ROGER HENRIQUE CARRIJO DE PAULA

INTEGRAÇÃO DA PNRD COM BANCO DE DADOS DISTRIBUÍDO HYPERLEDGER SAWTOOTH

Uberlândia, Minas Gerais

2021

ROGER HENRIQUE CARRIJO DE PAULA

INTEGRAÇÃO DA PNRD COM BANCO DE DADOS DISTRIBUÍDO HYPERLEDGER SAWTOOTH

Monografia de Conclusão de Curso apresentada no curso de graduação em Engenharia Mecatrônica da Universidade Federal de Uberlândia, como parte dos requisitos para obtenção de título de **BACHAREL EM ENGENHARIA MECATRÔNICA**.Área de concentração: Engenharia Mecatrônica

Universidade Federal de Uberlândia

Faculdade de Engenharia Mecânica

Orientador: José Jean-Paul Zanlucchi de Souza Tavares

Uberlândia, Minas Gerais

2021

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Curso de Graduação em Engenharia Mecatrônica
Rodovia BR 050, KM 78, Bloco 1D, 2º andar - Bairro Glória, Uberlândia-MG, CEP 38400-902
Telefone: (34) 2512-6771 - www.mecanica.ufu.br - mecatronica@mecanica.ufu.br

ATA DE DEFESA - GRADUAÇÃO

Curso de Graduação em:	Engenharia Mecatrônica				
Defesa de:	FEMEC42100 - Projeto de Fim de Curso II				
Data:	05/11/2021	Hora de início:	09:34	Hora de encerramento:	10:45
Matrícula do Discente:	11321EMT025				
Nome do Discente:	Roger Henrique Carrijo de Paula				
Título do Trabalho:	Integração da PNRD com banco de dados distribuído Hyperledger Sawtooth				

Reuniu-se de forma remota, através da Plataforma Online Microsoft Teams (https://teams.microsoft.com/l/channel/19%3aMRV9fnjC2SzSuUKW2GOa2ZRxibXVhjXGpGCpydGE6po1%40thread.tacv2/Geral?groupId=c104266c-6e8d-4e42-9554-83a2800821e8&tenantId=cd5e6d23-cb99-4189-88ab-1a9021a0c451), a Banca Examinadora, designada pelo docente orientador, assim composta: Prof. M.Sc. Rodrigo Hiroshi Murofushi - IFMG/Sabará-MG; Prof. Dr. Roberto Mendes Finzi Neto - FEMEC/UFU; e Prof. Dr. José Jean-Paul Zanlucchi de Souza Tavares - FEMEC/UFU orientador do candidato.

Iniciando os trabalhos, o presidente da mesa, Prof. Dr. José Jean-Paul Zanlucchi de Souza Tavares, apresentou a Comissão Examinadora e o candidato, e concedeu ao discente a palavra, para a exposição do seu trabalho. A duração da apresentação do discente e o tempo de arguição e resposta foram conforme as normas do curso.

A seguir o(a) senhor(a) presidente concedeu a palavra, pela ordem sucessivamente, aos(às) examinadores(as), que passaram a arguir o(a) candidato(a). Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o(a) candidato(a):

Aprovado sem nota.

As demandas complementares observadas pelos examinadores deverão ser satisfeitas no prazo máximo de 30 dias corridos a contar da data da defesa, para dar validade a esta aprovação.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **José Jean Paul Zanlucchi de Souza Tavares**, **Professor(a) do Magistério Superior**, em 05/11/2021, às 10:50, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Roberto Mendes Finzi Neto**, **Professor(a) do Magistério Superior**, em 05/11/2021, às 10:50, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Rodrigo Hiroshi Murofushi**, **Usuário Externo**, em 05/11/2021, às 10:52, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **3132708** e o código CRC **9195C5C7**.

Referência: Processo nº 23117.071504/2021-35 SEI nº 3132708

PAULA, Roger Henrique Carrijo de.**Integração da PNRD com banco de dados distribuído Hyperledger Sawtooth**. 2021. 120 f. Trabalho de Conclusão de Curso de Graduação em Engenharia Mecatrônica – Universidade Federal de Uberlândia, Uberlândia, 2021.

