.1: Objetos a guardar no *World State*

- **Category:**

  Este tipo de objetos definem as categorias existentes, e as possíveis *queries* que se podem fazer a anúncios que pertençam a uma dada categoria.

- **Identification:**

  Este tipo de objetos definem a identificação duma organização, ou seja, todas as entidades que podem criar anúncios assim como *queries* a anúncios. Serve para diferenciar entidades e também para poder encriptar respostas com utilizando uma *PublicKey* que reside neste objeto.

- **Announcement:**

  Este tipo de objetos representam um anúncio, que contém toda a informação sobre o mesmo, ou seja, as possíveis *queries* que podem ser feitos, os preços de cada *query*, a categoria a que o anúncio pertence, etc. Contém ainda informação para que o anunciante consulte os dados (DataId) que estão fora da rede, possivelente numa Base de Dados externa.

- **Query:**

  Este tipo de objetos represetam as *queries* que se fazem a um determinado anúncio, contendo o proponente, a *query* específica, o valor oferecido pela mesma, e a resposta do anunciante a essa *query*.

4.2 Smart Contracts

Para interagir com os objetos de negócio, ou seja, criá-los, modificá-los e inquiri-los, foi preciso criar um contrato para cada estrutura de dados abordada na secção anterior.

Desta forma, o resultado foi o seguinte.

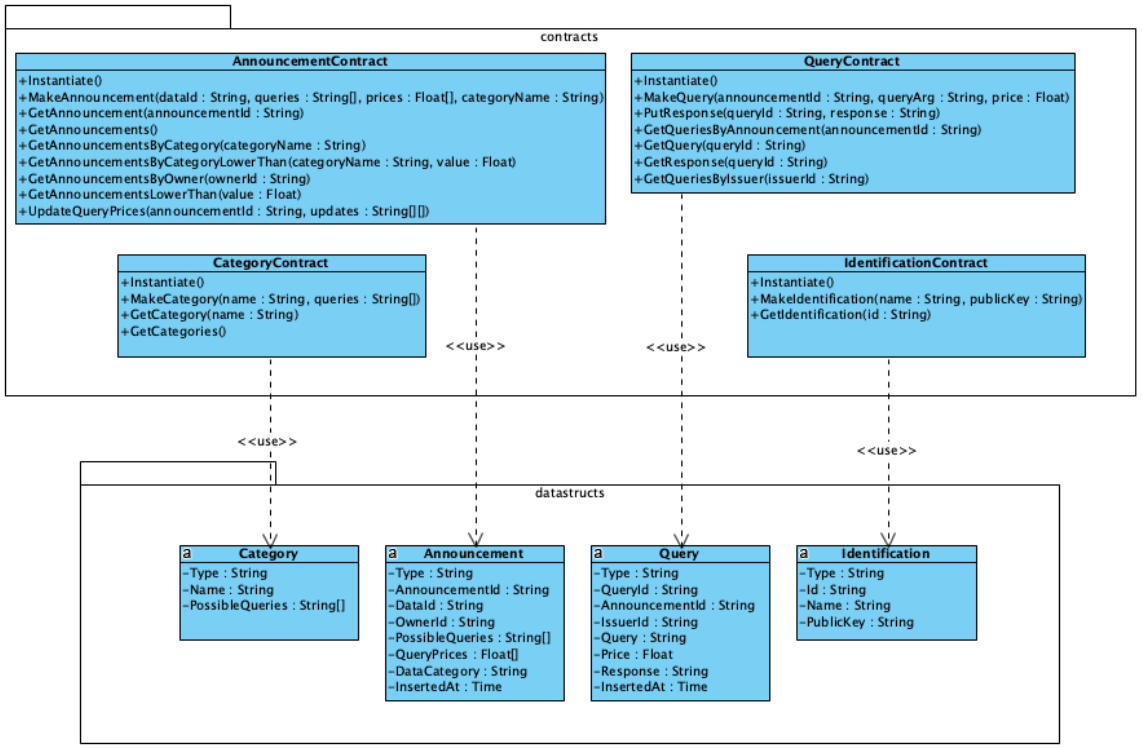

Figura 4.2: Contratos que exploram estruturas de dados

A única forma de interagir com o mercado de dados é através dos métodos que os contratos expõem, logo é necessário que toda a lógica e regras estejam definidas dentro destes métodos.

Para implementar toda a lógica foi necessário ter em consideração alguns fatores que influenciam o funcionamento da lógica do mercado, como por exemplo o uso de *Couch DB* e o controlo da identidade.

4.2.1 Couch DB

Para armazenar os dados do *chaincode* correspondente ao mercado de dados, escolhemos usar uma Base de Dados **Couch DB** por cada organização, que armazena dados num estilo *key-value* e permite modelar os dados do *Ledger* como sendo *JSON*, o que permite que se inquiram os dados usando *queries* mais ricas e se possam usar índices quando necessário.

Tirou-se proveito da composição de chaves para se criarem chaves únicas para cada objeto de negócio. Para cada tipo de objeto as chaves são compostas por:

- **Announcement:**
  - DataCategory
  - OwnerId
  - AnnouncementID
- **Query:**
  - AnnouncementID
  - IssuerID
  - QueryID
- **Identification:**
  - ID
- **Contract:**
  - Name

Desta forma, podem-se inquirir dados através da composição de chaves, ou podem-se criar *queries* ricas sobre parâmetros que não estejam presentes na chave.

4.2.2 Controlo de Acesso ao *Chaincode*

Dado o modelo de dados apresentado, é possível que, por exemplo, um anúncio seja mudado por uma entidade que não o criador desse mesmo anúncio, o que não é de todo desejado, pois o preço duma dada query poderia ser alterada para favorecer uma compra.

Uma forma de contornar este possível problema passa por usar as facilidades do *Chaincode access control*, que permite que o *Chaincode* utilize o certificado de controlo de acesso de quem está a invocar uma dada ação para tomar decisões sobre acesso ao *Chaincode*.

A seguinte imagem representa como uma aplicação externa se deve conectar a um *peer* que faça parte da rede. Para esse efeito são precisos uma **Connection Profile**, uma **Wallet** e uma **Gateway**.

