UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

CARLOS MANOEL VASCONCELOS SOUSA

MONGOCHAIN: um framework para implementação de sistemas transacionais

Recife
2020

CARLOS MANOEL VASCONCELOS SOUSA

MONGOCHAIN: um framework para implementação de sistemas transacionais

Este trabalho foi apresentado à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: BANCO DE DADOS

Orientador(a): VALERIA CESARIO TIMES

Recife

2020

Catalogação na fonte
Bibliotecário Cristiano Cosme S. dos Anjos, CRB4-2290

S725m Sousa, Carlos Manoel Vasconcelos
Mongochain: um framework para implementação de sistemas transacionais/ Carlos Manoel Vasconcelos Sousa. – 2020.
87 f.: il., fig.

Orientadora: Valeria Cesário Times.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2020.
Inclui referências.

1. Banco de Dados. 2. ACID. 3. NoSQL. 4. MongoDB. I. Times, Valeria Cesário (orientadora). II. Título.

025.04 CDD (23. ed.) UFPE - CCEN 2021 – 115

Carlos Manoel Vasconcelos Sousa

MongoChain: Um Framework para Implementação de Sistemas Transacionais

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação

Aprovado em: 14/02/2020.

BANCA EXAMINADORA

_____

Profa. Dra. Ana Carolina Brandão Salgado
Centro de Informática / UFPE

_____

Prof. Dr. Ricardo Rodrigues Ciferri
Departamento de Computação / UFSCar

_____

Profa. Dra. Valeria Cesario Times
Centro de Informática / UFPE
(**Orientadora**)

Dedico o resultado deste trabalho aos meus pais, Ivson Carlos e Haydée Vasconcelos, ao meu irmão, Ivson Fortunato, à minha querida esposa, Deisiane Ribeiro, e ao meu amado filho, Davi Ribeiro.

AGRADECIMENTOS

Agradeço a Deus, por me proporcionar concluir esta obra, pois não foi fácil conciliar os deveres de pai, esposo, pesquisador e colaborador.

Aos meus pais e irmão por sempre me apoiarem com vibrações positivas.

A minha querida esposa, pela compreensão nos momentos que eu estive ausente tanto fisicamente quanto perdido em pensamentos, pois a minha mente a quase todo instante estava a pensar em soluções para os problemas levantados nesta dissertação.

Ao meu filho, que muitas vezes chorou por querer brincar comigo, mas depois vinha com um sorriso cativante e renovava as minhas energias.

À Prof.ª. Valéria, por todos os ensinamentos, conselhos e principalmente por ter feito eu acreditar que era possível. Deus te abençoe sempre.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES, pelo incentivo financeiro.

Aos colegas e amigos do Instituto SENAI de Inovação para TICs, pelo apoio nesta reta final de mestrado.

A todos que me deram forças para conseguir obter esta vitória.

RESUMO

Sistemas de Gerenciamento de Banco de Dados (SGBD) relacionais têm como característica fornecer consistência forte aos dados por meio de transações que mantêm as propriedades de Atomicidade, Consistência, Isolamento e Durabilidade (ACID). Porém, não são triviais quando executados em sistemas distribuídos compostos por *clusters*, nos quais pode-se obter escalabilidade horizontal com a adição de mais *nodes*. Logo, os SGBD Não Apenas SQL (NoSQL) de agregados são usados pelos desenvolvedores para fornecer dados com maior disponibilidade. Todavia, a consistência é sacrificada ao adotar os conceitos Basicamente Disponível, Estado Leve e Consistência Eventual (BASE). Ademais, a *blockchain* é uma tecnologia que pode substituir o uso de ACID e BASE ao realizar transações seguras e transparentes em uma rede distribuída e descentralizada. Encontram-se no estado da arte, soluções que integram tecnologias divergentes para contemplar requisitos transacionais em diferentes domínios. Porém, não há um ambiente programável que auxilie os desenvolvedores a implementar e gerenciar sistemas transacionais com ACID, BASE e *blockchain*. Sendo assim, este trabalho apresenta o *MongoChain*, um *framework* proveniente da integração entre o SGBD NoSQL de agregados *MongoDB* com uma rede *blockchain*. Foi realizado um experimento para mostrar a capacidade do *MongoChain* em garantir a consistência dos dados ao executar transações ACID em múltiplos documentos do *MongoDB* e gerenciar uma rede *blockchain*. Além disso, para validar a capacidade de extensão do *MongoChain*, foram construídos dois *frameworks* especialistas que auxiliaram no desenvolvimento das seguintes aplicações: agendamentos em clínicas médicas e *marketplace* de produtos automotivos. Os resultados mostram que o *MongoChain* fornece os mecanismos necessários para provê dados consistentes, escaláveis, disponíveis, seguros e transparentes.

Palavras-chave: ACID; NoSQL; *MongoDB*; *Blockchain*; *Framework*.

ABSTRACT

Relational Database Management Systems (DBMS) have the characteristic of providing strong consistency to data through transactions that maintain the properties of Atomicity, Consistency, Isolation and Durability (ACID). However, they are not trivial when executed in distributed systems composed of clusters, in which it is possible to obtain horizontal scalability with the addition of more nodes. Therefore, aggregate Not Only SQL (NoSQL) DBMS are used by developers to provide data with greater availability. However, consistency is sacrificed when adopting the concepts Basically Available, Soft-State and Eventually Consistent (BASE). In addition, a blockchain is a technology that can replace the use of ACID and BASE when carrying out security and transparent transactions on a distributed and decentralized network. State-of-the-art solutions are found that integrate divergent technologies to address transactional requirements in different domains. However, there is no programmable environment that helps developers to implement and manage transactional systems with ACID, BASE and blockchain. Therefore, this work presents MongoChain, a framework derived from integrating between NoSQL DBMS of MongoDB aggregates with a blockchain network. An experiment was conducted to show MongoChain's ability to ensure data consistency when executing ACID transactions across multiple MongoDB documents and managing a blockchain network. In addition, to validate MongoChain's extensibility, two specialized frameworks were built to assist in the development of the following applications: appointments in medical clinics and the automotive products marketplace. The results show that MongoChain offers the means to provide consistent, scalable, available, secure and transparent data.

Keywords: ACID; NoSQL; MongoDB; Blockchain; Framework.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

2L	Protocolo de Bloqueio de Duas Fases
2PC	Protocolo de Confirmação de Duas Fases
ACID	Atomicidade, Consistência, Isolamento e Durabilidade
API	*Interface* de Programação de Aplicação
BASE	Basicamente Disponível, Estado Leve e Consistência Eventual
BD	Banco de Dados
CAP	Consistência, Disponibilidade e Tolerância à Partição
GUI	*Interface* Gráfica do Utilizador
HTML	Linguagem de Marcação de Hipertexto
HTTP	Protocolo de Transferência de Hipertexto
JSON	Notação de Objeto *Javascript*
LTM	Gerenciador de Transações Locais
MM	Gerenciador de *Middleware*
MVC	Modelo, Visão e Controle
MVCC	Controle de Concorrência Multiversão
NoSQL	Não Apenas SQL
NPVS	Armazenamento de Versão Não Persistente
ODM	Mapeamento de Documentos e Objetos (ODM)
PBFT	Tolerância Prática a Falhas Bizantinas
PoS	Prova de Participação
PoW	Prova de Trabalho
SGBD	Sistemas de Gerenciamento de Banco de Dados
SI	Isolamento Instantâneo
SQL	Linguagem de Consulta Estruturada
TM	Gerenciador de Transação
TO	*Oracle Timestamp*
TSO	*Status Oracle* de Transação
UML	Linguagem de Modelagem Unificada
URL	Localizador Uniforme de Recursos
XML	Linguagem de Marcação Extensível

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta uma contextualização com a importância de desenvolver aplicações que consigam provê dados consistentes, escaláveis, disponíveis, seguros e transparentes. Com isso, será vista a motivação para a construção do *framework* presente neste trabalho. Além disso, foi elaborado o problema de pesquisa e listados os objetivos gerais e específicos para solucioná-lo. Ademais, o uso de um *software* transacional para implementar aplicações que conseguem atender a diferentes domínios será justificado. Ao final, é descrito como esta dissertação está organizada.

1.1 CONTEXTUALIZAÇÃO

Grandes conjuntos de dados (*i.e., Big Data*) são processados por aplicações que executam transações *online* (*e.g.,* operações financeiras, realizações de agendamentos e interações em redes sociais) (ERL, KHATTAK e BUHLER, 2016). Estas aplicações devem estar preparadas para lidar com requisições concorrentes de leitura e escrita sobre um mesmo item de dados. Logo, Sistemas de Gerenciamento de Banco de Dados (SGBD) devem ser usados, pois gerenciam transações ao fazerem uso de algoritmos de controle de concorrência. Elmasri e Navathe (2011) afirmam que os SGBD relacionais conseguem alcançar o isolamento entre transações e garantir a consistência dos dados ao manter as propriedades de Atomicidade, Consistência, Isolamento e Durabilidade (ACID). Porém, por organizarem os dados com relacionamentos entre tabelas, acabam não representando tipos de dados aninhados facilmente (SADALAGE e FOWLER, 2013). Os SGBD Não Apenas SQL (NoSQL) orientados a agregados foram projetados para serem utilizados com grandes volumes de dados por meio do processamento em *clusters* (TIWARI, 2011). Então, aplicações escaláveis são desenvolvidas quando mais *nodes* são adicionados ao *cluster*. Além disso, uma maior disponibilidade é obtida com as seguintes técnicas: i) replicação, quando é feita a cópia e distribuição dos dados na rede. ii) fragmentação, ao manter partes dos dados alocadas em diferentes servidores (SADALAGE E FOWLER, 2013). Segundo Mccreary e Kelly (2014), isso é possível porque os SGBD NoSQL de agregados relaxam o controle da consistência ao adotar os conceitos de Basicamente Disponível, Estado Leve e Consistência Eventual (BASE). Porém, a

facilidade em distribuir os dados pode trazer problemas de inconsistência para as aplicações, pois ao fornecer um conteúdo que se encontra espalhado por diversos *nodes*, os usuários quando realizam uma grande quantidade de requisições ao SGDB podem ter como retorno informações incorretas.

De acordo com Tiwari (2011), em sistemas distribuídos baseados em *clusters*, existe a possibilidade de ocorrer partições, deixando os *nodes* sem comunicação. As falhas podem acontecer devido a problemas originados por componentes físicos, ou seja, relacionados ao *hardware* e à rede. Logo, o teorema (CAP), proposto por Brewer (2012) envolve as características de Consistência, Disponibilidade e Tolerância à Partição, demonstrando que em um ambiente distribuído pode-se obter ao mesmo tempo, duas delas no máximo. Nesse caso, é conveniente ter uma aplicação que seja tolerante à partição restando tratar a questão da disponibilidade com relação à consistência (SADALAGE e FOWLER, 2013). Robinson, Webber e Eifrem (2015) afirmam que dentre os SGBD NoSQL, aqueles que são orientados a *grafos*, também são capazes de fazer uso de transações e manter as propriedades ACID. Então, poderia ser uma solução por unir características presentes nos dois modelos. Porém, Sadalage e Fowler (2013) explicam que esse tipo de SGBD NoSQL não é baseado em agregados e acaba distribuindo os dados de forma semelhante aos SGBD relacionais, portanto não executam bem em *clusters*.

O estado da arte indica que a tecnologia *blockchain* vem sendo estudada e aplicada na indústria de *software* para garantir a segurança e a autenticidade das transações de dados trafegadas na *Internet* (SINGHAL; DHAMEJA; PANDA, 2018). A *blockchain* consiste em uma rede descentralizada e criptografada que certifica e guarda todas as informações transacionais entre as partes envolvidas (*i.e.,* remetente e destinatário). Antonopoulos (2017) afirma que a tecnologia *blockchain* pode ser utilizada para obter maior segurança e transparência ao realizar transações em uma rede distribuída e descentralizada, pois utiliza um protocolo de consenso que elimina a necessidade de adoção de um *node* como controlador central. Tai, Eberhardt e Klems (2017) mostram teoricamente que o processamento e armazenamento em uma rede *blockchain* pode substituir o uso de transações ACID e a aplicação de conceitos BASE. Porém, de acordo com Raval (2016), os dados armazenados na *blockchain* são úteis para operações de leitura, uma vez que não podem ser alterados, sendo necessário o uso de um outro *software* para lidar com a escrita, como um SGBD relacional ou NoSQL. Diante desse cenário, pesquisadores são desafiados a

encontrar soluções que suportem ao mesmo tempo ACID, BASE e *blockchain* para gerenciamento de dados.

1.2 MOTIVAÇÃO

Na literatura de Banco de Dados (BD) encontram-se estudos que visam fazer a integração entre ACID e BASE para manter os dados consistentes, escaláveis e disponíveis ao executar transações. Assim, Coelho *et al.* (2014) propõem um *middleware* capaz de fazer o intermédio entre o gerenciamento de transações ACID e a persistência de dados em um modelo flexível fornecido por algum SGBD NoSQL de agregados. Além disso, o SGBD *MongoDB* gerencia a execução de transações ACID sobre múltiplos documentos que podem ser replicados e fragmentados em um *cluster* (GIAMAS, 2019). No entanto, esses estudos não adotam a tecnologia *blockchain*. De acordo com Mcconaghy *et al.* (2016), o *BigchainDB* é um SGBD distribuído destinado a armazenar dados em uma rede *blockchain* e SGBD NoSQL de agregados. Porém, o *BigchainDB* não executa transações.

Com a tecnologia *blockchain* avançando em ritmo acelerado é necessário investigar se ACID, BASE e *blockchain* podem coexistir como alternativas de processamento de dados e modelo para construção de estruturas que auxiliem os desenvolvedores na implementação de aplicações. Portanto, o trabalho proposto nesta dissertação consiste em especificar um *framework* programável proveniente da integração dos recursos ACID e BASE do *MongoDB* com uma *blockchain* permissionada (*i.e.,* uma rede privada em que o desenvolvedor seleciona quais serão os *nodes* participantes).

1.3 PROBLEMA DE PESQUISA

Ao integrar ACID, BASE e *blockchain* para o gerenciamento de transações e persistência de dados em ambiente distribuído e descentralizado, foi elaborada a seguinte questão de pesquisa:

Como auxiliar os desenvolvedores na construção de sistemas transacionais executados em clusters e que atendam a diferentes domínios com dados consistentes, escaláveis, disponíveis, seguros e transparentes?

1.4 OBJETIVOS

Para responder à pergunta de pesquisa, foram definidos os seguintes objetivos, divididos em gerais e específicos:

1.4.1 Objetivos Gerais

- Auxiliar os desenvolvedores a construírem aplicações para diferentes domínios com dados consistentes, escaláveis, disponíveis, seguros e transparentes.
- Permitir a implementação de sistemas transacionais distribuídos e descentralizados.
- Construir um *framework* programável denominado *MongoChain* por meio da integração entre o SGBD *MongoDB* e uma *blockchain* permissionada.

1.4.2 Objetivos Específicos

- Criar uma rede *blockchain* permissionada, distribuída e descentralizada.
- Realizar a integração do contexto transacional proveniente do SGBD *MongoDB* com uma *blockchain*.
- Definir o gerenciamento de transações envolvendo o SGBD *MongoDB* e uma *blockchain*.
- Desenvolver um mecanismo para acessar os dados originários das transações executadas no SGBD *MongoDB* e na *blockchain*.
- Executar um experimento para avaliar o *MongoChain*, verificando a consistência dos dados e o funcionamento da rede *blockchain* ao realizar transações.
- Validar o *MongoChain* por meio dos seguintes ambientes transacionais distintos: agendamentos de consultas médicas e *marketplace* de produtos automotivos.

1.5 JUSTIFICATIVA

Manter as propriedades ACID ao realizar transações, utilizar um SGBD NoSQL de agregados e fazer uso de uma rede *blockchain* descentralizada são aspectos específicos e possuem propósitos divergentes, tornando-se desafiador realizar a

integração entre eles. Porém, verifica-se por meio do estado da arte que soluções híbridas conseguem contemplar um maior número de requisitos impostos pelas aplicações.

O SGBD *MongoDB* provê a possibilidade de realizar transações ACID em múltiplos documentos localizados em um *cluster*, no qual os dados podem ser replicados e fragmentados, ou seja, é garantida a consistência forte aos dados e, ao mesmo tempo, fornece alta disponibilidade. Além disso, ao hospedar o *cluster* na nuvem, se houver a necessidade de obter maior escalabilidade, poderão ser adicionados mais *nodes*. Diante disso, a proposta desta dissertação é estender as funcionalidades do SGBD *MongoDB*, por meio da adição de uma rede *blockchain* permissionada para garantir também as características de segurança e transparência. Para isto, se faz necessário a implementação do *framework MongoChain*, que vai auxiliar os desenvolvedores na construção de sistemas transacionais para diferentes domínios.

1.6 ORGANIZAÇÃO

Neste capítulo, foi apresentado uma breve introdução mostrando os desafios encontrados para o gerenciamento de transações em ambientes distribuídos e descentralizados. O restante deste documento está dividido em sete capítulos, contados a partir deste e organizados da seguinte maneira:

- Capítulo 2: apresenta o referencial teórico.
- Capítulo 3: mostra os trabalhos relacionados para verificar o estado da arte referente às transações ACID ou *blockchain* atuando em SGBD NoSQL de agregados.
- Capítulo 4: descreve a arquitetura, especificação e especializações do *MongoChain*.
- Capítulo 5: define a prototipação do ambiente de programação.
- Capítulo 6: discorre sobre a avaliação do *MongoChain* por meio da realização de um experimento e desenvolvimento de aplicações.
- Capítulo 7: expõe os resultados obtidos, listando as contribuições deste estudo e as recomendações para trabalhos futuros.

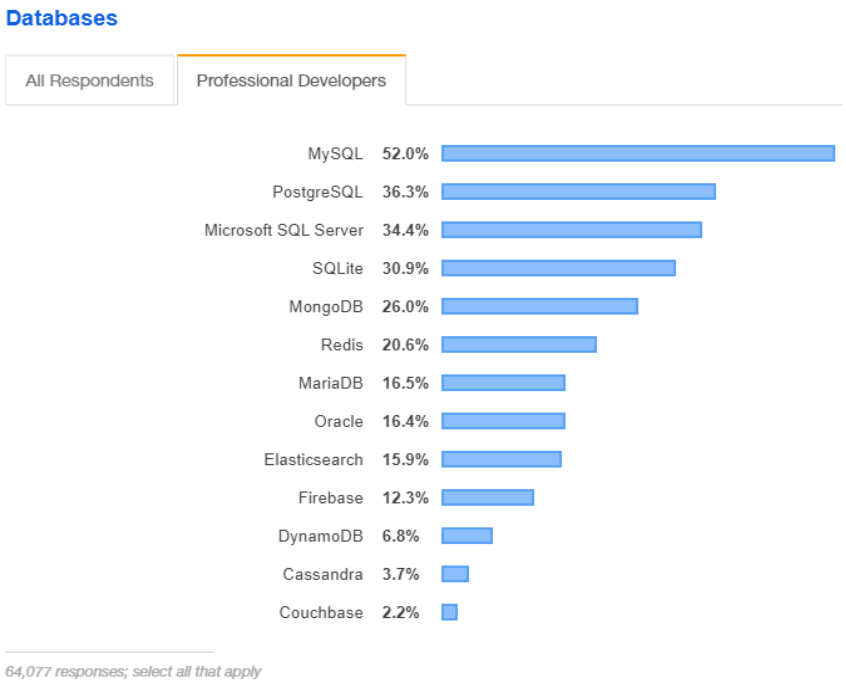
2 REFERENCIAL TEÓRICO

Este capítulo descreve com apoio da literatura, os principais temas que formam a base teórica deste trabalho, a saber: Transações; SGBD NoSQL Baseados em Agregados; Sistemas Distribuídos; Sistemas Distribuídos com Controle Centralizado; Sistemas Descentralizados; *Frameworks*; *Design Patterns*.

2.1 TRANSAÇÕES

Os SGBD relacionais, mesmo com o passar dos anos continuam no mais alto patamar de popularidade entre os desenvolvedores. A Figura 1 mostra o gráfico com um *ranking* dos SGBD mais utilizados proveniente de uma pesquisa realizada em 2019 no site *StackOverflow*, que aborda questões relacionadas à programação.