RESUMO

A tecnologia RFID é componente essencial no gerenciamento da cadeia de suprimento, pois proporciona rastreabilidade e identificação de produtos, mas a centralização de dados por parte das organizações participantes tornam o compartilhamento de informações deficitário, provocando divergências entre organizações, aumentando o tempo de entrega e custos do processo. Nesse contexto, o objetivo desse trabalho foi desenvolver um ambiente comum entre as organizações, atráves de um banco de dados distribuído, redes de petri e etiquetas RFID. Para isso, foi necessário modelar os processos através de redes de petri inseridas em tags RFID (PNRD) e provisionar um banco de dados distribuído. O resultado foi a criação da família de transações `PNRD_NET` e a sua utilização em aplicações descentralizadas.

Palavras-chave: Redes de Petri. Blockchain. Banco de dados distribuído. Hyperledger Sawtooth.

PAULA, Roger Henrique Carrijo de.**Integration of PNRD with Hyperledger Sawtooth Distributed Database**. 2021. 120 f.Monograph of the Mechatronics Engineering Course Completion, Federal University of Uberlândia, Uberlândia, 2021.

ABSTRACT

The RFID technology is an essential component in supply chain management, as it provides traceability and product identification, but the centralization of data by participating organizations made the sharing of information deficient, causing divergences between organizations, increasing delivery time and process costs. In this context, the objective of this work was to develop a common environment between organizations, through a distributed database, petri nets and RFID tags. For this, it was necessary to model the processes through petri nets inserted in RFID tags (PNRD) and provide a distributed database. The result was the creation of the `PNRD_NET` transaction family and its use in decentralized applications.

Keywords: Petri Net. Distributed database. Blockchain. Hyperledger Sawtooth.

LISTA DE ILUSTRAÇÕES

Figura 1 – Esquema de funcionamento tag RFID 15
Figura 2 – Esquema de construção de uma Tag RFID 16
Figura 3 – Representação interna de um EPC 17
Figura 4 – Estrutura do protocolo NFC (IGOE; COLEMAN; JEPSON, 2014) . . 20
Figura 5 – Representação de um semáforo em redes de petri no estado verde 21
Figura 6 – Disparo da transição T0 movimentando o token de P0 para P1 . . . 22
Figura 7 – Esquema de teste de peça usinada 24
Figura 8 – Comparativo de diferentes redes, em (A) uma rede centralizada, em (B) uma rede descentraliza e em (C) uma rede distribuída 28
Figura 9 – Representação genérica de um *Blockchain* (LANTZ; CAWREY, 2020) 28
Figura 10 – Representação genérica de um *Blockchain* 29
Figura 11 – Representação genérica de um bloco 30
Figura 12 – Arquitetura do Blockchain Hyperledger Sawtooth 32
Figura 13 – Arquitetura do Blockchain Hyperledger Sawtooth 33
Figura 14 – Estrutura básica de um lote (batch) em protobuf 33
Figura 15 – Esquema de funcionamento dos sistema criados 39
Figura 16 – Programa de interface com o programa Cliente 45
Figura 17 – Programa de interface (PALMS) 46
Figura 18 – Diagrama representando o funcionamento do Programa Cliente . . 47
Figura 19 – Rota para a criação de um proprietário 47
Figura 20 – Criação do proprietário Nikola Tesla 49
Figura 21 – Criação da gravação (Record) 50
Figura 22 – Representação em redes de petri de um esquema de teste de peça usinada 56
Figura 23 – Representação em redes de petri de uma máquina de teste de usinagem no Programa de Interface 57
Figura 24 – Pasta de arquivos gerados para a configuração do teste funcional . 59
Figura 25 – Arduino com placa NFC para leitura e tags 59
Figura 26 – Programa de interface na aba runtime 60
Figura 27 – Programa de interface na aba runtime - Conectado 60
Figura 28 – Programa de interface na aba runtime - Disparo de t0 61
Figura 29 – Programa de interface na aba runtime - Disparo de t1 61
Figura 30 – Programa de interface na aba runtime - Disparo de t0 62