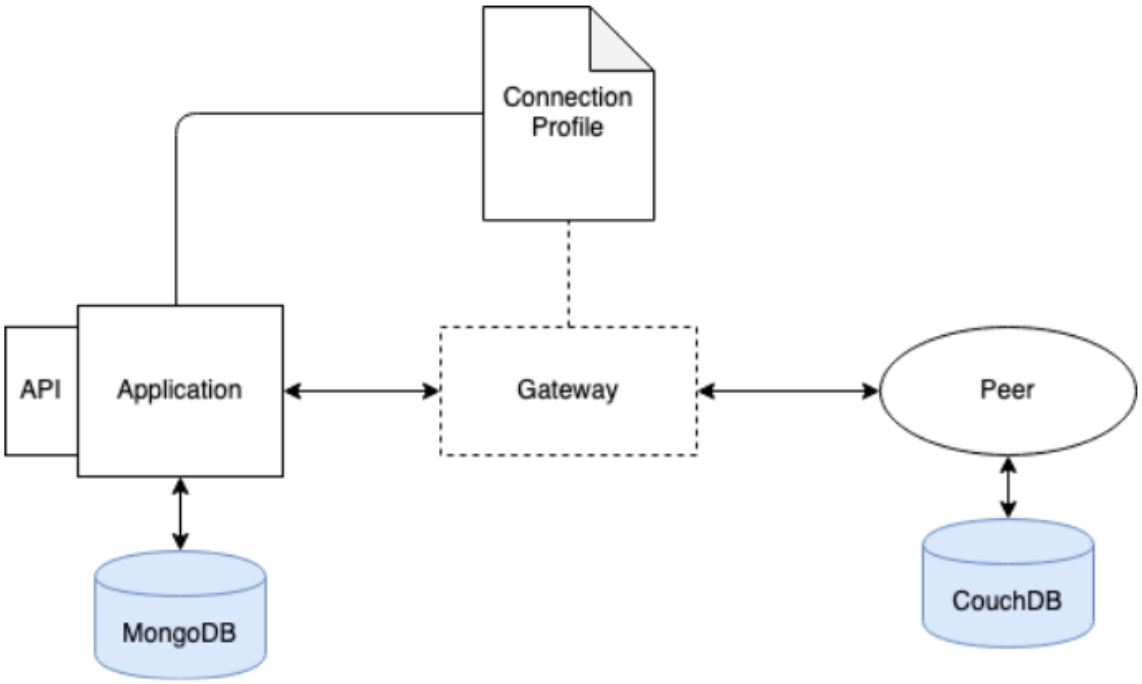

Figura 4.3: Perfil de Conexão

Para o controlo de acesso ao *Chaincode* usam-se principalmente o *Connection Profile* e a *Wallet*, pois estes funcionam como identificadores únicos para cada entidade. É esta a identidade que o *Chaincode* utiliza para identificar as entidades que lhe acedem, assim existe uma maior garantia de que quem está a alterar dados é de facto a entidade responsável por eles.

Ao nível do *Chaincode* esta verificação é feita recorrendo a um **BeforeTransationHandler** que para certos *Contracts* executa antes duma transação começar. É necessário redefinir o contexto de cada *Contract* para tornar possível esta identificação. Este novo contexto (**TransactionContext**) contém uma estrutura *Identification* descrevendo quem é a entidade que está a realizar a transação e a String que define a identidade única falada anteriormente. Caso não exista uma identificação, estes parâmetros são definidos como vazios e mais tarde durante a transação, caso esta necessite duma identidade é abortada, porque não existe nenhuma. Caso a identidade corresponda a uma entidade diferente daquela que criou os dados então a transação também é abortada.

Desta forma, é possível ter garantias que os dados do *World State* não serão mudados por alguém que não tenha permissão para tal.

4.3 Pressupostos

Para que o *Ledger* seja regido tendo em conta a lógica construída acima, é necessário assumir alguns pressupostos para que não se quebre a lógica da construção e manipulação de dados.

- **Existe apenas um *chaincode* instalado na rede:**

  Caso exista mais do que um *chaicode* podem existir outras lógicas que manipulem os dados e acabem por alterar dados de forma não desejada, quebrando regras importantes.

- **Operações de escrita exigem uma *Identification*:**

  Cada entidade que faça uma operação que exija escritas no *Ledger*, tem de estar devidamente identificada tendo que ter um objeto *Identification* no *Ledger*, criado antes da realização da operação de escrita.

Capítulo 5

Aplicação Externa - API

A plataforma Hyperledger disponibiliza um conjunto de SDKs que possibilitam a interação com smart contracts. Neste trabalho, optamos por utilizar o SDK de Node.js, desenvolvendo, sobre este, uma API de alto nível que permite efetuar as operações necessárias à troca de dados, através de pedidos HTTP. Esta visa servir de intermediária entre o utilizador e a plataforma Hyperledger, abstraindo toda a complexidade e detalhes técnicos da rede e das operações que suportam o funcionamento do mercado.

5.1 Fluxo de interação

De forma resumida, a API é composta por 4 rotas, à semelhança dos contratos desenvolvidos: **/announcement**, **/query**, **/identification** e **/category**. De seguida, detalha-se o processo de interação com esta, tenda em conta a ótica de comprador e de vendedor.

A interação com esta aplicação deve iniciar-se com a criação de uma identidade, de forma a que os pedidos seguintes possam ser validados na *blockchain*. Assim, é necessário fazer um `POST /identification`, passando como parâmetros o campo `name`. Este parâmetro pode ser passado através de um JSON ou, por exemplo, utilizando um `Content-Type: x-www-form-urlencoded`. A chave pública do utilizador é gerada pela aplicação. O resultado deste pedido é a identificação criada. De seguida, é necessário criar uma categoria para que esta possa ser atribuída a anúncios futuros. Para este efeito, é necessário fazer um `POST /category`, passando como parâmetros o `name` e as `queries` que essa categoria contempla (sob a forma de *array* de *strings*).

Uma vez realizadas estas duas operações, podemos usufruir em pleno dos contratos implementados.