Figura 1 - Gráfico com o *ranking* dos SGBD mais utilizados em 2019



Fonte: (Developer Survey Results, 2019)

O gráfico demonstra que os três SGBD mais utilizados pelos desenvolvedores são relacionais, mesmo com o aparecimento de novas tecnologias como os SGBD NoSQL. Isso acontece devido a fatores como: capacidade de manter a integridade referencial por meio do modelo relacional; fazer uso de transações para garantir a

consistência dos dados; utilizar a Linguagem de Consulta Estruturada (SQL) como padrão para realizar operações de escrita e leitura em itens de dados. A transação é o objeto de estudo desta dissertação, por isso será apresentado o conceito, funcionamento e importância para os SGBD relacionais, NoSQL e tecnologia *blockchain*. Uma transação é equivalente a um processo, sendo composta de uma ou mais operações de acesso ao BD, podendo ler, escrever, modificar ou remover itens de dados (ELMASRI e NAVATHE, 2011). Ao realizar várias operações, do ponto de vista do usuário, aparenta ter sido feita apenas uma (SILBERSCHATZ, KORTH e SUDARSHAN, 2006). Portanto, ao se cadastrar em um site, o usuário não se preocupa com as diversas operações que acontecem no BD, como a adição de um novo registro contendo seus dados e recuperação destas informações para serem exibidas no *browser*. Elmasri e Navathe (2011) definem a transação como sendo uma unidade atômica, isto é, ela deve ser executada completamente e, no caso da ocorrência de erros deve ser realizado o procedimento de cancelamento. Para isto, é preciso seguir os estados pelos quais uma transação pode passar, que são: *begin_transaction*, *read* ou *write*, *end_transaction*, *commit_transaction* e o *rollback*. Essas operações em SGBD relacionais podem ser manipuladas pelo desenvolvedor por meio de comandos SQL. Para exemplificar, primeiramente é mostrado na Figura 2 a execução de uma consulta SQL, retornando os clientes e suas respectivas contas que contém o saldo atual em um contexto bancário.

Figura 2 - Script SQL e o resultado da execução

```
SELECT c.nome_cliente, c.id_conta,
ct.saldo FROM cliente c
INNER JOIN conta ct
ON c.id_conta = ct.id_conta
ORDER BY c.nome_cliente
```

	NOME_CLIENTE	ID_CONTA	SALDO
1	João	25236	500
2	José	89197	100
3	Maria	67610	200

Fonte: Elaborado pelo Autor (2020)

O cliente João deseja realizar transferências para José e Maria com os respectivos valores 300 e 200. As operações podem ser executadas por meio de uma

transação. A Figura 3, mostra que a transação é configurada com o nome ‘transfere’ por meio do comando *SET TRANSACTION NAME* e são feitas operações de atualizações nos saldos dos clientes envolvidos. O resultado é exibido por meio de uma nova consulta com o mesmo script SQL mostrado na Figura 2.

Figura 3 - Transação de valores entre contas e exibição do resultado

```
SET TRANSACTION NAME 'transfere';
UPDATE conta SET saldo = saldo - 300 WHERE id_conta = 25236;
UPDATE conta SET saldo = saldo + 300 WHERE id_conta = 89197;
UPDATE conta SET saldo = saldo - 200 WHERE id_conta = 25236;
UPDATE conta SET saldo = saldo + 200 WHERE id_conta = 67610;
SAVEPOINT saldo_ant_cli;
```

	NOME_CLIENTE	ID_CONTA	SALDO
1	João	25236	0
2	José	89197	400
3	Maria	67610	400

Fonte: Elaborado pelo Autor (2020)

Foi criado um *SAVEPOINT* ‘saldo_ant_cli’, que consiste na possibilidade de reverter as operações efetuadas em determinado ponto. Sendo assim, o cliente João ao ficar com o saldo zerado decide reverter a transferência realizada para Maria com um novo valor de 150, conforme ilustrado na Figura 4. Portanto, é executada uma operação de *ROLLBACK* sendo atualizados os saldos de João e Maria para 50 e 350 respectivamente. Em seguida, a operação de *COMMIT* é executada para persistir as alterações no BD, não sendo mais permitido executar um *ROLLBACK*.

Figura 4 - Reversão com *Rollback* e exibição do resultado

```
ROLLBACK TO saldo_ant_cli;
UPDATE conta SET saldo = saldo - 150 WHERE id_conta = 25236;
UPDATE conta SET saldo = saldo + 150 WHERE id_conta = 67610;
COMMIT;
```

	NOME_CLIENTE	ID_CONTA	SALDO
1	João	25236	50
2	José	89197	400
3	Maria	67610	350

Fonte: Elaborado pelo Autor (2020)

Em aplicações *Web* os usuários muitas vezes acessam simultaneamente o mesmo item de dados. Silberschatz, Korth e Sudarshan (2006) afirmam que as transações ao serem executadas de forma simultânea podem ocasionar inconsistência nos dados. Além disso, as falhas estão propensas a acontecer devido a utilização de componentes físicos presentes no *hardware* e rede, gerando resultados diferentes do que era esperado. Podem acontecer falhas referentes ao *software*, como a execução de uma transação com erro ao realizar uma operação de divisão por zero (ELMASRI e NAVATHE, 2011). Ademais, o próprio usuário pode ocasionar erros e forçar a interrupção do sistema, sendo preciso mecanismos para prevenir e contornar possíveis intempéries. Logo, para garantir a integridade dos dados, Silberschatz, Korth e Sudarshan (2006) descrevem que é preciso manter as seguintes propriedades:

- Atomicidade
  Ao realizar uma transação, todas as suas operações devem constar no BD ou implicar a existência de nenhuma.
- Consistência
  Se existe apenas uma transação, sem a interferência de outra ocorrendo simultaneamente, então é garantida a manutenção dessa característica.
- Isolamento
  O isolamento acontece quando existem transações sendo executadas simultaneamente e o sistema garante que uma transação não saiba do acontecimento de outra.
- Durabilidade
  Mesmo na ocorrência de falhas depois que uma transação é finalizada, as mudanças realizadas no BD vão ser persistidas.

2.1.1 Concorrência

Na existência de muitos usuários realizando transações simultâneas a um item de dados deve haver um controle de concorrência para evitar dados inconsistentes. Alguns problemas que acontecem quando transações são executadas sem o devido gerenciamento são: atualização perdida, leitura suja, resumo incorreto, leitura não repetitiva e fantasma (ELMASRI e NAVATHE, 2011). A seguir, são apresentadas situações que podem desencadear essas anomalias:

- Atualização perdida

  Quando uma transação T1, ao alterar um item de dados X e uma segunda transação T2, ao realizar um procedimento de leitura nesse mesmo item X, pode enxergar os dados errados se a operação feita por T1 não tiver sido refletida no BD, sendo que T2 ao alterar o mesmo item vai gerar um valor incorreto. Supondo que em um sistema de ingressos para o cinema exista uma sala com cinco cadeiras disponíveis. Assim, João realiza a compra de cinco ingressos e, simultaneamente, Maria vai comprar um. Neste instante, Maria ao invés de visualizar que não há mais cadeiras disponíveis, consegue enxergar as cinco.

- Leitura suja

  Ocorre quando uma transação T1 falha ao atualizar um item de dados X no BD e, nesse mesmo momento, a transação T2 faz a leitura desse item X e visualiza um valor diferente do original. Se em um sistema de leilões João faz a compra da última peça de um lote e ocorre uma falha, Maria ao fazer o procedimento de compra no mesmo instante, o sistema deveria mostrar que a peça está disponível, mas Maria enxerga que a peça foi vendida.

- Resumo incorreto

  A transação T1 é responsável por realizar uma operação que faz o cálculo nos itens de dados X, Y, Z. Ao mesmo tempo, as transações T2 e T3 realizam alterações nesses itens podendo fazer com que a função exercida por T1 calcule os valores antes ou depois de serem atualizados pelas transações T2 e T3. Conjecturando que em um *e-commerce* no final do mês esteja sendo processado o relatório que informa os gastos do período e, no mesmo instante, estão ocorrendo vendas de produtos que irão constar no relatório, a quantidade de alguns produtos vai estar atualizada e de outros não.

- Leitura não repetitiva

  Acontece quando uma transação T1 lê duas vezes um determinado item X e não consegue visualizar o mesmo valor, pois simultaneamente existe uma transação T2 alterando esse mesmo item X. Supondo que em um sistema bancário, o cliente ao consultar seu saldo visualiza o valor de 500 e, neste instante, o banco realiza o débito de 10 referente a encargos financeiros. Em seguida, ao tentar realizar o saque, o cliente recebe a mensagem do sistema informando saldo insuficiente, pois foi lido um valor de 490.

- Fantasma

  Quando uma transação T1 é executada com base em uma condição, terá como retorno um conjunto de dados. Porém, se no mesmo instante uma outra transação T2 realizar uma operação de inserção que satisfaça a mesma condição estabelecida por T1, se ocorrer a repetição da leitura por T1 vai ser mostrado um registro que não existia anteriormente. Ao realizar uma consulta que retorna as compras feitas por um determinado cliente, o funcionário João terá um registro com todas as transações realizadas. Porém, se a funcionária Maria, ao mesmo tempo, realizar uma venda para o mesmo cliente que João está visualizando no sistema, ao fazer a consulta novamente vai ser mostrado para João a nova venda.

As anomalias descritas acima caracterizam a formação de quatro níveis de isolamento definidos por Berenson *et al.* (1995): *READ UNCOMMITTED*, *READ COMMITTED*, *REPEATABLE READ e SERIALIZABLE*. Estes níveis vão servir para o desenvolvedor especificar o quanto a sua aplicação vai estar vulnerável a violações de consistência quando transações concorrentes são executadas. A seguir, o Quadro 1 mostra a relação entre estes níveis e algumas anomalias.

Quadro 1 - Níveis de isolamento e anomalias

Nível	Anomalias
Serializable	Nenhuma
Repeatable Read	Fantasma
Read Commited	Leitura não repetitiva e fantasma
Read Uncommited	Leitura suja, leitura não repetitiva e fantasma

Fonte: Adaptado de (ELMASRI e NAVATHE, 2011)

O nível *Serializable*, não permite a existência desses problemas, pois não vai haver transações concorrentes executando operações sobre um mesmo item de dados. Porém ao fazer essa escolha, vai desencadear uma queda de desempenho, pois as transações serão executadas de forma serial, isto é, uma após a outra. Sendo assim, à medida que as transações vão chegando ao BD é preciso que cada uma aguarde a conclusão das operações que estão sendo realizadas pelas transações

mais adiantadas. O outro extremo é o nível denominado *Read Uncommitted*, o qual é passível de acontecer todos os problemas de violação, mas é capaz de realizar muitas transações concorrentes no mesmo item de dados tornando as aplicações altamente disponíveis. Diante disso, geralmente são utilizados como padrão nas aplicações os níveis *Read Commited* ou *Repeatable Read*, os quais respectivamente tendem a buscar maior disponibilidade e manter maior consistência. Os SGBD que fazem uso de transações precisam realizar o controle de concorrência. Para isto, existem duas abordagens que podem ser adotadas: pessimista e otimista. Na pessimista uma transação bloqueia um item de dados até finalizar as operações necessárias sobre ele (KARYDIS *et al.* 2016). Dentre os protocolos mais utilizados temos o Protocolo de Bloqueio de Duas Fases (2L), o qual Elmasri e Navathe (2011) explicam que a primeira fase consiste em uma transação obter o bloqueio para uma determinada estrutura de um BD, como uma tabela, impedindo que outras possam modificá-la. A segunda fase é a execução das operações para modificar o BD, finalizar a transação e liberar o bloqueio. Assim, esse tipo de técnica vai limitar a capacidade em realizar transações simultâneas, uma vez que outras transações não podem acessar o item de dados bloqueado, ocasionando latência. Um outro fato é a possibilidade de uma transação que obtém um determinado bloqueio precise esperar a liberação de um outro que pertence a uma transação que está sendo executada simultaneamente. Esta situação é conhecida na literatura como *deadlock* e consequentemente vai ocasionar sérios problemas. Portanto, em busca de evitar esses comportamentos, no protocolo otimista as transações são executadas com a expectativa de que não haverá conflitos e, se houver, as medidas necessárias serão tomadas, como um *rollback* (KARYDIS, *et al*. 2016). Alguns dos protocolos mais utilizados com base na abordagem otimista são o Controle de Concorrência Multiversão (MVCC) no qual (Elmasri e Navathe, 2011) explicam que consiste em guardar as cópias das versões antigas de um determinado item de dados para que possam ser acessadas posteriormente e o protocolo de *Timestamp*, que faz a alocação de valor único de *timestamp* para cada transação, garantindo a ordem de execução para operações de leitura e escrita (SILBERSCHATZ, KORTH e SUDARSHAN, 2006).

Atualmente, muitos SGBD utilizam o Isolamento Instantâneo (SI) que substitui a abordagem baseada em bloqueios para lidar com o controle de concorrência. Revilak e O'Neil (2011) explicam que o SI faz uso tanto do MVCC quanto do *Timestamp*. O SI Foi proposto por Berenson *et al.* (1995), com o intuito de fornecer

uma abordagem que estaria entre *Read Commit* e *Repeatable Read*. Sendo assim, segundo Ferro *et al.* (2014) utilizar SI garante que os clientes quando realizam leituras a um item de dados, mesmo sendo executadas transações concorrentes, conseguem visualizar o estado consistente do BD. Isso evita a imposição da exclusão mútua, ou seja, as operações de escrita realizadas em uma transação não fazem o bloqueio das outras (COELHO *et al*. 2014).

2.1.2 Recuperação

As falhas estão propensas a ocorrer nos sistemas devido a fatores como queda na rede, inconsistência no BD e existência de partição. Por isso, é necessário pensar em como preveni-las e, quando vierem a acontecer, saber como tomar as devidas providências. Por exemplo, Coulouris *et al.* (2012) elucidam que os sistemas distribuídos têm a característica de possuírem um controle de falhas, ou seja, mesmo se um componente falhar, os outros ainda conseguem se manter ativos. Em se tratando de falhas que acontecem devido às transações, o sistema deve manter o BD consistente com os dados sendo os mesmos antes do acontecimento da falha (ELMASRI e NAVATHE, 2011). Logo, é importante manter a propriedade de atomicidade, pois garante que a transação aconteça totalmente ou faça a sua reversão para o seu estado inicial. Outra questão é como a propriedade de durabilidade é essencial no tratamento de falhas, uma vez que mesmo na ocorrência de problemas no servidor da aplicação, existe um local a parte que foi estruturado para provê a persistência dos dados.

2.2 SGBD NOSQL BASEADOS EM AGREGADOS

Os SGBD NoSQL surgiram com o intuito de flexibilizar a forma pela qual os dados são armazenados por oferecer maior ênfase na distribuição em relação ao modelo relacional (TIWARI, 2011). Além disso, os SGBD NoSQL de agregados visam provê maior disponibilidade e escalabilidade, uma vez que trabalham muito bem em *clusters* e acabaram se tornando uma valiosa alternativa para lidar com grandes volumes de dados (MCCREARY e KELLY, 2014). Isso acontece devido ao seu esquema ser flexível, ou seja, não há necessidade de preocupação com normalização de dados, integridade referencial ou demais restrições encontradas nos SGBD

relacionais, sendo que esta forma mais dinâmica de armazenar os dados se deve ao modelo orientado a agregados (SADALAGE e FOWLER, 2013).

Os agregados foram definidos por Evans (2004) como sendo uma estrutura de dados com a capacidade necessária para armazenar elementos mais complexos por meio de objetos relacionados. Sadalage e Fowler (2013) mencionam que estes dados no formato de agregados são utilizados pelos tipos de SGBD NoSQL chave-valor, orientado a documentos e família de colunas. Ademais, existem os SGBD NoSQL de grafos que se diferenciam dos agregados, sendo úteis na representação de relacionamentos complexos (VAISH, 2013). A seguir serão apresentados os tipos de SGBD NoSQL baseados em agregados definidos por (SADALAGE e FOWLER, 2013).

2.2.1 Chave-valor

Esse tipo de SGBD NoSQL permite armazenar estruturas de diferentes tipos que podem ser recuperadas por meio da sua chave (MCCREARY e KELLY, 2014). Apesar de possuírem um modelo simples, são bastante eficientes para acessar os dados por meio de uma única chave (TIWARI, 2011). Portanto, os desenvolvedores devem ficar atentos ao fazer uso desse tipo de SGBD, uma vez que a sua flexibilidade pode acabar trazendo problemas de inconsistência para os dados. O Quadro 2 apresenta um exemplo de como diferentes itens de dados podem ser alocados nesse tipo de SGBD.

Quadro 2 - Itens que podem ser adicionados a um SGBD NoSQL chave-valor

Key	Value
e51f9f9747f15cc34fa5dd75b7d0967c	Hash Value
http://localhost:3000	Localhost
img.png	Image File

Fonte: Adaptado de (SADALAGE e FOWLER, 2013)

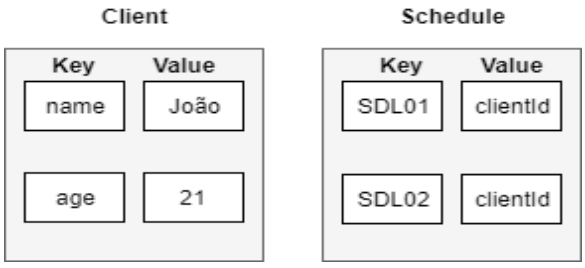
Os campos *Key* e *Value* compõem a estrutura que é definida por Sadalage e Fowler (2013) como tabela *hash*. Logo, cada valor representa um conteúdo único

relacionado a uma chave. Exemplos de SGBD chave-valor são *Riak*, *Redis* e *Amazon DynamoDB*.

2.2.2 Família de Colunas

Possuem um modelo mais complexo do que o chave-valor ao agrupar os atributos em famílias de colunas (POKORNY, 2011). Exemplos de SGBD família de colunas são *Cassandra*, *Hbase* e *Amazon SimpleDB*. Existem basicamente registros sendo armazenados em colunas separadas que possuem pares de chave-valor, como os clientes e seus agendamentos ilustrados na Figura 5. Segundo Sadalage e Fowler (2013), neste tipo de SGBD NoSQL, as linhas e colunas de uma tabela são combinadas para provê alto desempenho na execução de consultas, sendo ideais para o processamento de grandes volumes de dados.

Figura 5 - Registros de agendamentos em SGBD NoSQL família de colunas



Fonte: Adaptado de (SADALAGE e FOWLER, 2013)

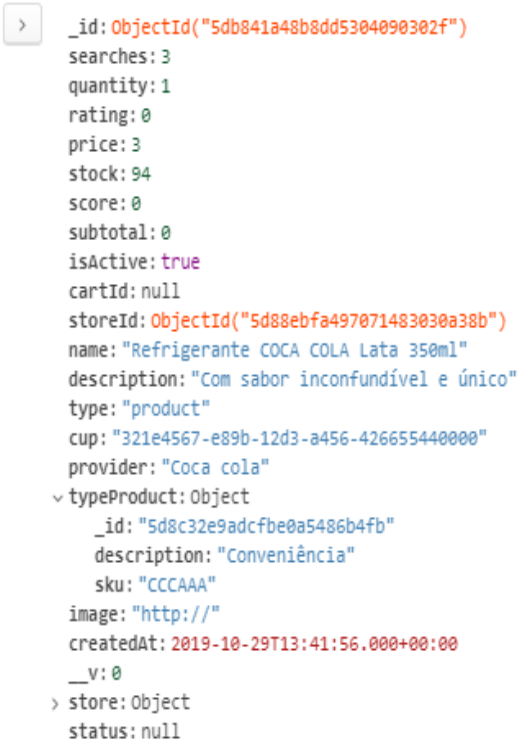
2.2.3 Orientado a Documentos

A estrutura destes SGBD NoSQL se diferencia dos chave-valor por possuir um conjunto de documentos. Cada documento tem campos que são chaves com seus respectivos valores (POKORNY, 2011). Os SGBD orientados a documentos são compostos por coleções que armazenam diferentes tipos de dados no formato da Linguagem de Marcação Extensível (XML) ou a Notação de Objeto *JavaScript* (JSON). Logo, fornecem flexibilidade para aninhar os registros e poder de consulta aos dados (MCCREARY e KELLY, 2014). Além do *MongoDB*, são exemplos de SGBD orientados a documentos o *RavenDB* e *CouchDB*. São bastante utilizados em

conjunto com tecnologias atuais como a *Interface* de Programação de Aplicação (API), que é um mecanismo para realizar fluxo de dados entre cliente e servidor.

A Figura 6 ilustra dados aninhados para compor as características de um produto relacionado a um sistema de *marketplace*. Cada documento pertence a uma coleção e contém um conjunto de campos que não precisa ser definido previamente como acontece com o modelo rígido de tabelas presente nos SGBD relacionais. Logo, cada produto é composto por um identificador único e outros campos que o caracterizam.

Figura 6 - Documento com dados de produtos para um sistema de *marketplace*



Fonte: Elaborado pelo Autor (2020)

2.3 SISTEMAS DISTRIBUÍDOS

De acordo com Coulouris *et al*. (2012) um sistema é distribuído quando os seus componentes estão localizados em computadores que fazem parte de uma rede e a comunicação entre eles é feita unicamente por meio da troca de mensagens. A *Internet* torna viável a criação de sistemas distribuídos, uma vez que as máquinas podem estar presentes em diferentes localizações e conseguem compartilhar seus

componentes. Portanto, observa-se que para desenvolver um sistema distribuído é preciso que os projetistas façam a definição de quais componentes vão compor a aplicação e os papéis que cada um vai desempenhar. Além disso, se faz necessário analisar como estes componentes irão interagir e quais as precauções e correções a serem tomadas nas ocorrências de falhas. Diante disso, Coulouris *et al*. (2012) descrevem as seguintes características presentes em sistemas distribuídos: heterogeneidade, abertura, segurança, escalabilidade, controle de falhas, concorrência, transparência e qualidade de serviço. As características de concorrência e controle de falhas (*i.e.,* recuperação) foram explanadas nas seções 2.1.1 e 2.1.2 respectivamente, as demais serão elucidadas a seguir.