LISTA DE TABELAS

Tabela 1 – Classes de Tags 16
Tabela 2 – Tipos de tags x frequência 16
Tabela 3 – Classes de Tags 17

LISTA DE ABREVIATURAS E SIGLAS

RFID	Radio-Frequency IDentification
PN	Petri Network
PNRD	Petri Net Inside RFID Database
SSOT	Single Source Of Truth
SOR	System Of Record
NDEF	NFC Data Exchange Format
EPC	Electronic Product Code
iPNRD	inverse Petri Net Inside RFID Database
RG	Reachability Graph
LTS	Labeled Transition System
P2P	Peer-to-Peer
PoW	Proof of Work

SUMÁRIO

Lista de códigos 11

1 INTRODUÇÃO 12
1.1 Objetivos 13
1.2 Justificativa 13

2 FUNDAMENTAÇÃO TEÓRICA 15
2.1 TECNOLOGIA RFID 15
2.1.1 ETIQUETAS RFID 15
2.1.1.1 Idetificador Único 16
2.1.2 ANTENAS DOS LEITORES 18
2.1.2.1 Polarização 18
2.1.2.2 Tipos de Antenas 18
2.1.2.3 Posição 18
2.1.2.4 Leitor 18
2.1.3 NFC 19
2.1.3.1 NDEF 19
2.2 REDES DE PETRI 21
2.2.1 DISPARO 22
2.2.2 DEFINIÇÃO FORMAL 22
2.2.2.1 Redes de Petri Lugares/Transições 22
2.2.2.2 Equação de Estado e Matriz de incidência 23
2.2.3 REPRESENTAÇÃO GRÁFICA 24
2.3 PNRD 25
2.3.1 EXCEÇÕES 26
2.4 SISTEMA DISTRIBUÍDO 27
2.5 BLOCKCHAIN 27
2.5.1 CADEIA DE BLOCOS 29
2.5.1.1 Blocos 29
2.5.1.2 Transação 31
2.5.1.3 Endereço 31
2.6 HYPERLEDGER SAWTOOTH 32
2.6.1 FUNCIONAMENTO 32
2.6.1.1 Merkle-Radix 32
2.6.1.2 Transações 33

2.6.1.3 Consenso 34
2.6.1.4 Camada de Aplicação 34
2.6.2 MÓDULOS PRINCIPAIS 34
2.6.2.1 Validador 34
2.6.2.2 Processador de Transação 35
2.6.2.3 REST API 35
2.6.2.4 Cliente 35

3 METODOLOGIA E DESENVOLVIMENTO 36
3.1 REQUISITOS DE PROJETO 36
3.1.1 COMPARAÇÃO DE DESEMPENHO ENTRE SOLUÇÕES DE BANCOS DE DADOS 37
3.1.1.1 MongoDB 37
3.1.1.2 Hyperledger Sawtooth 38
3.2 PROJETO DO SOFTWARE 38
3.3 GRUPO DE INFRAESTRUTURA 39
3.4 GRUPO APLICAÇÕES 41
3.4.1 FAMÍLIA DE TRANSAÇÕES PNRD NET 41
3.4.2 PROGRAMA DE INTERFACE 44
3.4.3 PROGRAMA CLIENTE 45
3.4.3.1 Testes 49
3.4.4 PROGRAMA PROCESSADOR DE TRANSAÇÕES 54
3.4.4.1 Classes para o Processamento de Transações 55
3.4.4.2 Método Apply 55
3.4.5 TESTE FUNCIONAL 55
3.4.5.1 Modelagem da Rede de Petri 56
3.5 RESULTADOS 64