Do ponto de vista de um comprador, podemos fazer um `GET /category`

para ficarmos a conhecer todas as categorias de dados existentes. Se utilizarmos o parâmetro `categoryName` (*query string*) podemos consultar uma categoria em específico, para, por exemplo, saber quais as queries possíveis. Para publicar um anúncio basta fazer um `POST /announcement`, utilizando `Content-Type: multipart/form-data` com os seguintes campos: `data_file`, `queries` e `category`. O `data_file` deve corresponder ao ficheiro de dados a publicar, e nesta versão da aplicação deve ser um ficheiro XML válido, com o formato descrito no capítulo 8. As `queries` às quais este anúncio irá responder, devem pertencer às queries possíveis da categoria a que o ficheiro de dados pertence. O preço inicial destas é calculado pela aplicação. No entanto, para acompanhar as tendências do mercado, o vendedor pode posteriormente alterar os preços, através de um `POST /announcement/UpdatePrices`, passando como parâmetros o `announcementId` e um mapa de `updates` (pares query-preço a atualizar). Para verificar se um determinado anúncio foi comprado, este deve fazer um `GET /query`, utilizando o parâmetro `announcementId`.

Do pontos de vista de um comprador, podemos consultar todos os anúncios existentes através de um `GET /announcement`, podendo filtrar esta procura por preço (`lt`), categoria (`category`) ou ainda pelo Id de quem publicou o anúncio (`ownerId`), através de uma *query string*. Para efetuar uma *query*, basta fazer um `POST /query`, passando como parâmetros o `announcementId`, a `query` (uma das *queries* a que o anúncio responde) e o `price` proposto. A resposta obtida depende da proporção entre preço pago e o preço anunciado, na medida em que, a título exemplificativo, se pagarmos metade do preço anunciado apenas receberemos metade do conteúdo total.

5.2 Descrição dos *endpoints*

De seguida, são listados todos os *endpoints* da API e os respetivos parâmetros:

- `POST /identification`
  - Parâmetros:
    - `name` (requirido)
      - Nome do utilizador
      - Tipo: String
- `GET /identification`
  - Parâmetros:
    - `identificationId` (requerido)

            - *Id* da identificação
            - Tipo: String

- `POST /category`
    - Parâmetros:
        - `name` (requerido)
            - Nome da categoria
            - Tipo: String
        - `queries` (requerido)
            - *Queries* que a categoria contempla
            - Tipo: Array de Strings

- `GET /category`
    - Parâmetros:
        - `categoryName` (opcional)
            - Nome da categoria a pesquisar
            - Tipo: String

- `POST /announcement`
    - Parâmetros (multipart/form-data):
        - `data_file` (requerido)
            - Ficheiro de dados a publicar
            - Tipo: Ficheiro
        - `queries` (requerido)
            - *Queries* a que o anúncio responde
            - Tipo: Array de Strings
        - `category` (requerido)
            - Nome da categoria a que anúncio pertence
            - Tipo: String

- `GET /announcement`
    - Parâmetros:
        - `ownerId` (opcional)
            - Filtrar por criador do anúncio
            - Tipo: String

        * `lt` (opcional)
            · Procurar anúncios com *queries* abaixo de determinado valor
            · Tipo: Number
        * `category` (opcional)
            · Filtar por categoria
            · Tipo: String

- `POST /announcement/UpdatePrices`
    - Parâmetros:
        * `announcementId` (requerido)
            · *Id* do anúncio a atualizar
            · Tipo: String
        * `updates` (requerido)
            · Pares query-valor que representam os preços a alterar
            · Tipo: JSON Object

- `POST /query`
    - Parâmetros:
        * `announcementId` (requerido)
            · *Id* do anúncio a que se dirige a *query*
            · Tipo: String
        * `query` (requerido)
            · *Query* a que se pretende obter resposta
            · Tipo: String
        * `price` (requerido)
            · Preço a pagar pela *query*
            · Tipo: Number

- `GET /query`
    - Parâmetros (exclusivos):
        * `announcementId` (opcional)
            · *Id* do anúncio a que se dirige a *query*
            · Tipo: String
        * `issuerId` (opcional)
            · *Query* a que se pretende obter resposta

    - Tipo: String
* `queryId` (opcional)
    - Preço a pagar pela *query*
    - Tipo: String

Capítulo 6

Eventos e Negociação Dinâmica

Até ao momento, abordamos a existência do *Chaincode* e de uma aplicação externa que operam em conjunto para oferecerem as funcionalidades do mercado de dados. Ainda assim, o que está a acontecer na realidade é que apenas a aplicação contacta um *peer* e espera a sua resposta, enquanto que o inverso não se verifica.

No entanto, seria útil que a rede de *Blockchain* contactasse as várias aplicações externas para tornar a negociação de dados dinâmica, isto é, não deveria ser a aplicação a pesquisar na rede por *queries* a um anúncio que postou anteriormente, mas sim a rede a notificar a aplicação que uma *query* foi feita a um anúncio detido pela aplicação. Nesta ótica surgem os **eventos**.

O mecanismo de emissão de eventos começa dentro do *Chaincode* onde se identificam os eventos bem como a informação que faz parte dum dado evento. Quando uma transação que contém um evento é *commited*, então o evento é propagado para as aplicações externas. Por si só, um *peer* pertencente à rede não consegue responder a estes eventos, mas mesmo que o conseguisse fazer este não conseguiria aceder aos possíveis dados que um evento poderia requerer. Desta forma, é necessário a existência dum aplicação externa que seja capaz de receber eventos e filtrá-los. Assim, para implementar um mecanismo de eventos e conseguir negociação dinâmica é necessário que exista uma aplicação externa conectada à rede de *Blockchain*.

6.1 Tipos de Eventos

Existem apenas 3 tipos de eventos para assegurar a negociação dinâmica no mercado de dados:

- **Query:AnnouncementId**

  Este tipo de evento é gerado quando é feita uma *query* a um anúncio, sendo direcionado à aplicação externa que criou o anúncio ao qual o *AnnouncementId* corresponde. Contém também todo o conteúdo da *query* criada;

- **Response:QueryId**

  Este tipo de evento é direcionado à aplicação externa que criou a *query* e está à espera duma resposta. Quando uma resposta é colocada nessa *query*, um evento contendo todo o conteúdo da query e com o *QueryId* correspondente é emitido, relatando a existência duma resposta;

- **Update:AnnouncementId**

  Este tipo de evento é emitido pelo *Chaincode*, aquando da alteração dos preços das *queries* de um anúncio. Este é direcionado à aplicação externa que criou o anúncio, devendo esta atualizar os seus preços pelos novos recebidos, de forma a responder coerentemente a futuras *queries*.