2.3.1 Heterogeneidade

Um sistema é considerado heterogêneo quando não se sabe a procedência dos componentes que o compõem. Os *clusters* por exemplo, não entram nessa classificação, uma vez que tendem a serem máquinas com configurações semelhantes (TANENBAUM e VAN STEEN, 2007). Logo, o fato de ser heterogêneo pode causar problemas como falta de compatibilidade entre os componentes e a não definição de padrões de acesso. Um *middleware* pode atuar como uma camada que trabalha para mascarar heterogeneidade relacionada ao *software*, *hardware* e rede.

2.3.2 Abertura

É quando o sistema tem a capacidade de ser estendido ao alocar novos módulos e realizar a comunicação entre os componentes de forma harmônica, sendo que quando houver a implantação, exista a tendência de ocorrer sem problemas (COULOURIS *et al.* 2012). Portanto, ao desenvolver um sistema distribuído é essencial elaborar documentações, especificações, ou seja, artefatos que sirvam de direcionamento para aqueles que vão fazer modificações no *software*.

2.3.3 Segurança

Com a presença de sistemas distribuídos na *Internet*, problemas de segurança podem aparecer de qualquer parte do mundo ocorrendo uma disseminação sem precedentes. Então, a segurança é um fator de extrema importância para proteger não

apenas o bom funcionamento do sistema, mas principalmente os usuários que irão usá-lo. Sendo assim, Coulouris *et al.* (2012) definem três componentes para garantir a segurança dos recursos relacionados à informação: confidencialidade, integridade e disponibilidade. A confidencialidade está relacionada a condição de que somente o destinatário de uma determinada mensagem tem a autorização de recebê-la (SINGHAL; DHAMEJA; PANDA, 2018). Isto pode ser obtido com a tecnologia *blockchain* ao utilizar criptografia por meio de chaves públicas e privadas. Um sistema que possui integridade é capaz de detectar e recuperar ações realizadas sem a devida autorização em ativos do sistema (*e.g.,* dados) (LAURENCE, 2019). Além disso, se ocorrer violação por alteração não permitida é necessário detectar a ocorrência e identificar o autor para tornar o sistema mais íntegro (SINGHAL; DHAMEJA; PANDA, 2018). A disponibilidade, neste caso, está relacionada ao *software* conseguir lidar com a interferência nos recursos disponíveis (COULOURIS *et al.* 2012). Logo, quando há muitos usuários, o sistema fica propenso a ocorrência de fraudes, sendo necessário provê mecanismos para evitar essa problemática, como validações de dados por meio de uma rede descentralizada. É importante também que os sistemas distribuídos ofereçam disponibilidade no sentido especificado por Tiwari (2011) no qual as respostas de requisições devem ser fornecidas no momento apropriado, ou seja, levando o menor tempo possível.

2.3.4 Escalabilidade

É a capacidade que o sistema tem de alocar mais recursos como escalar verticalmente aumentando a quantidade de *hardware* com a aquisição de equipamentos mais potentes (TIWARI, 2011). Porém, este processo de personalizar as máquinas, tornando-as aptas para conseguir atender ao aumento de dados e usuários, apresenta custos elevados (SADALAGE e FOWLER, 2013). Portanto, é viável utilizar a escalabilidade horizontal que por meio de um *cluster*, novos *nodes* compostos por máquinas mais modestas podem ser adicionados, sendo diminuídos custos de aquisição e manutenção (TIWARI, 2011).

2.3.5 Transparência

Acontece quando os componentes que fazem parte de uma aplicação, mesmo atuando de forma separada são vistos pelo usuário como se estivessem sendo

executados em um único sistema (COULOURIS *et al*. 2012). Isso acontece em sistemas que realizam processamento de transações que podem ser executadas de forma concorrente, fazendo parecer que apenas uma transação foi realizada. Por outro lado, Singhal, Dhameja e Panda (2018) trazem uma semântica diferente para a transparência em sistemas distribuídos descentralizados, pois nesse caso é importante que as informações sejam transparentes aos usuários que fazem parte da rede.

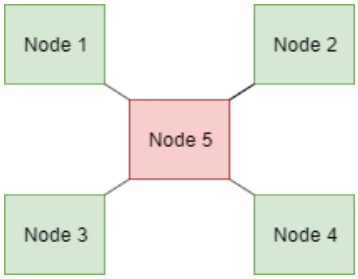
2.3.6 Qualidade de Serviço

Representa o *feedback* emitido pelo usuário de um determinado serviço ao mostrar os seus pontos positivos e negativos para que projetistas possam realizar uma melhoria contínua. Para Coulouris *et al*. (2012), parâmetros como segurança, desempenho, propensão a mudanças de configurações e capacidade de alterar recursos, protagonizam um serviço de qualidade.

2.4 SISTEMAS DISTRIBUÍDOS COM CONTROLE CENTRALIZADO

Os sistemas centralizados são aqueles que possuem um único *node*, responsável por fazer a comunicação com os demais para realizar atividades como o gerenciamento de dados (SADALAGE e FOWLER, 2013). A *Amazon* e a grande maioria das plataformas e aplicações *Web* são *softwares* centralizados, ou seja, existe um controlador que gerencia as operações feitas pelas outras unidades com um fluxo de informações único (RAVAL, 2016). De acordo com Singhal, Dhameja e Panda (2018) os *softwares* que adotam o modelo centralizado são projetados e mantidos sem grandes dificuldades e tornou-se o padrão adotado para desenvolver sistemas. A Figura 7 mostra a estrutura de comunicação de um sistema centralizado em um ambiente distribuído.

Figura 7 - Sistema distribuído com controle centralizado

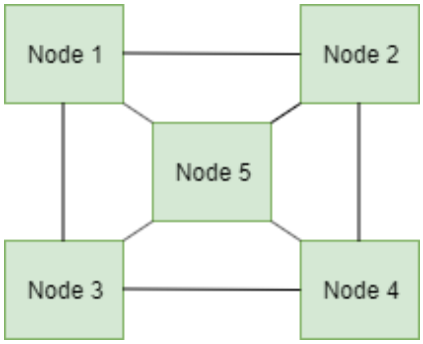


Fonte: Elaborado pelo Autor (2020)

2.5 SISTEMAS DESCENTRALIZADOS

Em sistemas descentralizados, não há um *node* responsável por delegar tarefas para outros que fazem parte da rede e, além disso, é distribuído por residir informações em vários computadores (RAVAL, 2016). A Figura 8 apresenta a estrutura desses sistemas. A *blockchain* consiste em uma tecnologia que registra as transações em um livro razão, ou seja, um local cujo conteúdo não pode ser adulterado e estas transações são replicadas para um grande número de *nodes*, sem a necessidade de terceiros (SINGHAL; DHAMEJA; PANDA, 2018). Logo, a *blockchain* emergiu como uma nova maneira para realização de transações *online* ao utilizar uma estratégia diferente dos sistemas centralizados.

Figura 8 - Sistema distribuído e descentralizado

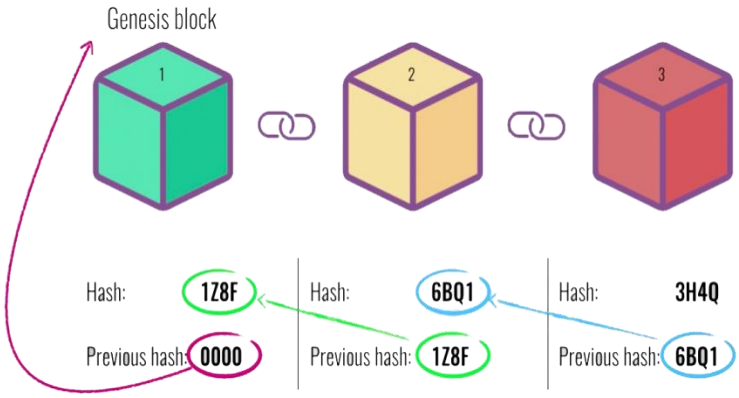


Fonte: Elaborado pelo Autor (2020)

A moeda digital *Bitcoin* foi pioneira em fazer uso da *blockchain* que por meio de uma *Peer-to-peer* (*i.e.,* rede descentralizada) os participantes realizam de forma prática e segura o armazenamento e troca de valores (ANTONOPOULOS, 2017). Diante disto, a *blockchain* trabalha com base na cooperação e confiança entre as participantes ao realizar transações transparentes, pois podem ser auditadas pelos próprios *nodes* da rede. Além disso, como todas as movimentações financeiras são monitoradas em um livro razão, a tecnologia *blockchain* é forte para tratar questões relacionadas à veracidade em *Big Data* (PENDYALA, 2018). Com a grande quantidade de dados trafegados na *Web*, a incerteza de que as informações geradas vão ser corretas é recorrente. Logo, é preciso além dos aspectos operacionais de como essas transações são gerenciadas por meio de um determinado *software*, é preciso considerar o fator humano. Então, existe a possibilidade de ocorrer interferências

diretas ao realizar acessos indevidos em um ambiente distribuído. Para tratar questões de segurança e gerar confiança nas operações, a *blockchain* faz uso de criptografia, tornando possível que apenas pessoas autorizadas façam parte da rede e a confidencialidade seja mantida ao mascarar as identidades (LAURENCE, 2019). A Figura 9 mostra o funcionamento da *blockchain*, que é composta por blocos interligados pela função *hash* (*i.e.,* responsável por verificar a integridade dos dados armazenados). Sendo assim, cada bloco tem o *hash* do seu antecessor e o valor zero corresponde ao *hash* do primeiro bloco (*i.e.,* bloco gênesis). Além disso, os dados são alocados nos blocos por meio de transações que são transmitidas para todos os *nodes* participantes da rede. Portanto, existe uma rede descentralizada contendo transações seguras e transparentes, uma vez que se um usuário mal-intencionado realizar alterações indevidas em um bloco, toda a cadeia ficará inválida e os *nodes* saberão que a rede foi comprometida.

Figura 9 - Funcionamento da *blockchain*



Fonte: (DECUYPER, 2017)

Devido ao sucesso da *Bitcoin* o uso da tecnologia *blockchain* está sendo aplicada em outros contextos. Singhal, Dhameja e Panda (2018) relatam que além de finanças, a *blockchain* é adotada em áreas como entretenimento, redes de varejo, seguros, cadeia de suprimentos e registro médico. As empresas que queiram aderir à tecnologia de *blockchain* devem estar dispostas a desburocratizar algumas operações realizadas por *stakeholders*, pois será colocado em prática um novo modelo de negócio, uma vez que a figura do controlador central é substituída por aqueles que fazem parte da rede. Logo, a Figura 10 ilustra uma transação para transferir valores

dentro de um contexto bancário tradicional com um sistema centralizado em relação a uma rede *Peer-to-peer* descentralizada.

Figura 10 - Transferência entre valores: intermediário central X rede *Peer-to-peer*.

Fonte: Adaptado de (SINGHAL; DHAMEJA; PANDA, 2018)

No modelo tradicional centralizado, o cliente João ao transferir o valor 100 para Maria sempre dependerá de um terceiro para que essa operação seja concluída. Além disso, do ponto de vista do *software*, existe um sistema engessado com pouca interação entre os componentes que poderiam melhorar a distribuição e replicação dos dados. Já na perspectiva da *blockchain*, o cliente João ao enviar o valor 100 para Maria, acontece a atuação dos demais *nodes* da rede *Peer-to-peer* como agentes reguladores, pois há interações para que possam entrar em consenso e validar ou não a transação. Portanto, sem a presença de uma autoridade central, para que as transações ocorram de maneira segura é preciso um mecanismo de validação. Logo, segundo Antonopoulos (2017), todos os *nodes* tem o registro completo da rede e devem entrar em consenso com relação aos dados que são adicionados. Para isto, de acordo com Singhal, Dhameja e Panda (2018) a tecnologia *blockchain* utiliza algum protocolo de consenso e, dentre eles, destacam-se: Prova de Trabalho (PoW), Prova de Participação (PoS) e Tolerância Prática a Falhas Bizantinas (PBFT). A seguir, são apresentados os conceitos de cada um deles:

- Prova de Trabalho (PoW)

  Este é o padrão adotado pela maioria das plataformas de criptomoedas e tem como característica evitar o gasto duplo, ou seja, quando um usuário mal-

intencionado da rede faz duas transações e nega que a primeira aconteceu (RAVAL, 2016). Logo, com o PoW as transações só serão adicionadas ao bloco e começam a fazer parte da rede quando verificadas por alguns *nodes* (*i.e.,* mineradores) (SINGHAL; DHAMEJA; PANDA, 2018). Segundo Antonopoulos, (2017) os mineradores oferecem uma grande força de trabalho computacional para acharem a solução do algoritmo de *hash SHA-256*.

- Prova de Participação (PoS)

  A ideia deste mecanismo de consenso de acordo com Raval (2016) é que os participantes com mais criptomoedas são aqueles que de fato querem ter uma rede segura, devido ao investimento feito. Portanto, não acontece o processo de mineração, apenas validação dos blocos referentes às transações e isso faz o PoS não ser tão seguro quando o PoW (SINGHAL; DHAMEJA; PANDA, 2018).

- Tolerância Prática a Falhas Bizantinas (PBFT)

  Cada node possui seu próprio estado interno e ao receber requisições compartilham suas características com outros participantes para chegar a um consenso e distribuir o resultado entre todos os envolvidos (SINGHAL; DHAMEJA; PANDA, 2018)

2.6 FRAMEWORKS

De acordo com Barbosa (2001), os *frameworks* consistem em sistemas que visam conceder a estrutura necessária para que outros *softwares* derivados deles sejam construídos. Logo, os usuários que utilizam um determinado *framework* esperam ganho de produtividade devido a reutilização de partes do sistema podendo estender as funcionalidades para atender a um domínio específico. Pree e Gamma (1995) descrevem conceitos importantes que estão relacionados a estruturação e comunicação dos componentes pertencentes a um *framework*, a saber: *frozen spots* e *hot spots*. Os *frozen spots* fazem a definição geral da arquitetura, não permitindo alterações que possam vir a ocorrer para possíveis adaptações. Por outro lado, *os hot spots* permitem a inclusão de novas *features* (*i.e.,* recursos ou funcionalidades) para conseguir atender àquilo que a aplicação necessita.

Ao construir um *framework* é preciso determinar qual o seu propósito, ou seja, de que forma ele vai conseguir auxiliar os usuários a desenvolver aplicações mais

rapidamente e de forma robusta. Existem diversos *frameworks* disponíveis na literatura e no mercado, sendo que Fayad, Schmidt e Johnson, (1999) fazem a seguinte classificação: infraestrutura de sistemas, integração e aplicação empresarial. Os *frameworks* de infraestrutura de sistemas são voltados para sistemas operacionais e linguagens de programação. Por outro lado, os *frameworks* de integração são *middlewares* que conseguem realizar a comunicação de componentes em um ambiente distribuído. Por fim, os *frameworks* empresariais atendem as demandas de grandes domínios de aplicação como manufatura e telecomunicação.

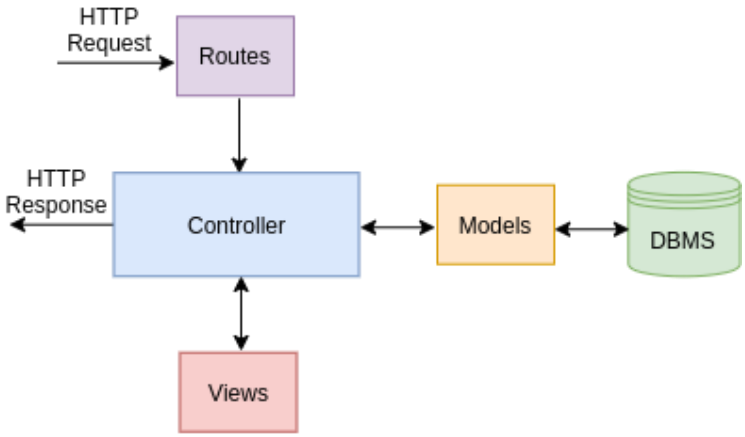
Por se tratar de um *software*, o *framework* para ter maior probabilidade de conseguir atender àquilo que é designado, é essencial passar por algumas fases que Barbosa (2001) define como sendo: desenvolvimento do *framework*, implementação da aplicação, manutenção e evolução do *framework*. Sendo assim, observa-se que é fundamental definir os papéis que vão ser desempenhados por cada componente que faz parte do *framework* e como vai ser realizada a comunicação entre eles, a fim de manter a forma correta de execução do sistema. Ao construir um componente, o ideal é pensar em como ele pode ser reaproveitado para evitar a criação de um novo. Portanto, o usuário que faz uso de um *framework* para desenvolver aplicações para um domínio específico ganha em produtividade ao ter uma estrutura predefinida, com componentes propícios para realização de tarefas rotineiras como construir uma determinada aplicação com acesso ao BD ou realizar manipulação de arquivos.

2.6.1 Modelo, Visão e Controle

Introduzido por Burbeck (1992), o acrônimo *Model-View-Controller*, ou seja, Modelo, Visão e Controle (MVC) tornou-se um padrão arquitetural para desenvolvimento de soluções computacionais, o qual visa separar a parte de apresentação da lógica do negócio. O *Model* faz a comunicação com os DBMS (*i.e.,* inglês de SGBD) ao manipular estruturas de dados como tabelas ou documentos. O *Controller* é o responsável por receber as requisições dos clientes para a aplicação e contém a regra de negócio necessária para retornar aquilo que foi pedido. A View é o último componente da tríade, sendo responsável pela apresentação que pode ser exibida pela Linguagem de Marcação de Hipertexto (HTML) com auxílio do formato JSON. Para que a comunicação entre os componentes aconteça é preciso fazer uso das *Routes* que por meio do Protocolo de Transferência de Hipertexto (HTTP)

direcionam as solicitações para um determinado recurso. A Figura 11 mostra os componentes da arquitetura e a comunicação entre eles.

Figura 11 - Fluxo de componentes presentes na arquitetura MVC



Fonte: Elaborado pelo Autor (2020)

2.7 DESIGN PATTERNS

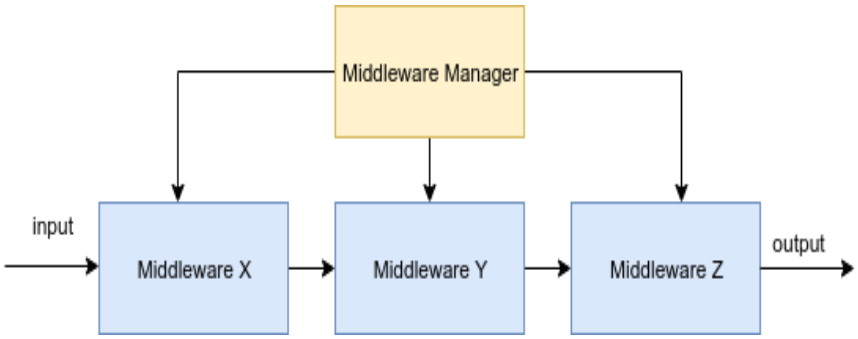
Segundo Stefanov (2010), os *design patterns* surgiram com o objetivo de facilitar a resolução de problemas recorrentes ao designar padrões para construção de soluções que comtemplem domínios específicos. É frequente associar este termo com o paradigma de orientação a objetos para desenvolvimento de sistemas. Ao lidar com *JavaScript*, a linguagem adotada para a implementação do *framework* proposto neste trabalho, não parece ser trivial adotar um padrão. No entanto, o *JavaScript* é multiparadigma e, por isso, tem a flexibilidade para provê padrões consistentes a serem seguidos. Além disso, Casciaro (2014) afirma que o *JavaScript* não possui classes e *interfaces* abstratas, mas os conceitos e meios que um determinado *pattern* provê para construir soluções são o que de fato vão importar. A seguir, serão listados os *patterns* que servirão de auxílio no processo de desenvolvimento do *framework* desta dissertação.