4 CONCLUSÃO 66
4.1 Trabalhos Futuros 66

REFERÊNCIAS 68

APÊNDICES 69

APÊNDICE A – INFRAESTRUTURA SAWTOOTH 70

APÊNDICE B – FAMÍLIA DE TRANSAÇÕES 76

APÊNDICE C – PROCESSADOR DE TRANSAÇÕES **80**

APÊNDICE D – PROGRAMA CLIENTE **91**

LISTA DE CÓDIGOS

2.1 Exemplo de um bloco gerado da rede PNRD 30
2.2 Exemplo de uma transação gerada pela rede PNRD 31
3.1 Entidade Owner da Família PNRD NET 41
3.2 Entidade Record da Família PNRD NET 42
3.3 Payload da Família PNRD NET 43
3.4 Endpoints de ação da família PNRD NET 46
3.5 Endpoints complementares da Família PNRD NET 48
3.6 Consulta de Lote ao criar o proprietário Nikola 49
3.7 Body para criar um novo registro 50
3.8 Body para criar um novo registro 50
3.9 Body para a busca de uma tag específica 52
3.10 Busca de uma tag específica 52
3.11 Arquivo de configuração *.palms para teste funcional 57
3.12 Arquivo secret.txt 59
3.13 Arquivo rest api url.txt 59
3.14 Body para a busca de uma tag 80fa2243 61
3.15 Resultado da busca pela tag 80fa2243 62
3.16 Resultado da busca pela tag 80fa2243 no *blockchain* 63

1 INTRODUÇÃO

RFID ou identificação por radiofrequência é uma tecnologia que utiliza ondas de rádio para identificar e rastrear objetos. O sistema consiste em um leitor, uma antena e uma etiqueta (tag). O leitor é responsável por criar um campo magnético (radiofrequência), receber os dados e decodificar o sinal percebido, a tag é a portadora dos dados e geralmente é anexada ao objeto desejado, já a antena serve para recepção e transmissão das informações em ambos os sentidos. A tecnologia RFID possui diversas aplicações, como monitorar cargas em tempo real, identificar se um produto foi desviado ou segue em trajeto definido, amazenar dados como origem, quantidades, características, etapas de produção e controle de acesso.

A rede de Petri (PN), por sua vez, é uma técnica de modelagem que permite a representação de sistemas, especialmente sistemas concorrentes, utilizando como alicerce uma forte base matemática (SILVA, 2017). Utilizada como uma ferramenta de análise e controle de sistemas discretos, ela é capaz de modelar casos de concorrência, paralelismo, assincronismo, conflitos e deadlocks (SILVA, 2017). Tal versatilidade faz com que as redes de Petri sejam aplicáveis na descrição e visualização de sistemas com fluxo de trabalho complexo, que é o caso de vários processos de produção da atualidade.(SILVA, 2017)

Unindo os conceitos de RFID e Redes de Petri foi criada uma estrutura denominada de *Redes de Petri Elementares inseridas em base de dados RFID* ou PNRD (SILVA, 2017). Com essa estrutura é possível que a tag RFID se torne uma fonte de dados dinâmica do processo no qual participa. No entanto, quando o processo está inserido em um sistema distribuído, a tag passa a atuar também como mensageira de ações do processo. Dessa forma, ela se torna um recurso compartilhado entre os leitores para execução de processos independentes.

Quando define-se um sistema distribuído segundo (TANENBAUM; STEEN, 2013) como sendo um conjunto de computadores independentes que se apresentam aos seus usuários como um sistema único e coerente e (COULOURIS et al., 2013) como sendo aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens , deve-se observar que além do processamento independente é necessário que ocorra uma intercomunicação entre componentes. No caso da PNRD essa conexão é feita por meio da tag RFID, sendo portanto, a Fonte única da verdade(SSOT) e o Sistema de registro (SOR) responsável por enviar a mensagem aos outros componentes .

Uma fonte única da verdade funciona como um local centralizado de toda a infor-

mação de um sistema. Segundo (SCHUH et al., 2015) é uma dentre quatro conceitos que impulsionam a industria 4.0. De acordo com (SCHUH et al., 2015) receber informações de uma única fonte da verdade, é requisito fundamental para que uma empresa incorpore todos os dados do ciclo de vida de um produto ao longo da cadeia de valor.