6.2 Listeners

Os *listeners* são definidos para cada tipo de evento referido na secção anterior, nas aplicações externas. São estes componentes que recebem os eventos provenientes dos *peers*, que, por sua vez, resultam do término de uma transação.

Os *listeners* são criados quando:

- **Um anúncio é criado:**

  É criado um *listener* para responder a *queries* sobre esse mesmo anúncio, uma vez que só essa aplicação o pode fazer porque detém os dados dele. É também tido em conta que os preços podem ser alterados, pelo que também se tratam esse tipo de eventos neste *listener*.

- **Uma *query* é criada:**

  É criado um *listener* para receber a eventual resposta da aplicação que detém o anúncio.

6.3 Caso de Uso

Para se tornar mais claro como se processam os eventos e como a negociação dinâmica é atingida através destes, iremos, de seguida, exemplificar o caso em que um anúncio é colocado no mercado, uma *query* a esse anúncio é feita e, posteriormente, esta é respondida.

Para uma melhor elucidação, apresentaremos qual seria o processo com e sem eventos.

6.3.1 Sem eventos

Sem a existência de eventos, as ações a realizar são claras e independentes, para cada uma das entidades que participam no processo.

Para o **anunciante**:

1. Publicar o anúncio;
2. De x em x tempo, não sendo este um tempo exato ou sequer conhecido, verificaria se existiam *queries* ao anúncio anteriormente publicado;
3. Existindo *queries* ao anúncio, ser-lhes-ia dada uma resposta.

Para o **comprador**:

1. Colocar a *query* a um anúncio;
2. Ir verificando se a *query* já contém alguma resposta.

6.3.2 Com eventos

Introduzindo os eventos, o processo parece ser um pouco mais complexo do que o processo sem a existência de eventos. No entanto, este adapta-se melhor à realidade de um mercado digital, na medida em que um utilizador obtém a resposta uma determinada *query* de forma imediata, não estando dependente de atuação humana. Este processo é detalhado de seguida:

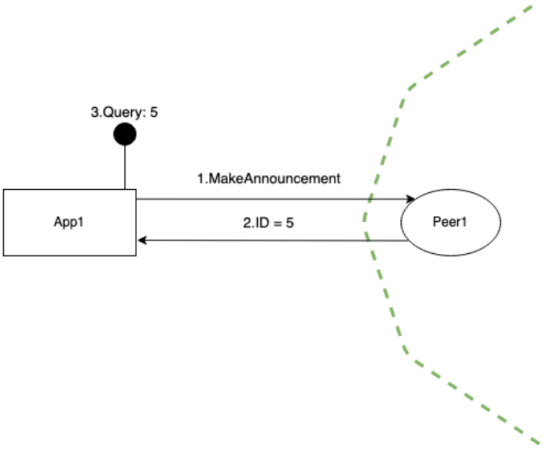

Figura 6.1: Exemplo de Processamento de Eventos

Numa primeira interação, uma aplicação *App1* cria um anúncio. Após a transação concluir, o *peer* retorna o identificador do anúncio, que é usado na aplicação para a criação dum *listener* que fica à espera de eventos com uma tag 'Query:identificador' .

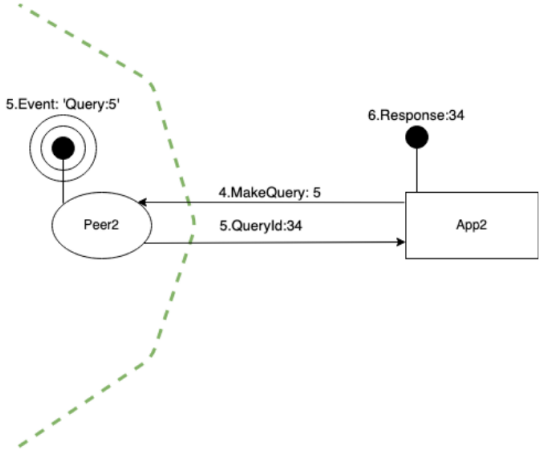

Figura 6.2: Exemplo de Processamento de Eventos

Num momento posterior, outra aplicação *App2* cria uma *query* direcionada ao anúncio anteriormente publicado. Após a transação concluir, um evento é emitido por todos os *peers* às aplicações externas que possivelmente estejam à espera de respostas, e o *peer* em que a *query* foi criada, retorna o identificador da *query*, *QueryID*, à aplicação que o vai usar para criar um *listener* que estará à escuta de respostas à query que deverão vir com uma tag 'Response:QueryId' .

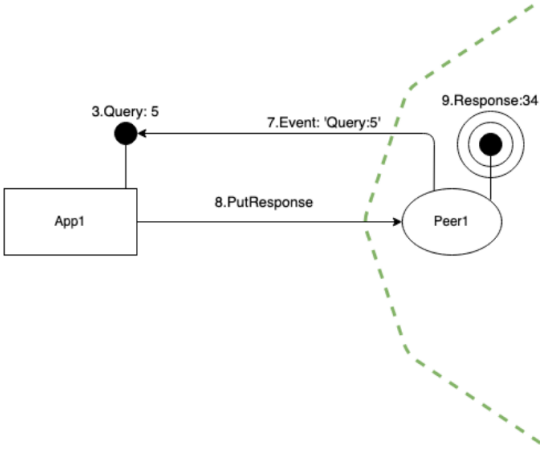

Figura 6.3: Exemplo de Processamento de Eventos

Como referido anteriormente, aquando do término da transação correspondente à criação da *query*, todos os peers disseminam o evento às aplicações externas dispostas a ouvi-lo. No caso da *App1*, esta contém um *listener* disposto a responder à *query* acabada de criar, então calcula a resposta e envia-la, criando assim um evento de resposta 'Response:QueryId' .