2.7.1 Middleware Pattern

A documentação do *framework Express* (2019) especifica que o *Middleware Pattern* é a base de toda aplicação construída e consiste em funções que recebem os parâmetros *req* e *res* para manipular os dados da requisição e resposta

respectivamente. Esses parâmetros podem estar presentes nas *Routes* ou globalmente, ou seja, independente da rota acessada, o *middleware* sempre vai ser chamado. Portanto os *middlewares* são interceptadores que realizam atividades como geração de *log* para saber quais são as *Routes* que estão sendo chamadas pelo cliente. De acordo com Casciaro (2014), o *Middleware Pattern* herdou características de outros dois, o *Intercepting Filter Pattern* e o *Chain of Responsibility Pattern*. Além disso, pode ser representado como uma *pipeline* e possui um coordenador central denominado Gerenciador de *Middleware* (MM), conforme ilustrado na Figura 12. Assim, o *Middleware Pattern* é composto por diversos *middlewares* que vão desempenhar as tarefas que forem designadas para eles por meio de uma *pipeline* de execução e são coordenados pelo MM. Portanto, os dados são direcionados para um determinado *Middleware* que pode interagir com outros para auxiliar na execução de tarefas e fornecer a saída esperada para quem requisitou um recurso específico. Ademais, seguindo essa estrutura, pode ser alocado um conjunto de *middlewares* derivados de outros, mas que contém funções específicas para atender a um determinado domínio.

Figura 12 - Componentes do *Middleware Pattern*



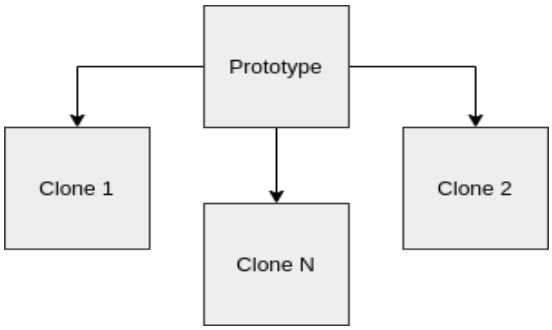
Fonte: Adaptado de (CASCIARO, 2014)

2.7.2 Prototype Pattern

O *Prototype Pattern* é útil para criar objetos que são baseados em um *template* de outro objeto previamente existente (OSMANI, 2012). Portanto, funciona como uma herança na qual pode fornecer propriedades para criação de outros objetos, não precisando ficar criando cada instância manualmente. A Figura 13 ilustra por meio de

um diagrama o objeto principal denominado *Prototype*, que instância outros objetos clones, ou seja, que vão possuir as mesmas características.

Figura 13 - O objeto *Prototype* e suas instâncias



Fonte: Elaborado pelo Autor (2020)

2.8 CONSIDERAÇÕES FINAIS

Este capítulo discorreu sobre os conceitos necessários para entender a proposta do *framework MongoChain*. No próximo capítulo serão apresentados os trabalhos relacionados presentes no estado da arte e que visam fornecer soluções para integrar tecnologias que consigam lidar com gerenciamento de transações em ambientes distribuídos e descentralizados.

3 TRABALHOS RELACIONADOS

Este capítulo aborda os principais trabalhos que compõem o estado da arte referente às tecnologias para construção, execução e registro de transações. Assim, foram selecionadas soluções que realizam integrações entre: transações ACID, tecnologia *blockchain* e SGBD NoSQL de agregados.

3.1 MEGASTORE

A proposta elaborada por Baker (2011) mostra que ocorre uma partição do BD e cada gravação é replicada de forma síncrona para uma grande rede por meio do *Paxos*, que consiste em um algoritmo de consenso. As garantias ACID acontecem em cada partição e utiliza-se o SGBD NoSQL *Bigtable* que possui recursos transacionais em um determinado grupo hierárquico. Os programadores são os responsáveis em fazer o vínculo entre grupo e item de dados, sendo que cada transação pode ter acesso a apenas um único grupo. Ademais, o controle de transações simultâneas é baseado no algoritmo MVCC.

3.2 CLOUDTPS

Wei, Pierre e Chi (2011) visam fornecer garantias transacionais para qualquer tipo de SGBD NoSQL. Os testes foram realizados nos SGBD *Hbase*, que é uma versão *open source* do *Bigtable* e no *Amazon SimpleDB*. Para garantir as propriedades ACID quando as aplicações *Web* realizam transações em múltiplos itens de dados são utilizados vários servidores. Cada servidor é denominado Gerenciador de Transações Locais (LTM) e possui uma cópia da partição para receber os itens de dados que são replicados. Assim, suportam o aumento na demanda e se tornam tolerantes às falhas. Nesta proposta utiliza-se o Protocolo de Confirmação de Duas Fases (2PC), que primeiro define um LTM como coordenador e solicita que todos os demais LTM verifiquem se a operação foi executada com êxito. Logo, existe uma votação entre eles e caso a maioria vote favoravelmente, realiza-se a segunda fase que basicamente é o *commit* da transação, caso contrário ocorre um *abort*. Uma transação é decomposta em sub-transações que podem entrar em conflito ao acessarem um determinado item de dados ao mesmo tempo. Para resolver isto, utiliza-se um *timestamp* em cada transação conflitante, para serem executadas

apenas se o registro de data e hora seja inferior em relação as transações que tiverem sido concluídas. Logo, ocorre a execução dessas sub-transações de forma sequencial. Os resultados apontam que a consistência forte foi mantida executando transações de forma local com o *HBase* e na nuvem utilizando o *Amazon SimpleDB*.

3.3 PERCOLATOR

O trabalho apresentado por Peng e Dabek (2010) mostra que os motores de busca como o *Google* conseguem indexar informações e rapidamente disponibilizar o conteúdo que os usuários procuram. Como a alta demanda de requisições requer um sistema altamente disponível, ocorre escalabilidade das bases de dados para diversos servidores e isto ocasiona desafios inerentes à infraestrutura. Estes dados podem estar armazenados em um repositório gerenciado por um SGBD, porém este tipo de *software* não é capaz de lidar com *petabytes* de dados como o sistema de indexação do *Google* armazena. A solução seria fazer uso do *Bigtable*, mas devido a problemas com atualizações simultâneas foi desenvolvido o *Percolator* que atua sobre o *Bigtable* e fornece transações que são compatíveis com ACID por meio do algoritmo de controle de concorrência SI. Além disso, utiliza-se o *Oracle Timestamp* (TO) para realizar o registro de data e hora em forma crescente e garante a operação correta do algoritmo SI. É realizado também a exclusão mútua que ajuda a detectar conflitos quando operações de gravações estão sendo realizadas. Porém, as operações somente leitura acabam adquirindo bloqueios, responsáveis por ocasionar queda no desempenho e não fornecer uma técnica de prevenção de *deadlocks*. Além disso, os metadados das transações são armazenados juntos com os dados provenientes dos SGBD. Ademais, a sobrecarga acaba ocorrendo devido ao intenso envio de requisições aos servidores e atraso no processamento das transações.

3.4 OMID

O trabalho proposto por Ferro *et al.* (2014) visa promover suporte transacional para o *HBase* por meio do algoritmo SI e não faz uso de bloqueios. É utilizado o *Status Oracle* de Transação (TSO) para manter os metadados que são necessários para detectar os conflitos e realiza-se o monitoramento dos *commits* feitos em todas as transações. Para amenizar o uso da memória, os metadados são truncados, ou seja, apenas as alterações mais recentes são mantidas. Além disso, uma cópia somente

leitura dos metadados é reproduzida relacionada as transações realizadas por clientes e atende a consultas locais. Ademais, o OMID pode ser aplicado em típicos SGBD chave-valor e não requer modificações direta no armazenamento.

3.5 A MIDDLE LAYER SOLUTION

O trabalho realizado por Lotfy *et al.* (2016) consiste em uma camada de *middleware* que visa garantir a consistência dos dados em SGBD NoSQL por meio de um protocolo de *commit* de quatro fases, que consistem em: utilização de um coordenador para coleta de dados residentes na camada de *middleware* ou no SGBD NoSQL; execução de uma transação por outro coordenador em paralelo no caso de ocorrência de falha do primeiro; realização de transações pelos dois coordenadores; o coordenador primário envia esses dados para a aplicação *Web* que fez a requisição e também ao SGBD NoSQL. Logo, resta ao secundário verificar se tudo ocorreu como previsto. Se algo estiver errado, o coordenador secundário torna-se o responsável por fornecer uma resposta para a aplicação *Web* e os dados não são persistidos no SGBD NoSQL.

É utilizado uma combinação entre o mecanismo de bloqueio e o *timestamp* para garantir a execução de transações concorrentes. Com isto, a chave de um item de dados possui uma versão e, na ocorrência de atualização sobre ela, ocorre o bloqueio. Quando chega ao término da transação, a chave é desbloqueada e uma nova versão é criada. A característica de escalabilidade também está presente, pois quando há aumento no número de camadas devido ao incremento na carga de trabalho, o balanceador de carga entre em ação para manter o equilíbrio do sistema. Os experimentos foram realizados utilizando o SGBD *MongoDB*.

3.6 PH1

A solução proposta por Coelho *et al.* (2014) visa intermediar as solicitações vindas dos clientes, oferecendo garantias transacionais ao realizar operações em qualquer SGBD NoSQL. Para isso, utiliza-se o SI que é alcançado devido a integração dos módulos Armazenamento de Versão Não Persistente (NPVS) e TSO que é o mesmo certificado utilizado pela ferramenta OMID que foi vista anteriormente. Logo, ao serem criadas transações, o TSO vai provê *timestamps* e acompanhar o estado das transações, ou seja, se estão sendo executadas com êxito ou sofreram

intercorrências durante o processo. Por outro lado, o NPVS tem como função manter os registros dos dados versionados e realizar a replicação para recuperação na ocorrência de falhas. Existe um conjunto de escrita determinado para armazenar as operações referentes a gravação, atualização e remoção do item de dados. O Gerenciador de Transação (TM) é quem faz a coordenação desse processo, sendo também responsável por atender às solicitações dos clientes por transações ao fazer o controle de início, fluxo de leitura ou gravação e término, em conjunto com o TSO. Todas as transações podem ser iniciadas, pois não há uma política de bloqueio proferida pelo TSO que poderia impedir a transação de obter o *timestamp*. Quando o número de clientes vai aumentando, o *PH1* consegue escalar ao fazer a criação de novas instâncias do sistema. O *PH1* foi projetado para ser utilizado com qualquer SGBD NoSQL. Os experimentos realizados envolveram os SGBD *Cassandra* e o *Hyperdex*.

3.7 MONGODB

De acordo com a documentação do *MongoDB* (2019), as operações são atômicas em um único documento e, a partir da versão 4.0 lançada em 2018, o suporte à transação foi aderido por meio da manutenção das propriedades ACID para leituras e escritas em múltiplos documentos. Logo, ao executar transações, a consistência dos dados é garantida seja em uma ou várias coleções de documentos. Além disso, os dados ficam em *Replica Sets* (*i.e.,* instâncias com cópias de dados) e, na versão 4.2, o *MongoDB* consegue escalabilidade a nível de *sharding* (*i.e.,* fragmentação do *cluster* ao dividir os dados nas instâncias e *replica sets*). O *MongoDB* usa sessões para gerenciar transações, ou seja, ao executar uma transação as alterações não confirmadas ficam visíveis apenas no contexto da sessão. Assim, o algoritmo de controle de concorrência SI que é utilizado pelo *MongoDB* garante que em operações de escrita no BD, os dados lidos fora do contexto da sessão não serão visualizados. Portanto, uma transação estará visível fora da sessão apenas quando ocorrer o *commit* e, se houver falha, a transação é abortada e os dados da sessão são descartados.

O *MongoDB* pode ser configurado para controlar a consistência ao lidar com o isolamento de leitura (*Read Concern*). Para isto, são disponibilizados os seguintes níveis:

- *Local*

  Ao realizar uma consulta, os dados são retornados sem a garantia de que a gravação foi realizada na maioria dos membros das réplicas.
- *Available*

  A diferença deste para o *Local* é que a leitura acontece no *cluster* secundário e o *Local* no primário.
- *Majority*

  Neste caso, a consulta vai retornar os dados que foram reconhecidos pela maioria dos conjuntos de réplicas.
- *Linearizable*

  Retorna os dados que foram das gravações reconhecidas como bem sucedidas pela maioria dos membros.
- *Snapshot*

  É garantido que as operações de transação tenham lido de uma captura instantânea de dados confirmado pela maioria.

  Os níveis de escrita (*Write Concern*) também podem ser configurados:
- *w:1*

  A operação de gravação foi direcionada para o *node* primário em um conjunto de réplicas.
- *w: Majory*

  As operações de gravação foram disseminadas para a maioria dos votantes, sendo primários e secundários.

Para executar transações no *MongoDB* estão disponíveis dois níveis: *Read Concern* para leitura com o *Snapshot Isolation* e para a escrita pode-se utilizar o *Write Concern w*:1 ou *Majority*.

3.8 RAVENDB

O *RavenDB* é um SGBD baseado em documentos e todas as operações de escrita são executadas com manutenção das propriedades ACID por meio de transações (EINI, 2018). Para isto, segundo a sua documentação RavenDB (2019) é utilizado um mecanismo de armazenamento interno denominado *Voron* que grava todas operações de modificações em um arquivo de *buffer* que é sincronizado com o

armazenamento de dados principal. Assim, se houver falha antes do *commit*, as informações são recuperadas, sendo que as transações presentes no *buffer* serão salvas no armazenamento principal. Ademais, utiliza o SI como algoritmo de controle de concorrência.

3.9 BIGCHAINDB

O *BigchainDB* tem características presentes nos SGBD tradicionais como taxa de transferência e latência que são integradas com propriedades de *blockchain*: descentralização e imutabilidade (MCCONAGHY *et al.* 2016). Os dados no *BigchainDB* são estruturados como ativos que representam objetos físicos ou digitais. Além disso, a partir da versão 2.0 do SGBD foi incluido o suporte à falha de um terço dos *nodes* por meio do protocolo de consenso PBFT (BIGCHAINDB, 2018). Isso se deve ao uso da rede *Tendermint* que realiza a comunicação entre os *nodes*. Cada *node* possui uma BD *MongoDB* local que realiza operações atômicas em um único documento sem implementação de algoritmo para controle de concorrência, ou seja, utiliza as configurações padrão do *MongoDB* com nível de leitura *Local* e escrita w:1. Portanto, não há garantia que os dados lidos são consistentes, uma vez que não foram confirmados pela maioria dos *nodes*.

3.10 HYPERLEDGER FABRIC

De acordo com Blummer *et al*. (2018), o *Hyperledger Fabric* é uma plataforma com foco na indústria e diverge da *Bitcoin* por provê uma rede *blockchain* permissionada. Além disso, a *Hyperledger Fabric* usa o protocolo de consenso PBFT que por meio de uma política de votação entre os pares verifica as transações de leitura e escrita (GAUR, 2018). Ademais, Gaur (2018) menciona que para a construção de *Smart Contracts* (*i.e.,* acordos comerciais programáveis executados por transações e armazenados em BD) podem ser utilizadas linguagens de programação como *Javascript*, *Java* e *Go*, tendo as transações armazenadas no *LevelDB*, um SGBD NoSQL chave-valor. A *Hyperledger Fabric* pode ser configurada para armazenar transações no SGBD NoSQL de agregados *CouchDB*. No entanto, o *CouchDB* não realiza transações ACID, pois trabalha apenas com operações atômicas de escrita em um único documento. Para operações de leitura faz uso do algoritmo de controle de concorrência MVCC (COUCHDB, 2019).

3.11 ANÁLISE COMPARATIVA

Os trabalhos apresentados evidenciam a existência de soluções que integram ACID, BASE e *blockchain*. Assim, *middlewares* estão sendo construídos para atuar como *interface* entre clientes que executam transações em SGBD NoSQL de agregados. Logo, esta camada intermediária é responsável por manter as propriedades ACID ao gerenciar transações com técnicas de controle de concorrência para operações de escrita e leitura no BD. Ademais, alguns SGBD NoSQL de agregados já fornecem recursos para realização de transações ACID. Por fim, nota-se o uso da tecnologia *blockchain* junto com SGBD NoSQL de agregados para construção de aplicações distribuídas e descentralizadas. O *framework* proposto, *MongoChain*, utiliza recursos transacionais do SGBD *MongoDB* e possui uma rede descentralizada na qual as transações são executadas e validadas para serem armazenadas em uma *blockchain* permissionada. O Quadro 3 apresenta uma comparação entre os trabalhos relacionados e o *MongoChain*, de acordo com os seguintes critérios:

I. Suporte a transações ACID.
II. Utiliza recursos transacionais de SGBD NoSQL de agregados.
III. Armazena transações na *blockchain*.
IV. É um *framework* ou plataforma programável.

Os *middlewares* oferecerem suporte para transações ACID em SGBD NoSQL de agregados, mas este tipo de funcionalidade já se encontra presente nos SGBD *MongoDB* e *RavenDB*. Por outro lado, os SGBD *MongoDB* e *RavenDB* não fazem uso da tecnologia *blockchain*. Logo, o *BigchainDB* tem propriedades de *blockchain*, mas por padrão, não é configurado para fazer uso dos recursos transacionais do *MongoDB* e não possui um ambiente programável para auxiliar os desenvolvedores na construção de aplicações. Ademais, a plataforma *Hyperledger Fabric* apesar de ser um *framework* para construção de aplicações descentralizadas, não utiliza um SGBD com suporte transacional como padrão e, ao mudar para o SGBD *CouchDB*, não consegue provê transações ACID para gravações em múltiplos documentos. Com isto, o diferencial do *MongoChain* em relação aos trabalhos relacionados é por ser um *framework* programável que consegue provê suporte para transações ACID e faz uso da tecnologia *blockchain* em um modelo de dados NoSQL de agregados.

Quadro 3 - Análise comparativa entre trabalhos correlatos e *MongoChain*

Trabalho	**Suporte a transações ACID**	**Utiliza recursos transacionais de SGBD NoSQL de agregados**	**Armazena transações na *blockchain***	***É* um *framework* ou plataforma programável**
Megastore	Sim	Não	Não	Não
CloudTPS	Sim	Não	Não	Não
Percolator	Sim	Não	Não	Não
OMID	Sim	Não	Não	Não
A Middle Layer Solution	Sim	Não	Não	Não
PH1	Sim	Não	Não	Não
MongoDB	Sim, se configurado	Sim	Não	Não
RavenDB	Sim, se configurado	Sim	Não	Não
BigchainDB	Sim, se configurado	Sim, se configurado	Sim	Não
Hyperledger Fabric	Não	Sim, se configurado	Sim	Sim
MongoChain	Sim	Sim	Sim	Sim

Fonte: Elaborado pelo Autor (2020)

3.12 CONSIDERAÇÕES FINAIS

Este capítulo apresentou soluções que gerenciam transações ACID para serem armazenadas nos SGBD NoSQL de agregados ou que possam ser alocadas em uma rede *blockchain* descentralizada. Ademais, foi verificado por meio de uma análise comparativa que o *MongoChain*, além de ser um *framework* programável, consegue comtemplar mais critérios de integração entre tecnologias divergentes do que as soluções presentes no estado da arte. O próximo capítulo descreve a arquitetura, especificação e especializações do *MongoChain*.

4 O FRAMEWORK MONGOCHAIN

Este capítulo apresenta o *framework* programável *MongoChain* e descreve sua arquitetura, especificação e especializações. Com código fonte[1] e API[2] disponíveis na *Internet*, o *MongoChain* auxilia os desenvolvedores na construção de sistemas transacionais customizados para diferentes domínios. Assim, transações seguras e transparentes são executadas e adicionadas a uma *blockchain* com o auxílio do protocolo de consenso PoW. Além disso, a consistência dos dados é garantida ao executar transações que mantêm as propriedades ACID quando afetam múltiplos documentos do *MongoDB*. Ademais, faz uso do algoritmo de controle de concorrência SI para leitura e escrita de dados. Por fim, a disponibilidade é obtida devido aos dados estarem persistidos e replicados em um *cluster* na nuvem, sendo que para conseguir ter escalabilidade horizontal basta adicionar mais *nodes*.

4.1 VISÃO ARQUITETURAL

O *MongoChain* é baseado no padrão arquitetural MVC. Logo, possui uma estrutura flexível para implementar regras de negócios e fazer reuso dos componentes de *software*. Além do MVC, foi utilizado no desenvolvimento do *MongoChain* um *Design Pattern* (*i.e., Middleware Pattern*) para gerenciar requisições de dados internas e solicitadas por aplicações externas. Assim, uma API foi desenvolvida para permitir que os serviços especificados no *MongoChain* possam ser consumidos. Por meio de requisições HTTP com métodos *POST, GET, PUT e DELETE*, são processadas operações de escrita e leitura de dados (*i.e.,* respectivamente *create*, *read*, *update* e *delete*) no *MongoChain*.