Já o sistema de registro (SOR) é a fonte de dados onde os objetos são mantidos, comumente implementado em um Sistema Gerenciador de Banco de Dados Relacional (SGBDR), um software que atua como unidade certificadora para acesso aos dados. No caso da PNRD o SOR é basicamente um processo de leitura e gravação de mensagens binárias, de forma atômica, em um microchip seguindo o padrão NDEF (*NFC Data Exchange Format*). Nessa troca de mensagens existem algumas limitações como a taxa de dados trafegados, qualidade dos chips, tempo de gravação, variação da intensidade do sinal, capacidade de memória e corrupção dos dados armazenados. Essas limitações em grande parte são suprimidas pela atomicidade do processo e por excessões padronizadas descritas por (SILVA, 2017), mas os aspectos físicos da gravação e a limitação de memória das tags RFID dificultam a obtenção de dados históricos consistentes e tolerantes a falhas.

1.1 OBJETIVOS

Este trabalho apresenta um modelo de envio de dados para um sistema distribuído visando melhorar o processo de gravação de dados históricos da PNRD e que seja imutável, seguro e escalável. O principal objetivo deste projeto consiste na criação de um tipo de transação que permita de forma prática o envio de dados históricos decorrentes da leituras de tags RFID. Os objetivos específicos desse projeto são:

1. Integrar a PNRD como processo local a um sistema distribuído em nós.
2. Provisionar a infraestrutura necessária para a criação de nós distribuídos.
3. Criar um tipo de transação que torne possível a comunicação da PNRD com os nós da rede distribuída.
4. Desenvolver a interface de comunicação entre os leitores da PNRD e o modelo de transação criado.

1.2 JUSTIFICATIVA

A cadeia de suprimentos é uma rede de distribuição e de processamento intermediário de produtos e serviços, em outras palavras (MADUMIDHA et al., 2019) é o

processo de fabricar um produto e distribuí-lo, onde será novamente processado e distribuído mais uma vez, de forma cíclica, até chegar ao consumidor final.

Nesse modelo, os participantes envolvidos são o produtor, o fornecedor, o processador, o distribuidor e o varejo, cada um deles com sua própria metodologia de trabalho, sistema de gerenciamento e controle de dados. Nesse fluxo tradicional da cadeia de suprimentos existem problemas como a falta de transparência dos participantes, baixa rastreabilidade do produto e tecnologias divergentes entre participantes. O resultado disso são os atrasos na entrega, manipulação de preços, perda de suprimentos e inconsistência de dados, além de um maior tempo de espera entre as etapas da cadeia e o prejuízo financeiro.

A tecnologia em RFID como é o caso da PNRD já se tornou uma soluções para grande parte desses problemas, mas ainda existe uma carência por uma solução que conecte todas as etapas da cadeia de suprimentos, atuando tanto como referência nos processos individuais de produção e beneficiamento de produtos como nos processos macros de distribuição e fornecimento.

A integração da PNRD com uma rede distribuída de dados atende a esses requisitos, pois proporciona o gerenciamento descentralizado de processos com o auxílio da PNRD e a transparência e interconexão dos participantes por meio de uma rede global descentralizada.

Esse trabalho apresenta a fundamentação teórica no capítulo 2, seguindo da descrição e da metodologia, desenvolvimento e resultados no capítulo 3. Por fim, a conclusões é apresentados no capítulo 4, seguidos pelas referências bibliográficas. O código-fonte está distribuído ao logo dos capítulos e de forma mais concentrada em apêndice.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 TECNOLOGIA RFID

Historicamente, o uso de scanners ópticos nos sistemas de código de barras causava e ainda causa muitas limitações técnicas e operacionais, como alternativa, foi pensando no uso de radiofrequência para identificação e rastreamento de objetos, essa ideia foi proposto pela primeira vez por Watson-Watt em 1935 e denominado como sistema de identificação por radiofrequência (RFID), nele o leitor envia um sinal de busca criando um campo eletromagnético em direção à etiqueta. Este sinal é então processado pela unidade de microchip da etiqueta e transmitido de volta em direção ao leitor carregando as informações de identificação da etiqueta (EPC), quando o sinal chega ao leitor, ele é demodulado e convertido em dados.