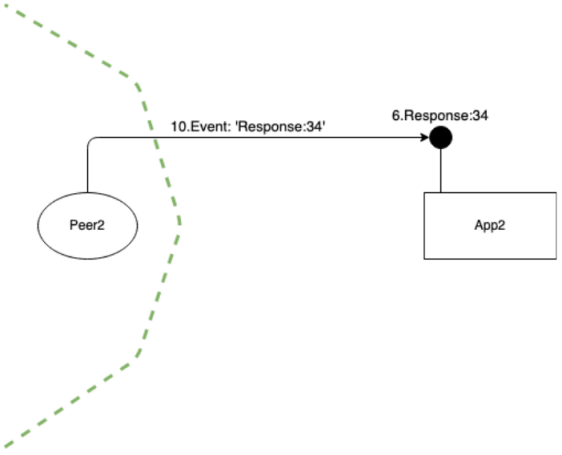

Figura 6.4: Exemplo de Processamento de Eventos

Aquando do término da transação anterior, o *peer2* emite o evento 'Response:QueryId' que é apanhado pelo *listener* existente na *App2* que está à espera duma resposta. Neste momento, o *listener* desencripta a resposta e pode apresentá-la.

Este caso de uso, quando feito sem recorrer a eventos, faz com que a colocação das respostas ou a procura por *queries* a um determinado anúncio seja uma tarefa longa e exaustiva, podendo ser até impossível, uma vez que é manual. Este caso não favorece claramente a negociação dinâmica entre entidades.

Através da utilização de eventos, este caso de uso tipicamente requer algum tempo (alguns segundos) até que que seja concluído, visto que são realizadas algumas transações, o que implica ordenação das mesmas na rede de *Blockchain*. Além disso, efetua também alguns acessos a valores do *World State*, como por exemplo as identificações, para que a resposta possa ser desencriptada. No entanto, este processo é preferível ao processo que não contempla eventos pois este favorece a negociação dinâmica, tornando-a quase instantânea.

6.4 Pressupostos

Inserido este mecanismo de eventos para atingir a negociação dinâmica partimos do pressuposto que uma aplicação externa a partir do momento que é iniciada nunca é desligada pois tem que manter sempre os *listeners* ativos para responder a eventos que possam surgir.

Este é um pressuposto forte mas considera-se que existe algum mecanismo de redundância e de recuperação quando existe alguma falha da aplicação, mantendo os *listeners* sempre ativos.

Capítulo 7

Proteção de Dados

No centro deste projeto estão os **dados**. Sendo o objetivo da plataforma a comercialização dos mesmos, é importante que estes estejam protegidos, sendo de máxima importância que só quem vende e quem comprou determinado conjunto de dados seja capaz de ter acesso a estes.

Como a nossa solução se baseia numa arquitetura em blockchain, onde todos os peers podem questionar a rede, e ter acesso ao conteúdo da mesma, surge a necessidade de impedir que um determinado peer "B"consiga ver os dados que o peer "A"colocou à venda na rede. Num primeiro momento, os dados são guardados numa base de dados local, à qual apenas tem acesso a aplicação responsável pela criação do anúncio. No entanto, quando uma *query* é respondida, o conteúdo da resposta é visível a todos os nós da rede, pelo que necessita de ser protegida. Para esse efeito, decidimos cifrar os dados inseridos na rede de maneira a que apenas quem os comprou consiga efetivamente ler o que lá está.

Para decidir o que cifrar e quando cifrar é importante perceber quando e onde é que vão estar os dados na blockchain. Sabemos então que quando um utilizador "A"publica um anúncio, este não divulga o conjunto de dados que tem para vender, apenas anuncia o seu tipo e as suas características. Quando outro utilizador "B"faz uma questão a esse conjunto por um determinado preço, o utilizador "A"é responsável por responder a essa questão, inserindo os dados correspondestes à query colocada.

Portanto, os dados vão estar presentes na rede apenas no contrato da **query** e vão ser inseridos pela pessoa que os anunciou com o objetivo de que apenas a pessoa que fez a query os consiga ler.

7.1 Métodos de Cifra

Existem dois tipos de cifra, as simétricas e as assimétricas, sendo que as primeiras exigem uma chave única usada para cifrar e decifrar a mensagem, enquanto as outras utilizam uma chave pública para cifrar a mensagem e uma chave privada para decifrar.

À partida o tipo de cifra mais indicado aqui seria o assimétrico, onde o utilizador "A", que quer fornecer a resposta ao utilizador "B", utilizaria a chave pública de "B"para cifrar a resposta, fazendo com que só o utilizador "B"conseguisse, com a sua chave privada, decifrar o criptograma.

Porém, como estamos a falar de, possivelmente, grandes conjuntos de dados a cifrar, este tipo de criptografia não é o mais indicado visto que para mensagens grandes a criptografia assimétrica apresenta um mau desempenho, além de necessitar de chaves de tamanho idêntico às mensagens. Um exemplo prático disto é a implementação do RSA em *nodejs*, que tem como limite de tamanho uma mensagem de 245 bytes, para uma chave de 2048 bits.

7.2 Solução

Para contornar o problema das cifras assimétricas, decidimos optar por uma solução híbrida. Fazendo um acordo de chaves *Diffie-Helman*, uma técnica assimétrica para a partir de uma troca de chaves públicas e das suas respetivas chaves privadas gerar uma chave comum.

Esta chave comum é depois usada para cifrar simetricamente os dados utilizando o algoritmo "AES-256-CBC".

Chaves

Para esta solução, é então necessário que cada utilizador da rede tenha divulgado a sua chave pública na mesma. Para isso, no contrato **identification**, existe um campo *publicKey*, que contém a chave pública do utilizador em questão.

A chave privada de cada utilizador fica guarda localmente na máquina deste, de forma a que ninguém, excepto o mesmo, tenha acesso.

Interação

No seguinte diagrama iremos exemplificar toda a interação dos utilizadores Alice e Bob na publicação e leitura de dados.