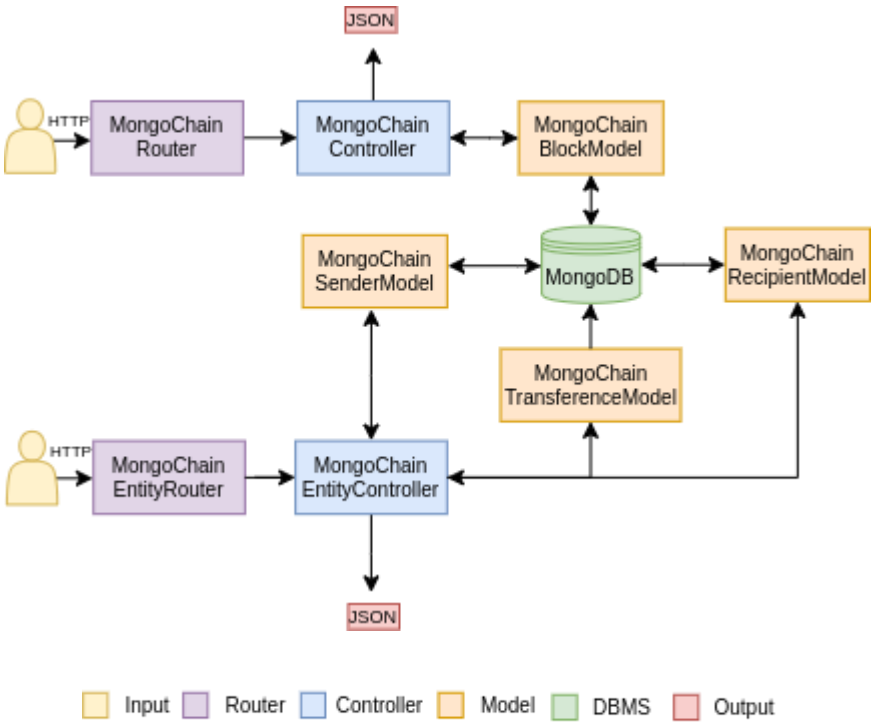
A Figura 14 mostra a arquitetura do *MongoChain*, que possui duas *Routes* responsáveis por direcionar os *inputs* de requisições dos clientes da aplicação para os *Controllers*. Deste modo, o *MongoChainRouter* fornece acesso ao componente principal do *MongoChain*, o *MongoChainController*, que usa *middlewares* para criar e registrar transações de leitura e escrita. Quando o *middleware storeBlockchainMongo* se comunica com o componente *ModelChainBlockModel*, a estrutura de objetos do *MongoChain* é mapeada para os documentos no *MongoDB* utilizando o Mapeamento

[1] https://github.com/carlosmvs/MongoChain

[2] https://sites.google.com/view/mongochain/api

de Documentos e Objetos (ODM). Assim, a coleção *blocks* é criada e contém um documento com o início da *blockchain* (*i.e.,* bloco gênesis). Então, um *node* da rede descentralizada adiciona outros por meio do *middleware storeBroadcastNode*. Cada *node* terá uma cópia da *blockchain*, a mesma que está persistida na coleção *blocks*.

Figura 14 - Arquitetura do *MongoChain*



Fonte: Elaborado pelo Autor (2020)

O *MongoChain* é composto pelas entidades *Sender*, *Recipient* e *Transference*. Logo, o componente *MongoChainEntityRouter* é responsável por receber requisições HTTP e interagir com o *MongoChainEntityController*, que possui *middlewares* para gerenciar operações de leitura e escrita nos dados, produzindo *outputs* no formato JSON. Assim, as propriedades das entidades *Sender* e *Recipient* (*i.e., id*, *name*, *amount* e *createdAt*,) são mapeadas pelos *Models* (*i.e., MongoChainSenderModel* e *MongoChainRecipientModel*) para serem persistidas em coleções do *MongoDB* (*i.e., senders* e *recipientes*) por meio de uma requisição HTTP *POST*. A entidade *Transference* é composta pela sua propriedade identificadora junto com as das coleções *senders* e *recipientes* (*i.e., id*, *senderId* e *recipientId*). As demais propriedades são: *status* que por padrão é configurado com o valor "*requested*" e *createdAt* para salvar o horário da requisição. Deste modo, estas propriedades são

mapeadas pelo *MongoChainTransferenceModel* tornando-se a coleção *transferences*. O Quadro 4 sumariza as entidades e propriedades presentes no *MongoChain*.

Quadro 4 - Entidades e propriedades do *MongoChain*

Sender	***Transference***	***Recipient***
id *name* *amount* *createdAt*	*id* *senderId* *recipientId* *status* *createdAt*	*id* *name* *amount* *createdAt*

Fonte: Elaborado pelo Autor (2020)

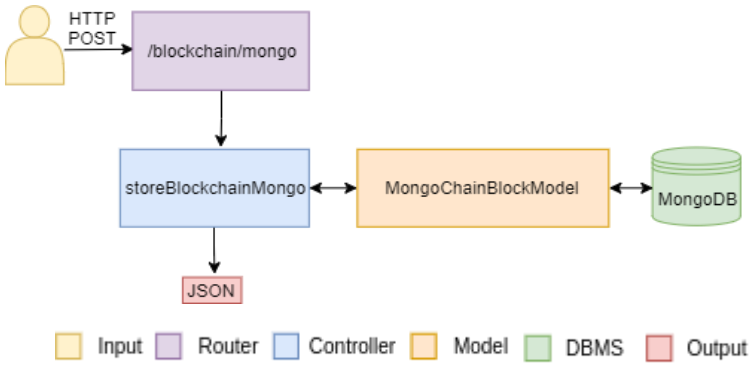
4.2 ESPECIFICAÇÃO

Para auxiliar na visualização dos fluxos de atividades realizadas pelos componentes do *MongoChain*, optou-se por não utilizar diagramas da Linguagem de Modelagem Unificada (UML) e sim outros mais genéricos. Isto acontece devido aos *patterns* utilizados pelo *MongoChain* (*i.e., Middleware* e *Prototype*) terem, segundo Osmani (2012), implementações baseadas em objetos e funções ao contrário de classes. Os *Controllers MongoChainController* e *MongoChainEntityController* são os componentes responsáveis pela lógica de negócio e utilizam um conjunto de *middlewares* que são interceptadores de requisições. Serão apresentados a seguir, os *middlewares* que compõem o *MongoChain*.

4.2.1 Middleware storeBlockchainMongo

A Figura 15 mostra que o usuário ao acessar a API do *MongoChain* pela rota */blockchain/mongo* via método HTTP *POST* aciona o *middleware storeBlockchainMongo*, que cria a coleção *blocks* no *MongoDB* ao mapear o bloco em formato de objeto por meio do *MongoChainBlockModel* e converte em um documento que representa o bloco gênesis da *blockchain*.

Figura 15 - Criação da coleção *blocks* e do bloco gênesis

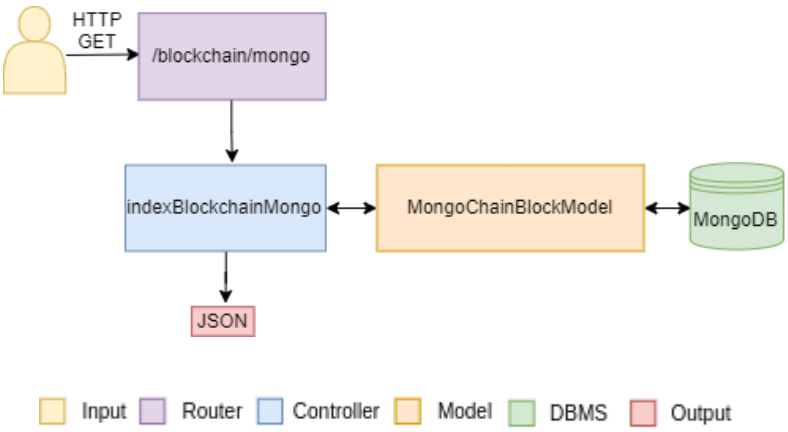


Fonte: Elaborado pelo Autor (2020)

4.2.2 Middleware indexBlockchainMongo

É responsável por retornar todos os documentos contidos na coleção *blocks* quando a rota */blockchain/mongo* é acessada via HTTP *GET* conforme mostra o fluxo da Figura 16.

Figura 16 - Obtém toda a *blockchain* persistida na coleção *blocks*



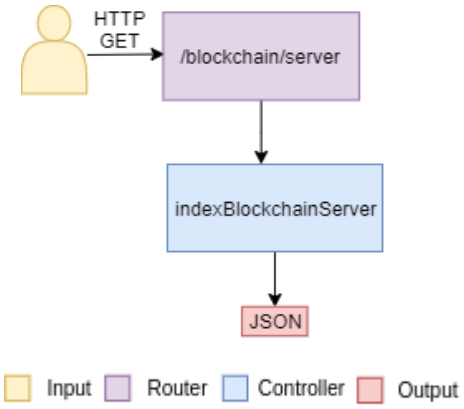
Fonte: Elaborado pelo Autor (2020)

Os clientes da aplicação podem consultar toda a *blockchain* armazenada no *MongoDB*. Mesmo com a ocorrência de transações concorrentes na coleção *blocks*, a consistência é mantida, pois os dados são gravados pelo *MongoChain* via transação ACID quando ocorre o processo de mineração.

4.2.3 Middleware indexBlockchainServer

A Figura 17 mostra o fluxo a ser executado pelos *nodes* para recuperar toda a cadeia de blocos criada nos servidores e que contém as transações geradas. Portanto, basta acessar a rota */blockchain/server* via HTTP *GET*.

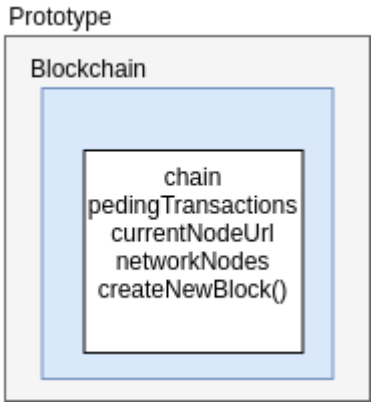
Figura 17 - Recuperação da *blockchain* que é criada dinamicamente no servidor



Fonte: Elaborado pelo Autor (2020)

Este *middleware* retorna um objeto com a *blockchain* atual que está sendo processada pelos *nodes* participantes da rede permissionada. O construtor da função *blockchain* é mostrado na Figura 18 sendo composto pela propriedade *currentNodeUrl*, arrays (*i.e., chain*, *pendingTransactions*, *networkNodes*) e a função *createNewBlock*. O array *chain* representa a *blockchain* e *pendingTransactions* são as transações pendentes que ficam armazenadas no servidor e já estão prontas para serem mineradas, ou seja, adicionadas à *blockchain* e persistidas na coleção *blocks* do *MongoDB*. O *node* representado pela propriedade *currentNodeUrl* é quem realiza a chamada para o *middleware indexBlockchainServer*. O conjunto de *nodes* que participa da rede é representado pelo array *netWorkNodes*. Ademais, a função *createNewBlock* faz a criação de um novo bloco ao receber como parâmetros os *hashs* do bloco anterior e atual, além do *nonce* que corresponde a quantidade de repetições feitas por um *node* minerador para encontrar um *hash* que se inicia com quatro zeros.

Figura 18 - Construtor da função *blockchain*

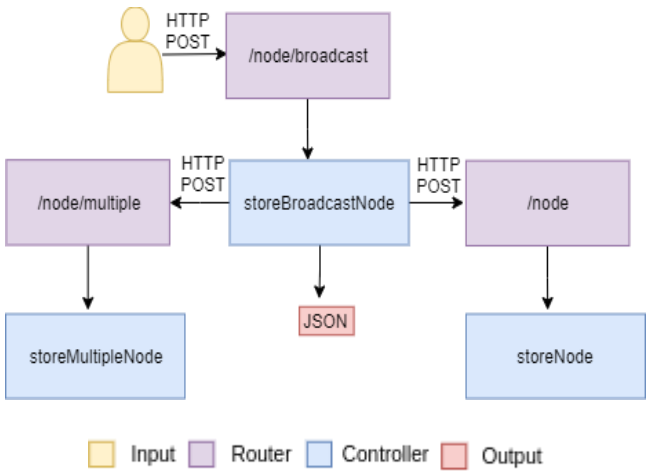


Fonte: Elaborado pelo Autor (2020)

4.2.4 Middleware storeBroadcastNode

A Figura 19 apresenta o fluxo de execução quando este *middleware* é acionado por meio da rota */node/broadcast* via HTTP *POST*. Portanto, ocorre uma comunicação com outros dois *middlewares* (*i.e., storeNode* e *storeNodeMultiple*). O *storeNode* utiliza o recurso localizado em */node* para possibilitar a adição na rede b*lockchain* de um novo *node* por vez pelos demais participantes. Por outro lado, o *storeNodeMultiple*, acessado em */node/multiple*, pode adicionar múltiplos *nodes* de uma única vez. Quando o *storeBroadcastNode* é chamado, ocorre um *broadcast* para a rede com metadados do *node* que foi adicionado.

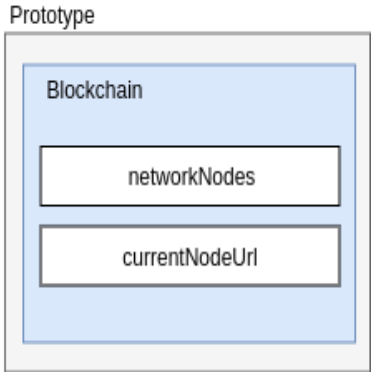
Figura 19 - Adição de um *node* na rede e realização de *broadcast*



Fonte: Elaborado pelo Autor (2020)

A Figura 20 mostra o *array networkNodes* que contém o conjunto de *nodes* e a propriedade *currentNodeUrl* que corresponde ao novo *node*.

Figura 20 - O *array networkNodes* e a propriedade *currentNodeUrl*



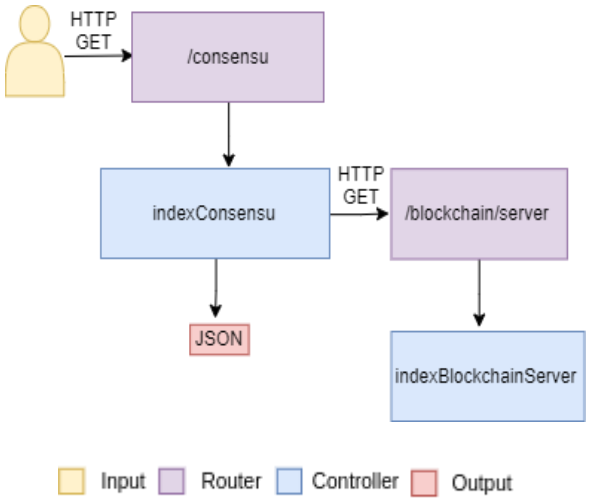
Fonte: Elaborado pelo Autor (2020)

4.2.5 Middleware indexConsensu

Por ser um ambiente distribuído, os *nodes* podem apresentar problemas, tais como, indisponibilidade, ocorrência de falhas ao adicionar blocos na *blockchain* e não receber o *broadcast* das transações que foram enviadas para a rede. Além disso, um usuário fraudulento pode fazer o envio de dados falsos pela rede, com o intuito de convencer todos os participantes que estão sendo realizadas transações legítimas. Sendo assim, algumas validações precisam ser realizadas e o *middleware indexConsensu* implementa um algoritmo para obter o consenso da rede que tem como regra o tamanho da *blockchain*. Isto significa que os dados são obtidos do *node* que deseja enviar as transações e realiza-se uma comparação entre o tamanho da sua *blockchain* com a que está sendo processada no momento pelos demais *nodes* da rede. A Figura 21 mostra o fluxo do *indexConsensu* que ao ser chamado via HTTP *GET* por meio da rota */consenso* comunica-se com o *middleware indexBlockchainServer* acessando a rota */blockchain/server* via HTTP *GET* que retorna a *blockchain* corrente no servidor para verificar se é maior do que a *blockchain* do *node* candidato. Em caso afirmativo, é necessário realizar uma substituição para que o *node* obtenha a *blockchain* do servidor. Porém, é preciso verificar a validade da estrutura de *blockchain* presente no *node* candidato, comparando os *hashs* de todos os blocos presentes. Então, a propriedade do *hash* corrente, denominada

previousBlockHash, é comparada com o *hash* do bloco anterior. Além disso, é necessário validar se cada bloco dentro da *blockchain* contém os dados corretos. Então, os blocos da cadeia são percorridos, sendo verificado se cada hash inicia com quatro zeros, pois assim serão blocos válidos. Ademais, algumas validações são realizadas no bloco gênesis, tais como: checar se o bloco está na posição zero da *blockchain*; certificar que o valor inicial da propriedade *nonce* foi definido pelo desenvolvedor; confirmar que a propriedade *previousBlockHash* e *hash* são iguais a zero; verificar a inexistência de transações.

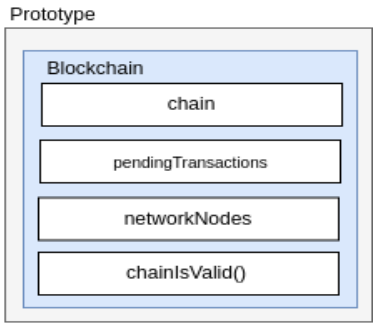
Figura 21 - Recebe um *node* para validação pelo protocolo de consenso



Fonte: Elaborado pelo Autor (2020)

A Figura 22 mostra os arrays (*i.e., chain*, *pendingTransactions* e *networkNodes*) e a função *chainIsValid*. O array netWorkNodes é percorrido para verificar se o *node* pertence à rede. Para isto, verifica-se o tamanho da sua *blockchain* ao comparar com a dos demais *nodes* por meio do array *chain*. A função *chainIsValid* é chamada para saber se existe uma *blockchain* válida ao comparar cada *hash* presente nos blocos, ou seja, o valor da propriedade *previousBlockHash* deve ser igual ao *hash* do bloco anterior. Por fim, o valor do array *pendingTransactions* do *node* candidato é atualizado com a adição de transações pendentes presentes nos demais *nodes* da rede.

Figura 22 - *Arrays* e a função *chainIsValid* do *middleware* de consenso

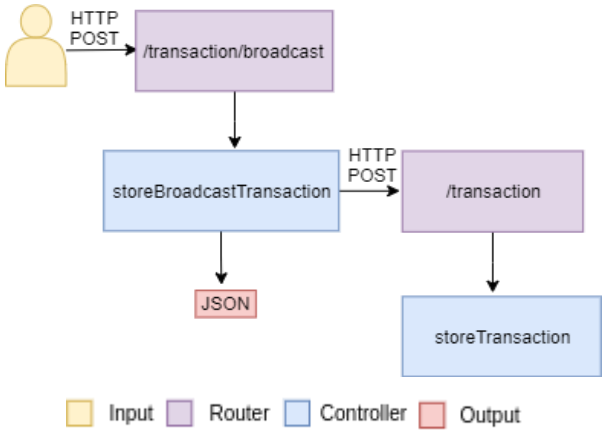


Fonte: Elaborado pelo Autor (2020)

4.2.6 Middleware storeBroadcastTransaction

Ao observar a Figura 23 nota-se que este *middleware* é acionado na rota */transaction/broadcast* via método HTTP *POST*, sendo responsável por receber uma transação e enviar para a rede por meio de um *node* que vai obter os dados e sincronizar com os demais *nodes*. Para isto, o *MongoChain* faz o armazenamento das transações no servidor e depois realiza um *broadcast* para toda a rede. Logo, comunica-se com o *middleware storeTransaction* que realiza transações quando é acionada a rota */transaction* via HTTP *POST*. Sendo assim, quando as transações são executadas permanecem na rede como pendentes, ou seja, ainda não passaram pelo processo de mineração para serem validadas e adicionadas a um bloco e depois persistidas na *blockchain*.

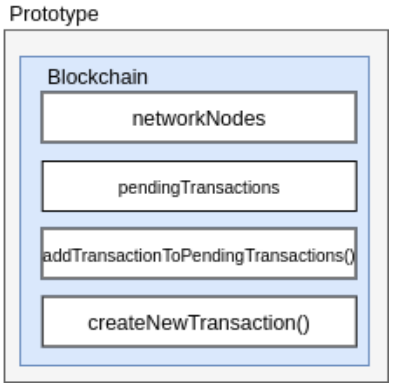
Figura 23 - Realização de *broadcast* das transações para serem mineradas



Fonte: Elaborado pelo Autor (2020)

Os *arrays* e funções envolvidos ao realizar o *broadcast* das transações são mostrados na Figura 24. Primeiramente é preciso criar uma transação e, para isto, é utilizada a função *createNewTransaction*, que no caso do *MongoChain* são passados os parâmetros *senderId*, *recipientId* e *amount*. Essas transações ficam em estado pendente e são alocadas no array pendingTransactions por meio da função *addTransactionToPendingTransactions*, pois esperam para serem mineradas e adicionadas a um bloco da *blockchain*. Portanto, para que toda a rede receba a lista de transações pendentes é preciso saber quais são os *nodes* participantes, sendo percorrido o *array networkNodes*.

Figura 24 - *Array* e funções do *middleware storeBroadcastTransaction*

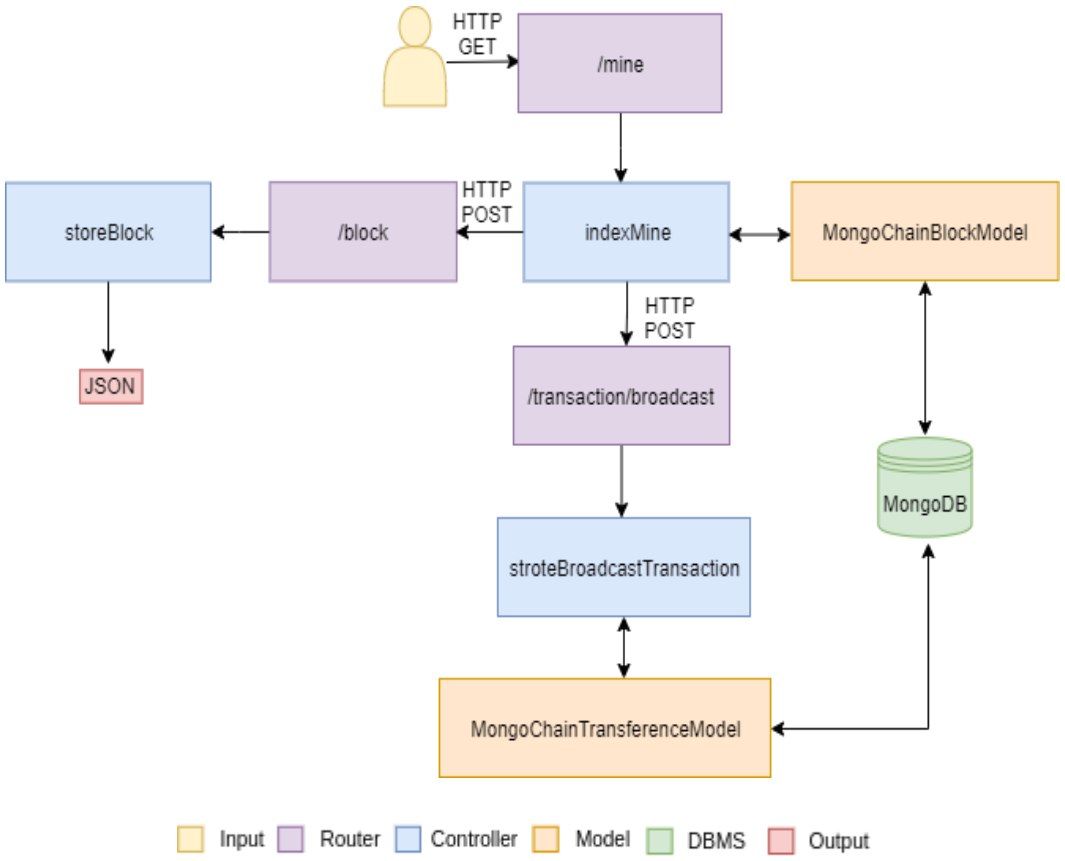


Fonte: Elaborado pelo Autor (2020)

4.2.7 Middleware indexMine

Este é o principal *middleware* do *framework MongoChain*, pois é nele que acontece a integração entre o SGBD *MongoDB* e a rede *blockchain* permissionada. A Figura 25 mostra que ao receber uma requisição de um *node* minerador por meio da rota */mine* via HTTP *GET* realiza-se uma comunicação com o *middleware* responsável por criar um novo bloco e armazenar as transações, o *storeBlock*, pela rota */block* via HTTP *POST*, tendo a resposta no formato JSON. Ao acessar a rota *transaction/broadcast* é chamando o *storeBroadcastTransaction* via HTTP *POST* para executar transações que serão adicionadas em blocos na rede *blockchain* e, ao mesmo tempo, terão seus dados persistidos em documentos pertencentes a diferentes coleções no SGBD *MongoDB* (*i.e., blocks* e *transferences*), mantendo as propriedades ACID.

Figura 25 - Mineração das transações e persistência na *blockchain* e *MongoDB*



Fonte: Elaborado pelo Autor (2020)

Quando o *MongoChain* é utilizado para outros domínios de aplicação, os desenvolvedores devem seguir um fluxo de execução de transação baseado em duas fases: i) registro e ii) confirmação. A primeira fase é ilustrada pelo Algoritmo 1 que mostra a possibilidade de um desenvolvedor customizar as transações que serão registradas no *MongoChain* e no *MongoDB*. Para isto, a função *createNewTransaction* é customizada ao receber como parâmetros as propriedades de remetente e destinatário (*i.e., senderProperties* e *recipientProperties*) (linha 4). Depois, o *middleware storeBroadcastTransaction* é modificado ao passar propriedades de remetente e destinatário como argumentos da função *createNewTransaction* (linha 5 e 6). Em seguida, o *middleware indexMine* é executado, sendo iniciada uma sessão do *MongoChain* que vincula uma transação que vai armazenar as propriedades de remetente e destinatário no *MongoChain* e *MongoDB* (linhas 7 a 9). Ademais, o *MongoChain* usa para leitura (*i.e., Read Concern*) o algoritmo SI para trabalhar com transações ACID e na escrita (*i.e, Write Concern*) o *Majority.* Logo, os dados lidos são aqueles confirmados pela maioria dos *nodes.* Porém, para provê mais disponibilidade

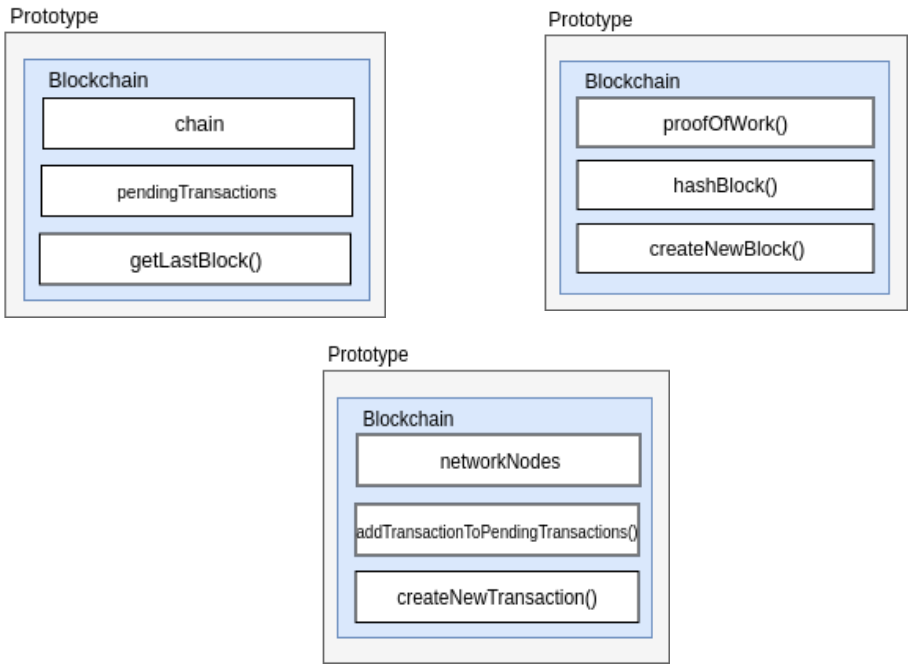
aos dados, o desenvolvedor pode mudar o valor do *Write Concern* para 1 (linhas 10 e 11). Com isto, o *indexMine* comunica-se com o *storeBroadcastTransaction* para obter transações que ao serem validadas são adicionadas a um bloco pelos mineradores e armazenadas na *blockchain* e *MongoDB* (*i.e.,* coleções *blocks* e *transferences*) (linhas 12 a 20). Se a transação apresentar erro, ocorre o cancelamento e a sessão do *MongoChain* é finalizada, senão a transação é efetivada e a sessão concluída, conforme descrito entre as linhas (21 a 26) do Algoritmo 1.

Algoritmo 1 - Registro de Transações

```
1:  Input: Broadcast Transactions
2:  Output: Void
3:  BEGIN
4:  SET Parameters of createNewTransaction as (senderProperties, recipientProperties)
5:  SET createNewTransaction in Middleware storeBroadcastTransaction
6:  SET Arguments of createNewTransaction as (senderProperties, recipientProperties)
7:      CALL indexMine
8:          START MongoChainSession
9:              START Transaction of MongoChainSession
10:                 GET Read Concern as Snapshot
11:                 GET Write Concern Majority OR Write Concern as 1
12:                 GET Middleware storeBroadcastTransaction
13:                     SET i to 1
14:                     FOR each transaction of Middleware storeBroadcastTransacion
15:                             STORE transaction in block
16                              STORE block.transaction in collection transferences
17:                         INCREMENT i
18:                         END FOR
19:                 STORE block in blockchain network
20:                 STORE block in collections blocks
21:                 IF error in Transaction THEN
22:                     ABORT Transaction
23:                     END MongoChainSession
24:                 END IF
25:             COMMIT Transaction
26:         END MongoChainSession
27: END
```

A Figura 26 apresenta os *arrays* e funções envolvidos para esse processo de mineração. A função *proofOfWork* obtém o bloco corrente e o *hash* do bloco anterior e tenta gerar um *hash* específico. Para o *MongoChain*, foi convencionado como sendo um *hash* que inicia com quatro zeros. Para isto, são realizadas adivinhação e checagem. Portanto, a função auxiliar *hashBlock* tem uma propriedade (*i.e., nonce*) a mais em relação a proofOfWork. Logo, a função *hashBlock* é executada muitas vezes e o *nonce* vai sendo gerado para cada *hash* de um novo bloco. Assim, a função *proofOfWork* retorna o valor correto do *nonce* e todos os participantes da rede sabem que se trata de um *hash* correto com transações validadas. O *hashBlock* trabalha não apenas em cima do bloco corrente, mas também no *hash* do bloco anterior e isso significa que toda a *blockchain* possui dados ligados e quem tentar fraudar deverá minerar novamente cada bloco.

Figura 26 - Os *arrays* e funções necessários para minerar as transações



Fonte: Elaborado pelo Autor (2020)

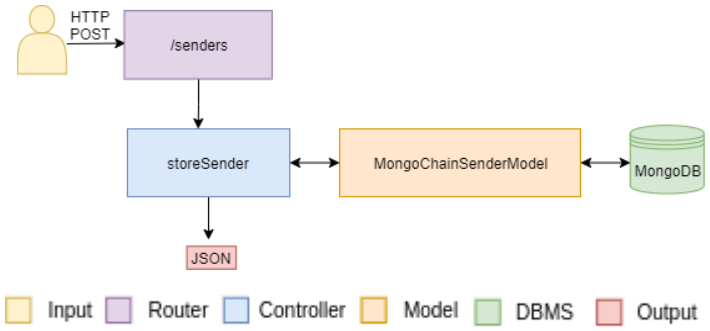
Quando ocorre a mineração, um novo bloco é criado. Para isto, a função *createNewBlock* é chamada e recebe como parâmetros o valor do *nonce* vindo da função *proofOfWork*, obtém o *hash* do bloco anterior por meio da função *getLastBlock* e gera automaticamente um *hash* via função *hashBlock*. É no *indexMine* que ao chamar a função *createNewTransaction* do *middleware storeBroadcastTransaction*, as

transações são adicionadas no *array pendingTransactions* por meio da função *addTransactionToPendingTransactions* e alocadas em um bloco para serem disseminadas pela rede depois do processo de mineração. Por fim, *todos* os blocos são armazenados no *array chain* e são disseminados a todos os *nodes* da rede ao percorrer o *array networkNodes*.

4.2.8 Middleware storeSender

Realiza o cadastro dos remetentes no *MongoDB* (coleção *senders*) ao acessar a rota */senders* via HTTP *POST* conforme mostra a Figura 27.

Figura 27 - Faz o cadastro do remetente

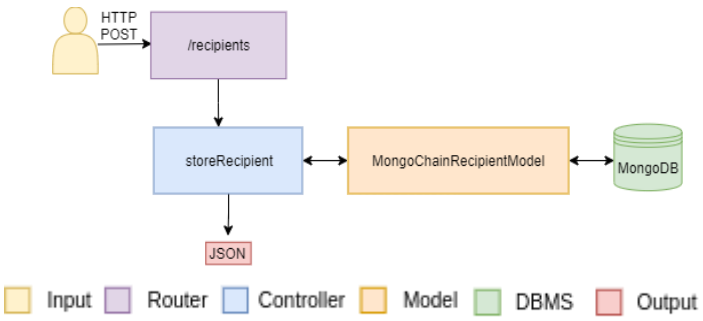


Fonte: Elaborado pelo Autor (2020)

4.2.9 Middleware storeRecipient

A Figura 28 mostra o fluxo para a inserção de dados referentes aos destinatários da transação no *MongoDB* (coleção *recipientes*) que são os recebedores de montantes. A rota para o recurso é a */recipientes* via HTTP *POST*.

Figura 28 - Realiza o cadastro do destinatário

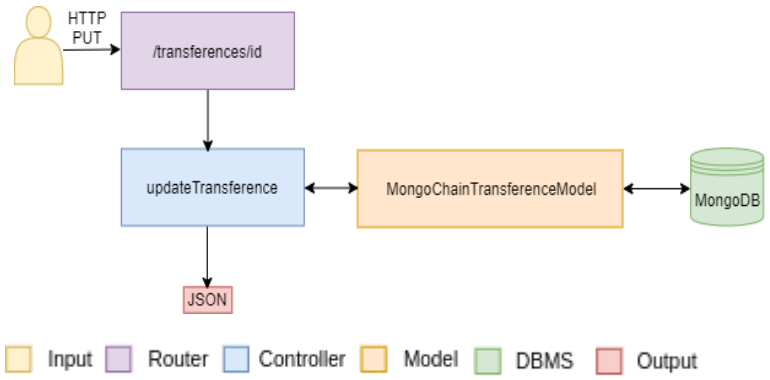


Fonte: Elaborado pelo Autor (2020)

4.2.10 Middleware updateTransference

Este *middleware* é responsável pela segunda fase do fluxo para execução de transação (*i.e.,* confirmação). Logo, quando acionado por meio da rota */transferences/id* via HTTP *PUT* realiza de fato a atualização de valores da coleção *transferences* no SGBD *MongoDB* conforme ilustrado na Figura 29.

Figura 29 - Confirma a transação alterando os valores da coleção *transferences*



Fonte: Elaborado pelo Autor (2020)

A fase de confirmação é ilustrada no Algoritmo e evidencia que o *middleware updateTransference* pode ser customizado ao receber valores identificadores da coleção *transferences* (*i.e., senderId*, *recipientId* e *transferencesId*) conforme descrito na linha 9. Em seguida, o *middleware updateTransference* é executado e pode modificar valores nas coleções *senders*, *recipients* e *transferences* (linha 10). Logo, em caso de erro, é levantada uma exceção, tendo a transação abortada e a sessão do *MongoChain* cancelada, senão a transação é executada e depois a sessão é encerrada, (linhas 11 a 16) do Algoritmo 2.

É imprescindível a obtenção do contexto transacional por meio das sessões, pois nesta transação ocorrem atualizações em documentos de várias coleções. Logo, na presença de requisições concorrentes de leituras por outros clientes da aplicação que buscam a lista atualizada de *senders*, *recipients* e *transferences*, se este *middleware* não estiver sido implementado por meio de uma sessão, pode-se obter valores inconsistentes. Quando o *middleware* é chamado, o valor do *amount* vai ser decrementado em *senders* e incrementado no *recipients*. Além disso, o *status* é modificado para "*done*" em *transferences*. Portanto, trata-se de uma transação ACID envolvendo múltiplos documentos em mais de uma coleção.

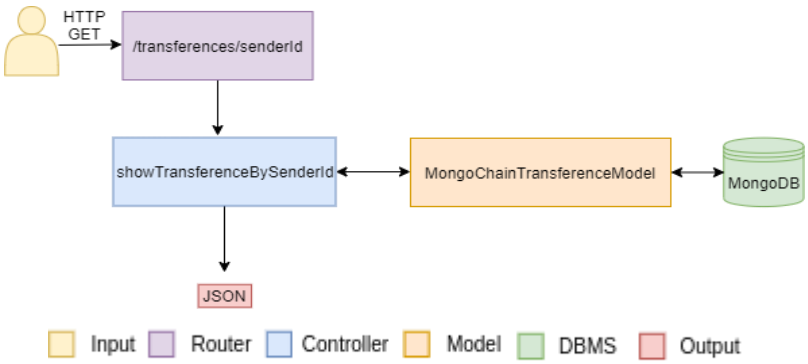
Algoritmo 2 - Confirmação de Transações

```
1:  Input: Entity identifier
2:  Output: Void
3:  BEGIN
4:      CALL Middleware updateTransference
5:          START MongoChainSession
6:              START Transaction of MongoChainSession
7:                  GET Read Concern as Snapshot
8:                  GET Write Concern as Majority OR Write Concern as 1
9:                  GET sendersId, recipientsId, transferencesId of collection transferences
10:                     UPDATE collections senders OR recipients OR transferences
11:                         IF error in Transaction THEN
12:                             END Transaction
13:                             ABORT MongoChainSession
14:                         END IF
15:             END Transaction
16:         END MongoChainSession
17: END
```

4.2.11 Middleware showTransferenceBySenderId

Este *middleware* tem o propósito de exibir todas as transações realizadas por remetente ao filtrar pelo id da coleção *senders* quando acessada a rota */transferences/senderId* via HTTP *GET* conforme ilustrado na Figura 30. Com o resultado pode-se realizar a confirmação, ou seja, debitar o *amount* do remetente e creditar no destinatário.

Figura 30 - Retorna as transações filtrando pelo *id* do remetente

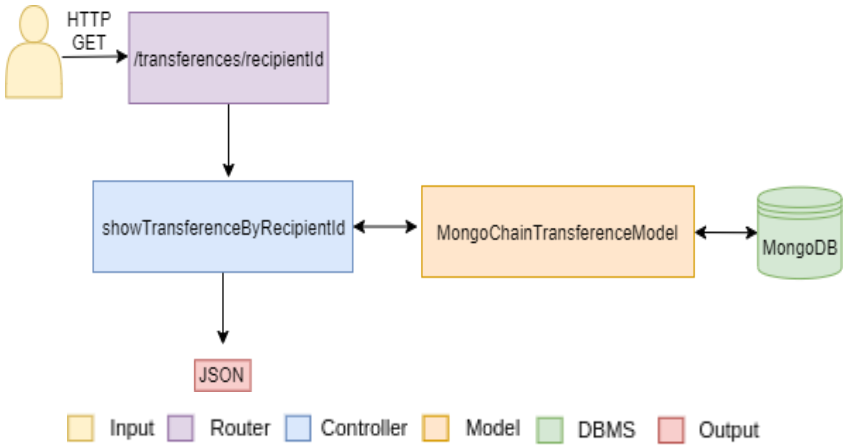


Fonte: Elaborado pelo Autor (2020)

4.2.12 Middleware showTransferenceByRecipientId

A Figura 31 mostra o fluxo para um destinatário visualizar se o remetente confirmou a transferência. Logo, ao buscar o recurso pelo *id* na rota */transferences/recipientId* via HTTP *GET* verifica-se o *amount* da coleção *recipients* atualizado.

Figura 31 - Obtém os dados das transações filtrando pelo *id* do destinatário



Fonte: Elaborado pelo Autor (2020)

4.3 ESPECIALIZAÇÕES

Para que possa ser mostrada a capacidade de extensão e reuso, o *MongoChain* foi especializado em dois *frameworks* para serem utilizados em domínios distintos, são eles: *MongoChainScheduleHealth* para registro de agendamentos em consultas médicas e *MongoChainMarketPlace* na criação de *marketplace* para produtos automotivos. A motivação do primeiro é a habitual necessidade dos desenvolvedores de realizarem implementações de recursos relacionados a agendamento nas aplicações tendo que lidar com a sincronização de escritas no BD e o gerenciamento de concorrência na leitura, sendo necessário tratar a questão da consistência em relação à disponibilidade. Além disso, é importante haver transparência nas operações relacionadas ao agendamento e registro das consultas realizadas. O segundo é proveniente da grande demanda por aplicações do tipo *e-commerce*, que envolvem transações financeiras, tendo que lidar com questões de segurança, provê dados com consistência forte e alta disponibilidade, além da

possibilidade de serem escaláveis para suprir o aumento na demanda de usuários. Os dois *frameworks* trazem recursos customizados, os quais servirão de base para as aplicações que serão implementadas e que irão atuar como mecanismo de validação para este trabalho. Para a criação dos *frameworks*, alguns *middlewares* foram modificados e outros adicionados. Foi visto que, no *MongoChain*, para que uma transação possa entrar na rede *blockchain* é necessário a interação entre as entidades *Sender* e *Recipient* e a geração de uma outra que faça a integração destas duas partes, a *Transference*. Portanto, para estender o *MongoChain* basta os desenvolvedores seguirem este modelo e fazerem as adaptações necessárias para contemplar os requisitos da aplicação. A seguir serão apresentados os *frameworks* com as suas funções especializadas.