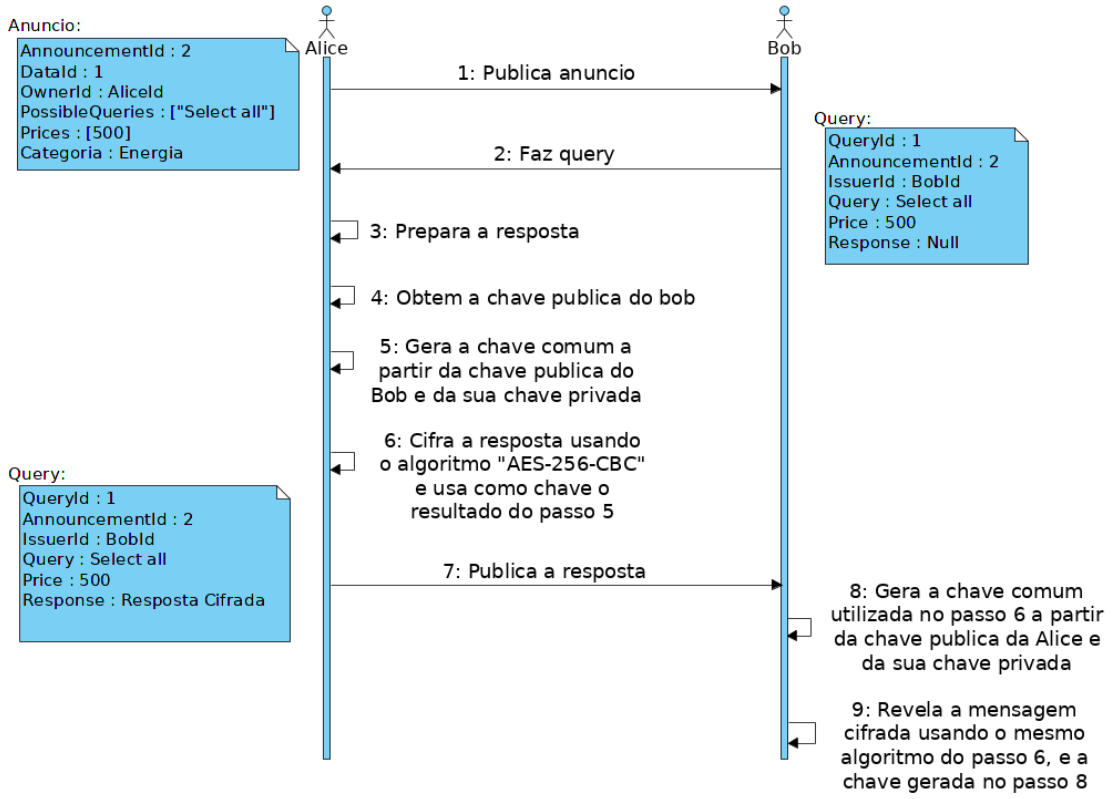

Figura 7.1: Diagrama da interação entre 2 utilizadores envolvidos numa venda

Pressupostos

Para que esta implementação seja considerada segura, há que assumir que cada utilizador não divulga a sua chave privada. Além disso, a comunicação entre os utilizadores e a rede deve ser segura, não sendo possível executar ataques *man-in-middle*, como, por exemplo, um terceiro utilizador "Evil"poder substituir a public key no **Identification** contract do "Bob"e a partir daí conseguir decifrar ele as *queries* que o "Bob"compre.

Capítulo 8

Caso de Estudo

A um mercado de dados está subjacente o conceito de monetização, isto é, de atribuição de valor aos dados disponibilizados pelos vendedores. Num cenário de utilização real, a plataforma deveria ser capaz de possibilitar a compra e venda de diversos tipos de dados, pertencentes a diferentes categorias. Ora, a questão da monetização torna-se então bastante pertinente, relevando-se um desafio de elevada complexidade, na medida em que devemos ser capazes de atribuir valor a dados heterogéneos, que apresentam conteúdos distintos e uma representação díspar. Por questões de simplicidade, optamos por restringir o âmbito da aplicação que interage com o *chaincode*, limitando-se esta a processar ficheiros que pertençam a um *dump* XML da Wikipedia.

De seguida, apreseta-se o formato genérico desses ficheiros:

```
<doc>
    <title>Title</title>
    <url>Url</url>
    <abstract>Abstract</abstract>
    <links>
        <sublink linktype="nav">
            <anchor>Section 1 Title</anchor>
            <link>Section 1 Link</link>
        </sublink>
        (...)
        <sublink linktype="nav">
            <anchor>Section N Title</anchor>
            <link>Section N Link</link>
        </sublink>
    </links>
</doc>
```

Partindo do exemplo acima, podemos então constituir a categoria Wikipédia, que servirá como caso de estudo. Esta é composto por 4 tipos de *queries*: **Title**, **Abstract**, **Url** e **Sections**. Para atribuição de valor a cada uma destas *queries*, utilizamos o tamanho do respetivo campo. Assim, a título exemplificativo, um título com 30 carateres, será avaliado em 30 unidades monetárias.

Recorrendo a este tipo de ficheiros e a esta forma simplificada de monetização, conseguimos simular a compra e venda de dados. A generalização desta operação seria bastante complexa, não só no que diz respeito à valorização dos dados mas também à construção das respostas às *queries*, uma vez que teriamos de ser capazes de processar qualquer tipo de dados, em qualquer formato.

8.1 Simulação de mercado

De forma a simular uma dinâmica de mercado num cenário competitivo de procura/oferta na compra e venda de dados, desenvolvemos dois *scripts*, um que visa imitar o comportamento dos vendedores, e outro para simular o comportamento dos compradores.

Estes encontram-se na pasta `/demo`, e permitem configurar o número de compradores e vendedores, assim como o número de ações que cada um realiza. A cada ação um comprador pode executar uma das seguintes tarefas:

- ***Query* mais barata** - escolhe uma *query* aleatória dentro das possíveis da categoria. Dada essa *query*, consulta todos os anúncios da categoria e escolhe aquele que tem um preço mais barato para a *query* escolhida anteriormente;
- ***Query* aleatória** - escolhe um anúncio aleatório e direciona-lhe uma das *queries* a que responde;
- ***Query* abaixo de determinado preço** - consulta todos os anúncios, de seguida escolhe uma *query* aleatória e calcula o preço médio dessa *query*. Com base nesse preço, questiona a rede sobre anúncios com *queries* abaixo desse preço, escolhendo um aleatóriamente e questionando-o sobre a sua *query* mais barata;
- **Consultar *query* realizada anteriormente** - escolhe o identificador de uma *query* realizada anteriormente e volta a consultar o seu resultado.