4.3.1 MongoChainScheduleHealth

O *framework MongoChainScheduleHealth*, além das funcionalidades herdadas do *MongoChain* tem seus próprios esquemas presentes nos *Models*. Além disso, foram realizadas modificações nos *Controllers* e *Routes*. As entidades que representam o funcionamento básico para as aplicações que requerem agendamentos em clínicas médicas são definidas como sendo *Patient* e *Clinic* que representam respectivamente o remetente e destinatário. Ademais, existe a entidade *Appointment* que consiste no agendamento da consulta e foi adicionada a entidade *Change,* que receberá das clínicas as solicitações para alterações de datas em agendamentos para manter o histórico.

4.3.2 MongoChainMarketPlace

O *MongoChainMarketPlace* foi desenvolvido por meio do *MongoChain* e seus *Models*, *Controllers* e *Routes* são especializados em *Marketplace*. Diante deste contexto, existe a entidade *Customer,* que representa o remetente e, para o destinatário, utiliza-se a *Store.* Ademais, a entidade *Item* refere-se a um produto automotivo a ser comercializado e *Order* contempla o pedido solicitado. O Quadro 5 mostra o mapeamento das entidades do *MongoChain* com os *frameworks* especializados.

Quadro 5 - Mapeamento entre entidades do *MongoChain* e suas especializações

MongoChain	***MongoChain ScheduleHealth***	***MongoChain MarketPlace***
Sender	*Patient*	*Customer*
Recipient	*Clinic*	*Store*
Transference	*Appointment*	*Order*
	Change	*Item*

Fonte: Elaborado pelo Autor (2020)

4.4 CONSIDERAÇÕES FINAIS

Este capítulo descreveu o *MongoChain* mostrando sua arquitetura, especificação e especializações. Foram detalhados os *middlewares* que atuam como interceptadores de requisições e como eles se comunicam para realizar as ações necessárias para o bom funcionamento do *framework*. Além disso, foram implementados dois algoritmos que podem ser usados pelos desenvolvedores para execução de transações em outros domínios de aplicação. Ademais, foram desenvolvidos o *MongoChainScheduleHealth* e *MongoChainMarketPalce* que são extensões do *MongoChain* respectivamente para registro de agendamentos em consultas médicas e *marketplace* de produtos automotivos. O próximo capítulo mostra o ambiente de programação e as tecnologias utilizadas para construção do *MongoChain*.

5 PROTOTIPAÇÃO DO AMBIENTE DE PROGRAMAÇÃO

Este capítulo tem como objetivo apresentar o protótipo do ambiente de programação referente ao *MongoChain* observando aspectos de implementação e *interface*.

5.1 IMPLEMENTAÇÃO

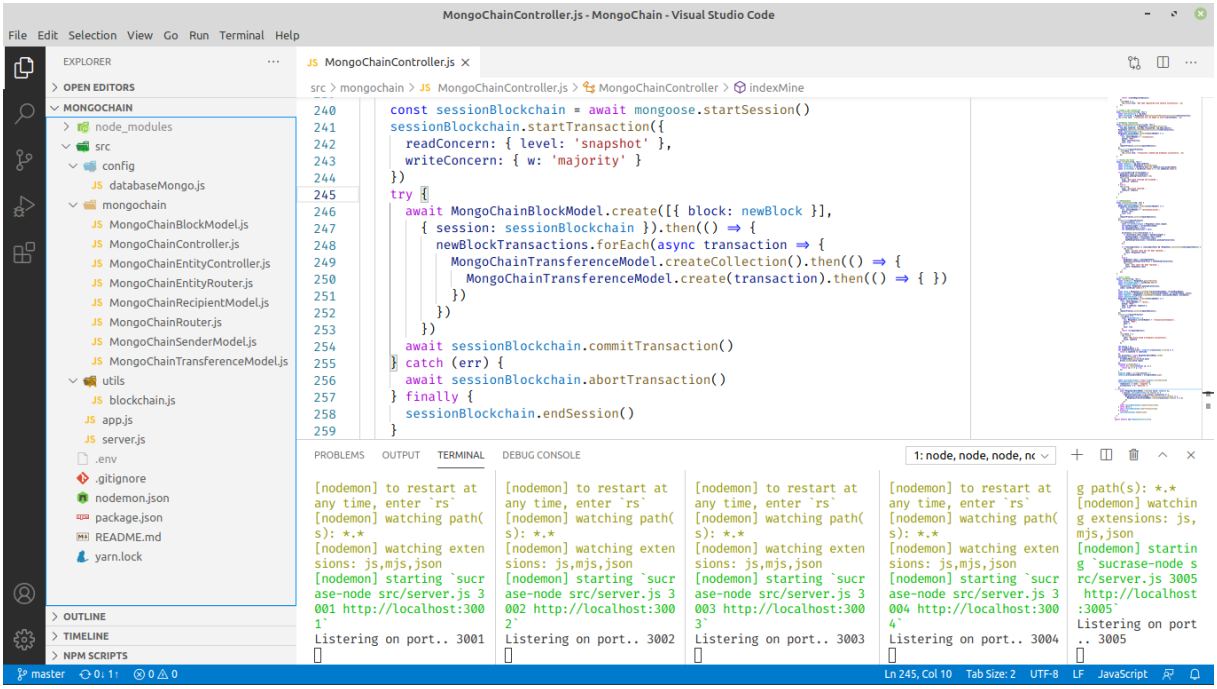
A linguagem de programação *Javascript* foi escolhida para implementação do *MongoChain*, pois tem a capacidade de auxiliar no desenvolvimento de sistemas com excelente performance de execução devido ao seu interpretador V8. Logo, as funcionalidades presentes no *back-end* são executadas com o *software NodeJS*. Além disso, o *MongoChain* é baseado no *framework* MVC *Express* por ser um sistema robusto para criação de API. As seguintes ferramentas foram utilizadas para implementar o *MongoChain* e suas extensões:

- *Javascript*: Linguagem de programação. Versão: *ECMAScript* 2018. Disponível em: http://www.ecmascript.org/;
- *NodeJS*: Ambiente de execução *Javascript*. Versão: 12.13.1. Disponível em: https://nodejs.org/;
- *MongoDB*: SGBD baseado em documentos. Versão: 4.2.2. Disponível em: https://www.mongodb.com;
- *Mongoose*: ODM *MongoDB* para *NodeJS*. Versão: 5.8.7. Disponível em: https://mongoosejs.com/;
- *MongoDB Atlas*: Serviço em nuvem do *MongoDB*. Disponível em: https://www.mongodb.com/cloud/atlas;
- *Postman*: Plataforma colaborativa para desenvolvimento de API. Versão: 7.13.0. Disponível em: https://www.getpostman.com/
- *MongoDB Compass*: *Interface* Gráfica do Utilizador (GUI) para *MongoDB*. Versão: 1.20.3. Disponível em: https://www.mongodb.com/products/compass;
- *Visual Studio Code*: Editor de código fonte. Versão: 1.4.1. Disponível em: https://code.visualstudio.com/;
- *Git*: Sistema de controle de versão distribuído. Versão 2.17.1. Disponível em https://git-scm.com/;

5.2 INTERFACE

A Figura 32 mostra o editor *Visual Studio Code* com a estrutura do *MongoChain*. No menu lateral esquerdo ficam os arquivos *Javascript*. O *databaseMongo.js* faz a configuração com o SGBD *MongoDB*, contendo os parâmetros necessários para estabelecer uma conexão, ou seja, configurando um Localizador Uniforme de Recursos (URL) para acesso ao BD que está localizado na nuvem em um *cluster* composto por três *nodes* com dados replicados. Na pasta *utils* tem o *blockchain.js* que contém todas as propriedades, *arrays* e funções que constituem a rede descentralizada e são utilizadas pelos *middlewares*. O *app.js* é um arcabouço de código, ou seja, habilita o acesso ao BD, o uso dos *middlewares* e a utilização das rotas que foram definidas. O *server.js* executa instâncias de servidores que vão atuar como *nodes* da rede *blockchain*. No centro fica a área de trabalho composta pela codificação de uma parte do *middleware indexMine* que realiza a integração entre funcionalidades transacionais provenientes do SGBD *MongoDB* e da *blockchain*. Na parte inferior está o terminal, que foi dividido para executar um sistema distribuído composto por servidores que se comunicam de forma local por diferentes portas.

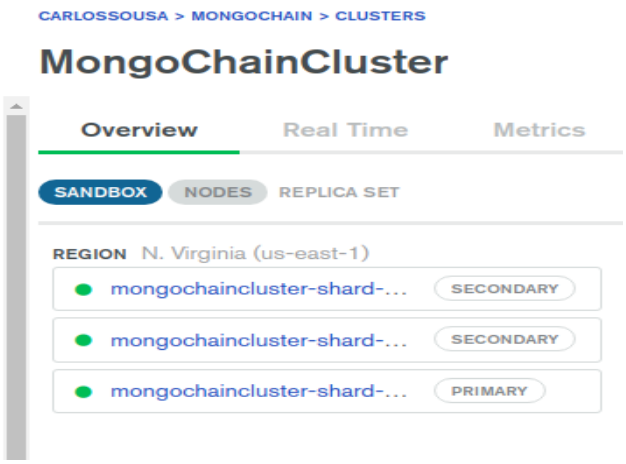
Figura 32 - *Interface* do *MongoChain*



Fonte: Elaborado pelo Autor (2020)

As requisições ao *MongoChain* são realizadas diretamente na sua API utilizando o *Postman* e o retorno é exibido no formato JSON. Logo, um *front-end* pode montar a parte visual da aplicação para exibir os dados no *desktop*, *browser* ou em um aplicativo *mobile*. A Figura 33 mostra o *cluster* pertencente ao *MongoChain*, sendo que os dados são replicados por meio do *Replica Set* com um *node* primário e dois secundários, localizados na região *N. Virginia*.

Figura 33 - O *cluster* do *MongoChain*



Fonte: Elaborado pelo Autor (2020)

5.3 CONSIDERAÇÕES FINAIS

Este capítulo discorreu sobre aspectos de implementação e *interface*. Assim, verificou-se que foram utilizadas tecnologias variadas para a construção do *framework* proposto. No próximo capítulo, um experimento será executado para avaliar o *MongoChain*. Além disso, para validar o uso do *MongoChain* em diferentes domínios, serão construídas duas aplicações.

6 AVALIAÇÃO DO MONGOCHAIN

Neste capítulo é apresentado um experimento para consolidar a proposta do *MongoChain*. Logo, a Seção 6.1 investiga os recursos do *MongoChain* para garantir a consistência dos dados e o funcionamento da rede *blockchain* ao realizar transações. Além disso, o uso do *MongoChain* na construção de aplicações para diferentes domínios será mostrado na seção 6.2.

6.1 EXPERIMENTO

Para avaliar o *framework* proposto, foi conduzido um experimento que investigou três recursos principais especificados no *MongoChain*, são eles: i) a garantia da consistência dos dados por meio das transações ACID; ii) a corretude do protocolo PoW na criação dos blocos da rede *blockchain*; iii) a segurança da rede *blockchain* para tratar transações fraudulentas.

Sendo assim, a fim de garantir as mesmas condições de igualdade para as questões avaliadas, foi construída uma rede local com as seguintes configurações: Sistema Operacional: *Linux Mint* 19.3 *Tricia*. *CPU*: *Intel(R) Core (TM) i7-8550U CPU @ 1.80GHz*. Memória *RAM*: 8*GB*. *HD*: *ATA Disk*, *WDC WD20SPZX-75U* com 1863*GiB* (2*TB*). Para a criação do ambiente de avaliação do *MongoChain*, as seguintes atividades foram realizadas:

I. Desenvolvimento de uma rede *blockchain* permissionada utilizando cinco servidores virtualizados, sendo dois deles mineradores.
II. Configuração de um *cluster* no *MongoDB Atlas* para persistir na nuvem os dados provenientes das transações.
III. Criação da coleção *blocks* com adição do bloco gênesis.
IV. Ajustes no *software Postman* para realizar requisições HTTP na API do *MongoChain*.

Terminada as atividades de construção e configuração do ambiente de avaliação realizaram-se as seguintes requisições HTTP:

- *POST* para adicionar uma instância do bloco gênesis e instâncias das entidades *Sender* e *Recipient* no *MongoDB*. Além disso, instâncias da entidade *Transference* serão validadas por dois mineradores e adicionadas à rede *blockchain* (*MongoChain*) e no *MongoDB*.

- *PUT* para modificar as instâncias *Sender*, *Recipient* e *Transference*.
- *GET* para visualizar os dados atualizados das instâncias.
- *DELETE* para remover a instância *Transference* e retornar os dados originais de *Sender* e *Recipient*.

Após o término das atividades que geraram as transações de escrita e leitura de dados no *MongoChain* e *MongoDB*, avaliou-se: a consistência das transações ACID, a eficiência da construção dos blocos na rede *blockchain* e a segurança da rede para tratar transações fraudulentas. Os principais resultados obtidos serão apresentados a seguir.

Os *nodes* que compõem a rede descentralizada do *MongoChain* compartilham os dados do bloco gênesis. Assim, para que a *blockchain* seja replicada no *MongoDB* deve ser criada a coleção *blocks* com os metadados do bloco gênesis da *blockchain*, como ilustra a Figura 34. Logo, para criar esse primeiro documento basta uma transação atômica, pois ainda não há transações concorrentes sendo executadas.

Figura 34 - Bloco gênesis adicionado no *MongoDB* pelo *MongoChain*

Bloco gênesis


```
{
    "chain": [
        {
            "index": 1,
            "timestamp": 1592336999036,
            "transactions": [],
            "nonce": 100,
            "hash": "0",
            "previousBlockHash": "0"
        }
    ],
    "pendingTransactions": [],
    "currentNodeUrl": "http://localhost:3001",
    "networkNodes": []
}
```


blocks


```
_id: ObjectId("5ee8e191b507b11f98675547")
block: Object
  index: 1
  timestamp: 1592320239342
  transactions: Array
  nonce: 100
  hash: "0"
  previousBlockHash: "0"
__v: 0
```


Fonte: Elaborado pelo Autor (2020)

A Figura 35 mostra duas instâncias das entidades *Sender* e *Recipient* cadastradas no *MongoDB* nas coleções *senders* e *recipientes* respectivamente.

Figura 35 - Instâncias de *Sender* e *Recipient* cadastradas pelo *MongoChain*

senders


```
_id: ObjectId("5ee8e1e4b507b11f98675548")
name: "João"
amount: 1000
createdAt: 2020-06-16T15:14:44.206+00:00
__v: 0
```


recipients


```
_id: ObjectId("5ee8e20cb507b11f98675549")
name: "Maria"
amount: 0
createdAt: 2020-06-16T15:15:24.987+00:00
__v: 0
```


Fonte: Elaborado pelo Autor (2020)

Para realizar a transferência entre valores presentes nos campos *amount* das instâncias *Sender* e *Recipient* é necessário fazer o registro da transação na *blockchain* e no *MongoDB*. Logo, quando a transação é executada, fica acessível para todos os *nodes* da rede e deve ser validada por um *node* minerador. Assim, conforme ilustra a Figura 36, a transação é adicionada no segundo bloco da rede *blockchain*.

Figura 36 - Transação validada e adicionada a um bloco da *blockchain*


```
{
    "note": "New block mined & broadcast successfully",
    "block": {
        "index": 2,
        "timestamp": 1592320661111,
        "transactions": [
            {
                "senderId": "5ee8e1e4b507b11f98675548",
                "recipientId": "5ee8e20cb507b11f98675549",
                "amount": 300,
                "transactionId": "71b12320afe411eaaa08232870fd8cc0"
            }
        ],
        "nonce": 5391,
        "hash": "0000684135c612d15c86b65c202d5f52bf1652e4cef2324c85f17c1ed4ac4beb",
        "previousBlockHash": "0"
    }
}
```


Fonte: Elaborado pelo Autor (2020)

A transação, além de ser adicionada à *blockchain*, também é persistida em múltiplas coleções do *MongoDB*, ou seja, em um documento da coleção *blocks* e de uma nova coleção denominada *transferences*, conforme ilustra a Figura 37. Quando a transação é validada pelo *node* minerador é garantida a consistência dos dados devido a manutenção das propriedades ACID.

Figura 37 - Transação ACID adicionada nas coleções *blocks* e *transferences*

blocks


```
_id: ObjectId("5ee8e295fa7e751fc4bff43e")
block: Object
  index: 2
  timestamp: 1592320661111
  transactions: Array
    0: Object
      senderId: "5ee8e1e4b507b11f98675548"
      recipientId: "5ee8e20cb507b11f98675549"
      amount: 300
      transactionId: "71b12320afe411eaaa08232870fd8cc0"
  nonce: 5391
  hash: "0000684135c612d15c86b65c202d5f52bf1652e4cef2324c85f17c1ed4ac4beb"
  previousBlockHash: "0"
__v: 0
```


transferences


```
_id: ObjectId("5ee8e295fa7e751fc4bff43f")
status: "requested"
senderId: ObjectId("5ee8e1e4b507b11f98675548")
recipientId: ObjectId("5ee8e20cb507b11f98675549")
amount: 300
createdAt: 2020-06-16T15:17:41.478+00:00
__v: 0
```


Fonte: Elaborado pelo Autor (2020)

Além da rede *blockchain* do *MongoChain*, os documentos que representam os blocos na coleção *blocks* formam uma cópia da *blockchain* persistida no *MongoDB*. Com isso, são fornecidos dados transparentes não apenas aos *nodes* da rede *blockchain* permissionada, pois os demais clientes da aplicação podem acessar metadados referentes aos blocos e as transações por meio do *MongoDB*. Ademais, a *blockchain* e sua cópia são imutáveis, ou seja, não podem ser alteradas. Portanto, são realizadas apenas operações de leitura. Porém, atualizações em itens de dados ou a remoção da transação são operações constantemente realizadas em aplicações. Logo, no *MongoChain*, a transação registrada deve ser confirmada para que de fato a transferência entre valores do campo *amount* de uma instância *Sender* para *Recipient* seja concretizada. Para isto, é realizada uma transação ACID que vai alterar múltiplos documentos simultaneamente e garantir a consistência dos dados ao debitar o valor do campo *amount* da coleção *senders*, creditar na *recipients* e alterar o valor do campo *status* em *transferences* de "*requested*" para "*done*", conforme ilustrado na Figura 38. A transação deve ser executada completamente, senão será desfeita.

Figura 38 - Múltiplos documentos alterados pela transação ACID de confirmação

senders

```
_id: ObjectId("5ee8e1e4b507b11f98675548")
name: "João"
amount: 700
createdAt: 2020-06-16T15:14:44.206+00:00
__v: 0
```

recipients

```
_id: ObjectId("5ee8e20cb507b11f98675549")
name: "Maria"
amount: 300
createdAt: 2020-06-16T15:15:24.987+00:00
__v: 0
```

transferences

```
_id: ObjectId("5ee8e295fa7e751fc4bff43f")
status: "done"
senderId: ObjectId("5ee8e1e4b507b11f98675548")
recipientId: ObjectId("5ee8e20cb507b11f98675549")
amount: 300
createdAt: 2020-06-16T15:17:41.478+00:00
__v: 0
```

Fonte: Elaborado pelo Autor (2020)

Caso seja preciso reverter a transação de confirmação é necessário executar uma outra transação ACID que vai realizar a remoção do documento cadastrado anteriormente na coleção *transferences.* Além disso, *senders* e *recipients* ficam com os valores que constavam antes da transação de confirmação para seus campos *amount*, conforme foi apresentado na Figura 35. Ao realizar mais um registro de transação entre instâncias de *Sender* e *Recipient* verifica-se que os blocos são construídos de forma lógica e eficiente na rede *blockchain* do *MongoChain*, pois além

do seu *hash*, cada bloco possui a informação do *hash* anterior como mostra a Figura 39. Sendo assim, se um usuário mal-intencionado quiser fraudar a rede ao inserir uma nova transação no bloco já minerado, o *hash* é alterado criando um efeito cascata pois todos os blocos estão conectados. Além disso, observa-se que o protocolo de consenso PoW utilizado no *MongoChain* está funcionando corretamente, pois sempre gera *hashs* com sequências iniciais de quatro números zero.

Figura 39 - Corretude do protocolo de consenso PoW do *MongoChain*


```
{
    "note": "New block mined & broadcast successfully",
    "block": {
        "index": 3,
        "timestamp": 1592321267770,
        "transactions": [
            {
                "transactionId": "85418790afe411eab6b323293d872ec5"
            },
            {
                "senderId": "5ee8e4a1b507b11f9867554a",
                "recipientId": "5ee8e4beb507b11f9867554b",
                "amount": 800,
                "transactionId": "e9fc0240afe511eaaa08232870fd8cc0"
            }
        ],
        "nonce": 65705,
        "hash": "0000f03e1214c76d6c05082314f6725da3612c13e6697523997e0ebdf2f42f7a",
        "previousBlockHash": "0000684135c612d15c86b65c202d5f52bf1652e4cef2324c85f17c1ed4ac4beb"
    }
}
```


Fonte: Elaborado pelo Autor (2020)

Durante a construção da *blockchain* podem ocorrer falhas nos *nodes* e, para que possam novamente entrar na rede, o *MongoChain* possui um algoritmo de consenso. Assim, verifica-se o tamanho da *blockchain* e os dados contidos em cada bloco. A Figura 40 mostra um usuário mal-intencionado tentando executar uma transação maliciosa no *MongoChain* ao entrar com o *node* na rede.

Figura 40 - Tentativa de executar uma transação maliciosa no *MongoChain*


```
{
    "chain": [
        {
            "index": 1,
            "timestamp": 1592320250214,
            "transactions": [],
            "nonce": 100,
            "hash": "0",
            "previousBlockHash": "0"
        }
    ],
    "pendingTransactions": [
        {
            "senderId": "5ee8e1e4b507b11f98675548",
            "recipientId": "5ee8e659b507b11f9867554c",
            "amount": 700,
            "transactionId": "e84a1710afe611eab3103985b2babd31"
        }
    ],
    "currentNodeUrl": "http://localhost:3005",
    "networkNodes": []
}
```


Fonte: Elaborado pelo Autor (2020)

Como o bloco gênesis tem uma transação pendente que não está presente nos blocos dos demais *nodes*, quando o usuário mal-intencionado faz uma solicitação para o *node* entrar novamente na *blockchain*, o *MongoChain* não vai permitir como mostrado na Figura 41, mantendo a rede segura.

Figura 41 - *Node* impedido de entrar na rede do *MongoChain*


```
{
    "note": "Current chain has not been replaced.",
    "chain": [
        {
            "index": 1,
            "timestamp": 1592320250214,
            "transactions": [],
            "nonce": 100,
            "hash": "0",
            "previousBlockHash": "0"
        }
    ]
}
```


Fonte: Elaborado pelo Autor (2020)

6.2 DESENVOLVIMENTO DE APLICAÇÕES

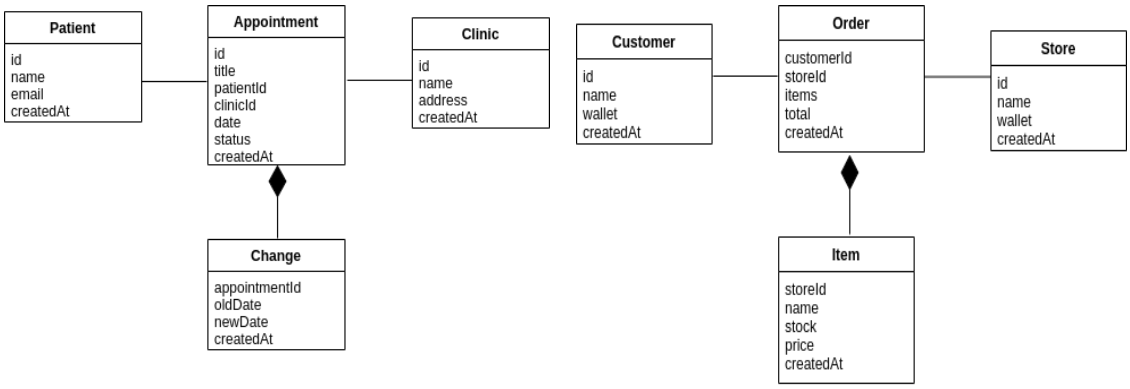
Para validar a capacidade do *MongoChain* de atender a diferentes domínios, foram desenvolvidas duas aplicações. A primeira aplicação foi motivada pela necessidade de manter os dados sempre disponíveis quando pacientes executam ações para agendar e reagendar consultas médicas. A segunda aplicação foi escolhida devido à demanda pelo processamento de vendas em *marketplace* de produtos automotivos, no qual os dados devem ter consistência forte. O desenvolvimento dessas aplicações visa ilustrar a aplicabilidade do *MongoChain* na criação de sistemas transacionais com diferentes requisitos de dados.

Com o intuito de facilitar o entendimento dos dois domínios de aplicação, o agendamento de consultas médicas (Figura 42a) e *marketplace* de produtos automotivos (Figura 42b) são ilustrados como entidades, relacionamentos e atributos. Assim, foi utilizado o ambiente computacional descrito na Seção 6.1 para executar os dois *frameworks* provenientes do *MongoChain* (*i.e., MongoChainScheduleHealth* e *MongoChainMarketPlace)* que são responsáveis por:

I. Mapear as propriedades das aplicações ilustradas na Figura 42.
II. Fazer comunicação com a API do *MongoChain* para realizar operações de escrita e leitura de dados.

Figura 42 - Entidades, relacionamentos e atributos para domínios de aplicações

(a) Agendamento de Consultas médicas (b) *Marketplace* de Produtos Automotivos



Fonte: Adaptado de (SADALAGE e FOWLER, 2013)

O Quadro 6 apresenta um conjunto de requisições que foi especificado para verificar o serviço de manipulação de dados e permitiu avaliar a manutenção das propriedades ACID e segurança das transações que realizam escrita e leitura de dados no *MongoChain*. Os resultados obtidos serão descritos a seguir.

Quadro 6 - Requisições de escrita e leitura de dados no *MongoChain*

Requisição	**URL**	***Middleware***	**Ação**
POST	*http://localhost:3001/patients*	*storePatient*	*Create*
POST	*http://localhost:3001/clinics*	*storeClinic*	*Create*
PUT	*http://localhost:3001/appointments/id*	*updateAppointment*	*Update*
PUT	*http://localhost:3001/changes/appointmentId*	*updateChange*	*Update*
POST	*http://localhost:3002/customers*	*storeCustomer*	*Create*
POST	*http://localhost:3002/stores*	*storeStore*	*Create*
POST	*http://localhost:3002/items*	*storeItem*	*Create*
PUT	*http://localhost:3002/orders/id*	*updateOrder*	*Update*
POST	*http://localhost:3003/transaction/broadcast*	*storeBroadcastTransaction*	*Create*
GET	*http://localhost:3004/mine*	*indexMine*	*Read*

Fonte: Elaborado pelo Autor (2020)

O processamento de requisições foi iniciado pela operação de inserção em instâncias de *Patient* e *Clinic* (*i.e., create*). Depois foi realizado um agendamento e o resultado pode ser visto na Figura 43. Assim, um bloco foi minerado (*i.e., read*) e adicionado à *blockchain* e no *MongoDB*. Com isso, a transação presente no bloco foi simultaneamente adicionada em uma coleção *appointments* mantendo as propriedades ACID para garantir a consistência dos dados.

Figura 43 - Adicionada instância de agendamento no *MongoChain* e *MongoDB*

appointments

blockchain do MongoChain

blocks

Fonte: Elaborado pelo Autor (2020)

Finalizada essa etapa, foi executada uma transação de confirmação, sendo que em *appointments* o campo *status* que antes estava com o valor "*requested*" foi alterado para "*scheduled*" (Figura 44).

Figura 44 - Agendamento confirmado com o valor do campo *status* alterado

appointments

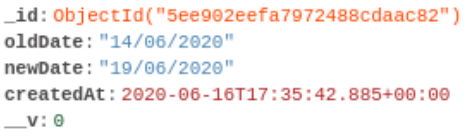
Fonte: Elaborado pelo Autor (2020)

A operação de confirmação de agendamento não afetou múltiplos documentos. Assim, não houve a necessidade de utilizar o contexto transacional. Além disso, conforme ilustrado na Figura 45, a clínica pode reagendar uma consulta sendo necessário uma transação ACID em múltiplos documentos alterando o valor do campo *status* em *appointments* para "*rescheduled*" e do campo "*date*" para uma nova data. Ademais, é criada uma instância para o registro da alteração realizada, sendo armazenada na coleção *changes*.

Figura 45 - Coleção *appointments* alterada e *changes* criada via transação ACID

appointments

changes



Fonte: Elaborado pelo Autor (2020)

No outro domínio de aplicação foram cadastradas instâncias de *Customer*, *Store* (Figura 46) e *Items* (Figura 47) no *MongoDB*.

Figura 46 - Adição de instâncias *Customer* e *Store* no *MongoDB*

customers

stores

Fonte: Elaborado pelo Autor (2020)

Figura 47 - Instâncias de *Items* inseridas no *MongoDB*

Items

Fonte: Elaborado pelo Autor (2020)

Com isso, uma solicitação de compra é realizada e adicionada na *blockchain*. Logo, como a transação é ACID, acontece a persistência dos dados em múltiplos documentos (*i.e.,* coleção *blocks* e *orders*, que acabou de ser criada), como mostra a Figura 48.

Figura 48 - Instância de *Order* criada e adicionada na *blockchain* e *MongoDB*

orders

blockchain do MongoChain

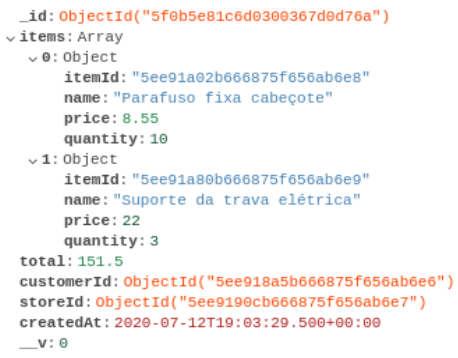
blocks

Fonte: Elaborado pelo Autor (2020)

O pedido está pronto para ser confirmado pelo cliente e, quando isto ocorrer, múltiplos documentos de várias coleções terão seus campos atualizados de maneira simultânea. Portanto, a Figura 49 mostra que ao executar a transação ACID, o valor do campo *total* na coleção *orders* será calculado. Além disso, será debitado o valor do campo *amount* de *customers* e creditado em *stores*. Ademais a quantidade do campo *stock* em *items* será decrementada.

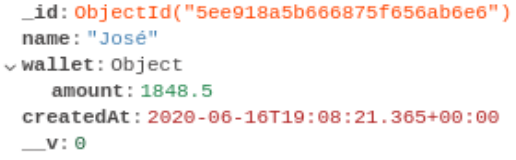
Figura 49 - Alterações em *orders*, *customers*, *stores* e *items* via transação ACID

orders



items

customers



stores

Fonte: Elaborado pelo Autor (2020)

6.3 CONSIDERAÇÕES FINAIS

Neste capítulo, foi realizado um experimento que mostrou a capacidade do *MongoChain* em garantir a persistência de dados consistentes utilizando transações ACID em múltiplos documentos. Além disso, verificou-se que as transações executadas pelo *MongoChain* são seguras e transparentes. Ademais, foi visto que o *MongoChain*, com o auxílio de *frameworks* especialistas como o *MongoChainScheduleHealth* e *MongoChainMarketPlace* consegue provê um ambiente programável para construir aplicações que podem ser utilizadas em diferentes domínios.

7 CONCLUSÃO

Este trabalho apresentou o *MongoChain*, um *framework* que auxilia no desenvolvimento de aplicações transacionais para diferentes domínios. Sua API é capaz de receber requisições de transações e devolver respostas para as aplicações clientes em uma rede distribuída e descentralizada. Com isto, as transações são persistidas mantendo as propriedades ACID em um modelo flexível que faz uso dos conceitos BASE ao utilizar o SGBD NoSQL de agregados *MongoDB.* Além disso, as transações são transmitidas para todos os *nodes* da rede e devem ser validadas por mineradores para serem armazenadas em uma *blockchain* permissionada que garante a segurança e transparência dos dados. Ademais, os dados são alocados na nuvem em *nodes* pertencentes a um *cluster* replicado.

7.1 PRINCIPAIS CONTRIBUIÇÕES

A principal contribuição deste trabalho é o *framework MongoChain* que se diferencia das soluções presentes no estado da arte por provê um ambiente programável que além de gerenciar transações, consegue auxiliar no desenvolvimento de aplicações. Isto acontece por meio da integração de recursos presentes nos SGBD relacionais, NoSQL e tecnologia *blockchain*. Outras contribuições originadas deste estudo foram as implementações de dois *frameworks* provenientes do *MongoChain*, o *MongoChainScheduleHealth* para desenvolver aplicações que realizam agendamentos em clínicas médicas e o *MongoChainMarketPlace* especializado em *marketplace* de produtos automotivos. Foi realizado um experimento no *MongoChain* para avaliar a garantia da consistência dos dados em operações de leitura e escrita no BD, a eficiência do protocolo PoW quando criados blocos na *blockchain* e a segurança da rede ao tratar transações fraudulentas. Além disso, duas aplicações para domínios distintos foram construídas com os *frameworks* estendidos para mostrar a capacidade de customização e aplicabilidade do *MongoChain*.

Os resultados mostram que o *MongoChain* auxilia os desenvolvedores na construção de sistemas transacionais executados em *clusters* e que atendem a diferentes domínios com dados consistentes, escaláveis, disponíveis, seguros e transparentes.

7.2 TRABALHOS FUTUROS

Para propostas de trabalhos futuros recomenda-se a realização de testes em que haja conexão com a *Internet* para acesso à rede *blockchain*, pois como se trata de uma prova de conceito e para tornar ágil a construção e execução das transações, utilizou-se um ambiente de desenvolvimento local. Além disso, novos protocolos de consenso podem ser alocados no *MongoChain* para serem avaliados quando executados no contexto transacional ACID. Ademais, pode ser investigado se o *MongoChain* consegue ser adaptado para uso com outros SGBD NoSQL de agregados.

REFERÊNCIAS

ANTONOPOULOS, Andreas M. **Mastering Bitcoin: Programming the open blockchain**. " O'Reilly Media, Inc.", 2017.

BAKER, Jason et al. Megastore: **Providing scalable, highly available storage for interactive services**. 2011.

BARBOSA, Álvaro CP. **Middleware para integração de dados heterogêneos baseado em composição de frameworks.** PhD theses, PUC-Rio, Brazil, 2001.

BERENSON, Hal et al. A critique of ANSI SQL isolation levels. In: **ACM SIGMOD Record**. ACM, 1995. p. 1-10.

BIGCHAINDB. BigchainDB 2.0. The Blockchain Database – **white paper**. May 2018. Disponível em: <https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf>. Acesso em: 11/05/2019.

BLUMMER, Tamas et al. The Blockchain Database – **white paper**. May 2018. Disponível em: <https://www.hyperledger.org/wp-content/uploads/2018/07/HL_Whitepaper_IntroductiontoHyperledger.pdf>. Acesso em: 26/05/2019.

BREWER, Eric. CAP Twelve years Later: how the. **Computer**, n. 2, p. 23-29, 2012.

BURBECK, Steve. **Applications programming in smalltalk-80 (tm): How to use model-view-controller (mvc)**. Smalltalk-80 v2, v. 5, p. 1-11, 1992.

CASCIARO, Mario. **Node.js Design Patterns**. Packt Publishing Ltd, 2014.

COELHO, Fábio André Castanheira Luís et al. pH1: A Transactional Middleware for NoSQL. In: **2014 IEEE 33rd International Symposium on Reliable Distributed Systems**. IEEE, 2014. p. 115-124.

COUCHDB: **Technical Overview**. Documentation 2019. Disponível em: <https://docs.couchdb.org/en/stable/intro/overview.html#acid-properties>. Acesso em: 04/05/2019.

COULOURIS, George F.; DOLLIMORE, Jean; KINDBERG, Tim; BLAIR, Gordon. **Distributed systems: concepts and design**. pearson education, 2012.

DECUYPER, X. **How does a blockchain works**? 2017. Disponível em: https://www.savjee.be/videos/simply-explained/how-doesablockchain-work/. Acesso em: 25/06/2019.

DEVELOPER Survey Results. **Stackoverflow**. 2019. Disponível em: <https://insights.stackoverflow.com/survey/2019>. Acesso em: 13/04/2019.

EINI, Oren. **Inside RavenDB 4.0**. 2018. Diponível em: https://ravendb.net/learn/inside-ravendb-book/reader/4.0/1-welcome-to-ravendb Acesso em: 04/05/2019.

ELMASRI, Ramez; NAVATHE, Shamkant B. **Sistemas de Banco de Dados**. 6a edição. São Paulo: Person, 2011.

ERL, Thomas; KHATTAK, Wajid; BUHLER, Paul. **Big data fundamentals: concepts, drivers & techniques**. Prentice Hall Press, 2016

EVANS, Eric. Domain-driven design: **tackling complexity in the heart of software**. Addison-Wesley Professional, 2004.

EXPRESS. **Middleware Pattern**. Documentation, 2019. Disponível em: http://expressjs.com/en/guide/using-middleware.html#middleware.application Acesso em: 11/07/2019.

FAYAD, M. E., SCHMIDT, D. C., AND JOHNSON, R. E. Application Frameworks. InM.Spencer (Ed.), **Building application frameworks: object-oriented foundations of framework design.** John Wiley and Sons, New York, USA, pp. 3–29, 1999.

FERRO, Daniel Gómez et al. Omid: Lock-free transactional support for distributed data stores. In: **2014 IEEE 30th International Conference on Data Engineering**. IEEE, 2014. p. 676-687.

GAUR, Nitin et al. **Hands-On Blockchain with Hyperledger: Building decentralized applications with Hyperledger Fabric and Composer.** Packt Publishing Ltd, 2018.

GIAMAS, Alex. **Mastering MongoDB 4.x: expert techniques to run high-volume and fault-tolerant database solutions using MongoDB 4. x.** 2019.

KARYDIS, Ioannis et al. (Ed.). **Algorithmic Aspects of Cloud Computing:** First International Workshop, ALGOCLOUD 2015, Patras, Greece, September 14-15, 2015. Revised Selected Papers. Springer, 2016.

LAURENCE, Tiana. **Blockchain for dummies**. For Dummies, 2019.

LOTFY, Ayman E. et al. A middle layer solution to support ACID properties for NoSQL databases. **Journal of King Saud University-Computer and Information Sciences**, v. 28, n. 1, p. 133-145, 2016.

MCCONAGHY, Trent et al. BigchainDB: a scalable blockchain database. **white paper**, BigChainDB, 2016.

MCCREARY. D; KELLY. A. **Making Sense of NoSQL: A GUIDE FOR MANAGERS AND THE REST OF US.** Manning Publication, 2014.

MONGODB. **Transactions**. Documentation, 2019. Disponível em: <https://docs.mongodb.com/manual/core/transactions/>. Acesso em: 04/05/2019.

OSMANI, Addy. **Learning JavaScript Design Patterns: A JavaScript and jQuery Developer's Guide**. " O'Reilly Media, Inc.", 2012.

PENDYALA, Vishnu. **Veracity of Big Data: Machine Learning and other approaches to verifying truthfulness**. Apress, 2018.

PENG, Daniel; DABEK, Frank. **Large-scale incremental processing using distributed transactions and notifications.** 2010.

POKORNY, J. NoSQL databases. A step to database scalability in web environment.

In Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services (iiWas '11). ACM, New York, NY, USA, 278 – 283, 2011.

PREE, Wolfgang; GAMMA, Erich. **Design patterns for object-oriented software development.** Reading, MA: Addison-wesley, 1995.

RAVAL, Siraj. **Decentralized applications: harnessing Bitcoin's blockchain technology**. " O'Reilly Media, Inc.", 2016.

RAVENDB. **Storage Engine Voron**. Documentation, 2019. Disponível em: <https://ravendb.net/docs/article-page/4.2/csharp/server/storage/storage-engine>. Acesso em: 04/05/2019.

REVILAK, Stephen; O'NEIL, Patrick; O'NEIL, Elizabeth. Precisely serializable snapshot isolation (PSSI). In: **2011 IEEE 27th International Conference on Data Engineering.** IEEE, 2011. p. 482-493.

ROBINSON, Ian; WEBBER, Jim; EIFREM, Emil. **Graph databases: new opportunities for connected data**. " O'Reilly Media, Inc.", 2015.

SADALAGE, Pramod J.; FOWLER, Martin. **NoSQL distilled: a brief guide to the emerging world of polyglot persistence.** Pearson Education, 2013.

SILBERSCHATZ, Abraham; KORTH, Henry F.; SUNDARSHAN, S.; **Sistema de banco de dados.** Elsevier Brasil, 2006.

SINGHAL, Bikramaditya; DHAMEJA, Gautam; PANDA, Priyansu Sekhar. **Beginning Blockchain: A Beginner's Guide to Building Blockchain Solutions**. Apress, 2018.

STEFANOV, Stoyan. **JavaScript Patterns: Build Better Applications with Coding and Design Patterns**. " O'Reilly Media, Inc.", 2010.

TAI, Stefan; EBERHARDT, Jacob; KLEMS, Markus. Not ACID, not BASE, but SALT. In: **Proceedings of the 7th International Conference on Cloud Computing and Services Science.** SCITEPRESS-Science and Technology Publications, Lda, 2017. p. 755-764.

TANENBAUM, Andrew S.; VAN STEEN, Maarten. **Distributed systems: principles and paradigms**. Prentice-Hall, 2007.

TIWARI, Shashank. **Professional NoSQL**. John Wiley & Sons, 2011.

VAISH, Gaurav. **Getting started with NoSQL**. Packt Publishing Ltd, 2013.

WEI, Zhou; PIERRE, Guillaume; CHI, Chi-Hung. CloudTPS: Scalable transactions for Web applications in the cloud. **IEEE Transactions on Services Computing**, v. 5, n. 4, p. 525-539, 2011.