As ações da parte do comprador são mais simples. Na sua primeira, ação este publica um anúncio, sendo que nas restantes apenas atualiza o preço das *queries*. Para uma determinada *query* a que o anúncio responda, se entre ações consecutivas (espaçadas temporalmente por 10 segundos) não for registada nenhuma consulta, o

seu preço será reduzido em 10%. Por outro lado, por cada consulta que um *query* registe, o seu preço aumentará em 3%.

Conjugando estes comportamentos, conseguimos imitar o comportamento de um mercado competitivo. De certa forma, as ações dos compradores, ao darem preferência a *queries* mais baratas, faz com que os vendedores tentem adaptar os preços à procura existente. Por outro lado, uma vez que estes também aumentam os preços quando verificam que os seus anúncios foram consultados, as preferências vão mudando ao longo da simulação, permitindo que todos vendam.

Capítulo 9

Deployment

Para ser possível correr este projeto, são necessárias várias peças, de forma a que tudo funcione como esperado. Neste capítulo iremos explicar o que é necessário e como se dá *deploy* de todos os componentes necessários ao projeto.

Os requisitos para possuir o projeto a correr são, uma instalação do *docker*, o *fabric-ledger* na versão 2.2, um container *mongo* a correr, a *test-network* do *fabric* com o chaincode do projeto instanciado, o *nodejs* instalado versão 12+, e o *npm*.

Para cada um destes componentes possuímos scripts que automatizam a instalação e abstraem certas complicações do utilizador. Estas scrips estão localizados na diretoria **bin** do projeto.

9.1 Passos

Os passos e *scripts* a seguir descritos foram construídos num sistema operativo *Ubuntu 18.04 LTS*.

Rede e Chaincode

1. Entrar na pasta do *fabric-setup*, partindo da root do projeto

   ```
   $ cd bin/fabric-setup
   ```

2. Correr o script fabric-setup-part1.sh

   ```
   $ ./fabric-setup-part1.sh
   ```

   Este script altera algumas váriáveis do sistema pelo que no fim da execução, caso esta seja bem sucedida, é necessário sair da sessão do utilizador e voltar a entrar para que as variáveis sejam persistidas no sistema, ou então usar algum comando que faça este trabalho.

Caso algo não corra bem, dever-se-á ao facto de alguma ferramenta não estar instalada pelo que é necessário instalar essa ferramenta e voltar a executar este ficheiro.

3. Correr o script fabric-setup-part2.sh

```
$ ./fabric-setup-part2.sh
```

Este script instala os binários necessários, assim como as imagens de *docker* necessárias para a rede do *fabric*, e cria, ainda, a pasta *fabric-samples*.

4. Entrar na pasta do *fabric-samples*

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

   (a) Por a rede *test-network* a correr com a couchDB e com um canal já criado.

```
$ ./network.sh up createChannel -ca -s couchdb
```

5. Entrar na pasta do deploy-chaincode

```
$ cd ../../bin/deploy-chaincode
```

6. Dar deploy do chaincode na rede

```
$ source ./deploy-chaincode.sh
```

Este script dá *deploy* do chaincode na rede, instalando-o no *channel* existente e nas duas organizações existentes. As duas organizações são também *endorsement peers* pelo que as transações devem ser aprovadas por estas organizações.

Durante a execução deste script existe um momento de interação com o mesmo onde é necessário inserir um *input* que depende do resultado dum comando do script.

API e Base de Dados

1. Entra na pasta mongo-container

```
$ cd ../mongo-container
```

2. Dar deploy de um container mongo executando o script

```
$ ./crate-mongo-container.sh
```

3. Entrar na pasta *application*

```
$ cd ../../application
```

4. Correr o script enrollAdmin, especificando a organização, para que a wallet seja criada.

   ```
   $ node scripts/enrollAdmin.js 1
   ```

5. Correr o servidor da API especificando a porta e a organização.

   ```
   $ ORGNUMBER=1 PORT=3000 npm start
   ```

   Para colocar mais aplicações a correr, basta, neste momento, repetir os dois últimos comandos, alterando os valores relativos à organização e à porta em que a aplicação irá correr.

A maior parte das *scripts* necessárias contêm um README na pasta em que residem, para o caso de ser preciso mais alguma informação.

Neste momento, podem existir algumas aplicações a correrem na mesma máquina. Estas estão a aceder à mesma Base de Dados mas tomam conta apenas dos seus dados e nunca acedem aos dados das outra aplicações. Num contexto real teríamos uma aplicação e Base de Dados por organização em máquinas separadas e este inconveniente não se verificaria.

Simulação de mercado

Para correr os *scripts* de comprador e vendedor é necessário estar na pasta `/demo`. As seguintes instruções assumem que na porta 3000 se encontra uma aplicação conectada ao *peer* da organização 1 e que na porta 3001 se encontra uma aplicação conectada ao *peer* da organização 2.

1. Instalar os packages necessários

   ```
   $ pip install -r requirements.txt
   ```

2. Criar as identificações e a categoria Wikipedia:

   ```
   $ python3 init.py -n Org1
   $ python3 init.py -n Org2 -u "http://localhost:3001"
   ```

3. Correr o *script* de vendedor:

   ```
   $ python3 seller.py -n 10 -c "Wikipedia" -d "./docs"
   -a 10 -u "http://localhost:3000"
   ```

4. Correr o *script* de comprador:

```
$ python3 buyer.py -n 10 -a 10 -u "http://localhost:3001"
```

Estes *scripts* podem ser configurados com outros parâmetros, sendo que aqui se apresentam aqueles que são utilizados por defeito. De seguida, enumeram-se esses parâmetros:

-n - número de compradores/vendedores, conforme o *script*

-a - número de ações a executar

-c - categoria a utilizar

-u - URL da API a utilizar

-d - *path* onde se encontram os anúncios a publicitar

Capítulo 10

Trabalho Futuro

Como trabalho futuro, somos da opinião que este projeto tem ainda algumas áreas de possível exploração. Não fomos capazes de tocar nessas áreas devido ao tempo que possuímos para a concretização do mesmo, e também porque o cerne do trabalho não passava por explorar essas áreas. Deixamos então neste capítulo algumas sugestões de trabalho futuro para enriquecer o trabalho já feito.

Interface Gráfica

Neste momento possuímos uma **API** que permite ao utilizador interagir com a rede através de pedidos REST, e caso queiramos abranger uma maior diversidade de utilizadores, um dos passos a tomar é criar uma interface gráfica amigável, que permita a um utilizador menos qualificado utilizar a nossa plataforma.

Isto poderia ser integrado facilmente e sem interferir no que já está implementado, construindo uma UI em *React* ou *Vue* que consumisse os endpoints já existentes da API, e apresentasse a plataforma para o utilizador de maneira gráfica.

Instalação automática do pacote utilizador

Sempre que um novo utilizador, seja este uma empresa ou uma só pessoa, se quer juntar à plataforma este precisa de criar a sua organização juntá-la à rede, instalar a sua base de dados e dar deploy da sua API.

Uma melhoria interessante, mais na área de DevOps, seria criar um pacote utilizador, suficientemente genérico, transferível, que fosse executável pelo mesmo e fizesse automaticamente a criação da nova organização, a conexão desta à rede existente, desse deploy da API, e instalasse a base de dados.

Assim depois de executar este pacote tudo o que o utilizador teria de fazer era utilizar a plataforma através da API.

Permitir vários utilizadores na mesma organização

Atualmente, a infraestrutura inicialmente proposta está a ser cumprida, ou seja, por cada entidade (organização ou cliente individual), existe um peer ao qual se conecta uma aplicação, que por sua vez explora uma base de dados. No entanto, o acesso à rede (ao *peer*) é feito tendo em conta apenas um perfil de conexão, que é o do administrador do *peer*, portanto mesmo que houvessem múltiplos clientes a usarem a mesma aplicação, estes iriam estar a utilizar o mesmo perfil de conexão à rede, o que não é desejado pois utilizadores dentro da mesma organização conseguiriam consultar dados uns dos outros, a menos que esta lógica de detenção de de dados fosse mantida na aplicação.

Uma outra forma de manter a identidade e dados dos utilizadores preservada mesmo estando a utilizar a mesma aplicação seria cada utilizador ter o seu perfil de conexão à rede guardado na *wallet* e utilizá-lo quando necessário. Desta forma, o *chaincode* não teria apenas as *Identification* das organizações mas sim de todos os clientes de todas as aplicações externas, dando assim mais uma camada de segurança no que toca à privacidade dos dados.

Avaliação de Dados

Na plataforma atual, o preço dos dados é inferido consoante o número de bytes que os dados possuem. Por exemplo, uma possível *query* a dados da categoria *Wikipédia* seria consultar o título do artigo. Se o título possui 20 bytes, então 20 é o preço desta *query*. Fundamentalmente calcula-se o preço ao byte.

De modo a enriquecer o projeto, poderia ser feito um módulo responsável por avaliar um determinado conjunto de dados atribuindo-lhes um preço. Este trabalho seria interessante para a área de Sistemas Inteligentes e Inteligência Artificial, estas áreas poderiam dar resposta com um algoritmo ou método de avaliação de um determinado conjunto de dados.

Inserção de novas categorias na Aplicação externa

Atualmente, o *Chaincode* está preparado para receber qualquer tipo de categoria de dados, no entanto, a aplicação externa está apenas preparada para receber dados *Wikipédia*. A aplicação está preparada para receber mais tipos de dados mas não existem módulos que sejam capazes de os tratar e avaliar.

Futuramente poder-se-iam acrescentar mais categorias que fossem possíveis de tratar e de calcular o seu preço, se bem que esta questão é um bocado *plug-in*, pois à medida que fosse necessário negociar dados de novas categorias poder-se-iam criar novos módulos de tratamento das mesmas.

Recuperação de Listeners em caso de falha

Atualmente, não se tem em conta o que acontece quando uma aplicação é desligada por algum motivo e mais tarde recupera. Desta forma, todos os *Listeners* que existiam na aplicação perdem-se e consequentemente a negociação dinâmica em anúncios que estavam a ser escutados perde-se.

Uma forma simples de resolver este problema seria guardar, por exemplo, em Logs, os *listeners* ativos, e na recuperação reconstruir esses *listeners*. Ainda assim, não seria possível evitar a não resposta às *queries* feitas durante o período de *downtime* do serviço.

Capítulo 11

Conclusões

Olhando em retrospetiva para o trabalho realizado, constatamos que é possível construir um mercado de dados tendo por base uma tecnologia de *Blockchain*, neste caso especificamente o *HyperLedger Fabric*. Bastou construir o *chaincode* que corre na rede de *Blockchain* para que a lógica do mercado ficasse funcionável e usável.

No entanto, para se conseguir ter um ambiente de negociação dinâmica sem que as entidades detentoras dos dados tenham que interferir no processo, necessita-se de algo mais do que apenas o *chaincode*. Referimo-nos aos eventos que consequentemente levam à existência de uma aplicação externa.

A aplicação externa serve para albergar os *listeners* que permitem que haja negociação dinâmica mas também oferece uma abstração no acesso aos métodos da rede, respeitando um estilo arquitetural REST que facilmente é consumido por clientes.

Mais tarde, avaliou-se todo o sistema recorrendo a *scripts* de demonstração comprovando que o sistema está funcional e a lógica de mercado de dados dinâmico está próxima de algo que poderia ser utilizado diariamente.

Ao longo deste projeto, o grupo encontrou algumas dificuldades sendo que estas estiveram mais presentes na fase inicial do projeto, pois foi extremamente complicado criar uma intuição para o problema em causa e para a tecnologia que iríamos usar. O *HyperLedger Fabric* é uma ferramenta (DLT) demasiado complexa com muita documentação, mas que exigiu uma curva de aprendizagem muito longa. A execução dos primeiros tutoriais foi complicada pois não conseguíamos entender com precisão o que estava a acontecer.

Ainda assim, arriscamos a conceber o *Chaincode* numa linguagem desconhecida (Go), o que surpreendentemente não causou muitos problemas, e o mesmo se aplicou à aplicação externa, onde usamos NodeJS.

Dado todo este semestre de trabalho, sentimos que soube a pouco e que o projeto em si tinha muito mais por onde explorar. Ainda assim, conseguimos construir um sistema contido mantendo sempre em mente os objetivos a atingir, não

nos desviando muito daquilo que era pretendido.

Em jeito de conclusão, achamos que fizemos um trabalho completo e que cumpre todos os requisitos e objetivos a que inicialmente nos propusemos na introdução e descrição do problema.