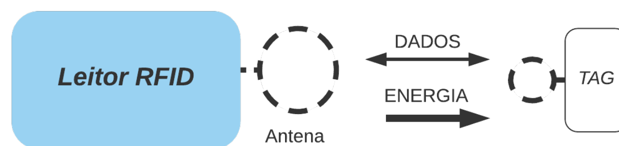

Figura 1 – Esquema de funcionamento tag RFID

O RFID possui três modos de comunicação, ativo, passivo e semi-passivo. A comunicação RFID passiva (Figura 1) envolve um leitor/gravador e uma etiqueta que não possui fonte de alimentação embarcada. As etiquetas obtêm sua fonte da energia do próprio campo eletromagnético. Geralmente é uma quantidade muito pequena, apenas o suficiente para enviar um sinal de volta ao leitor. Já a comunicação de um RFID ativo envolve uma tag com alimentação independente e um leitor/gravador conforme o caso passivo, como consequência dessa alimentação, sua resposta ao leitor pode viajar a distâncias bem maiores. Na comunicação semi-passiva, existe uma fonte de alimentação interna na tag, porém ela ainda necessita da fonte de energia do campo eletromagnético.

2.1.1 ETIQUETAS RFID

As etiquetas RFID (tags) passivas possuem 2 componentes, um microchip e uma antena conforme imagem abaixo (Figura 2), enquanto as etiquetas RFID ativas têm três componentes: um microchip, uma antena e uma bateria. O enrolamento da antena na

Tabela 1 – Classes de Tags

Classes	Descrição
Classe 0	Somente leitura
Classe 1	Uma escrita, muitas leituras
Classe 2	Regravável
Classe 3	Semi-passiva
Classe 4	Ativa

Fonte: (ZELBST; SOWER, 2021, RFID para supply chain and operations professinal)

Tabela 2 – Tipos de tags x frequência

Tipo	Descrição	Distância
Baixa frequência (LF)	125 - 134,3 kHz	Curta (até 0,1 Metros)
Alta frequência (HF)	13,56 MHz	Média (0,3 Metros)
Ultra alta frequência (UHF)	860 - 960 MHz	Até 3 Metros

Fonte: (ZELBST; SOWER, 2021, RFID para supply chain and operations professinal)

tag por ser aumentado para melhorar alcance do sinal.A regra é que para uma leitura consistente o comprimento da antena na etiqueta precisa ser de pelo menos um quarto da distância dos leitores. Por exemplo, se a distância da tag ao leitor for de 4 metros, a antena na etiqueta precisará ter 1 metro de comprimento quando esticada.(ZELBST; SOWER, 2021) Caso contrário, o sinal não alcançará o microchip para lhe dar energia para transmitir as informações necessárias.

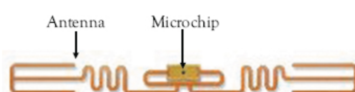

Figura 2 – Esquema de construção de uma Tag RFID

Fonte: RFID for Supply Chain and operations professional (ZELBST; SOWER, 2021)

As tags são geralmente definidas por classes (Tabela 3) e por frequências de operação(Tabela 2), com relação a classes, elas indicam a capacidade da tag para certas funções, como de gravação ou escrita. Além disso, a classe também pode indicar se a tag é do tipo ativa ou é passiva.A etiqueta UHF passiva mais comumente usada é a de Classe 1 Gen 2. Gen 2 refere-se aos requisitos de conformidade para a faixa de operação UHF de 860 a 960 MHZ para etiquetas RFID passivas. A seleção da etiqueta dependerá do uso pretendido e do ambiente operacional.

2.1.1.1 Idetificador Único

(ZELBST; SOWER, 2021)Um Código Eletrônico do Produto (EPC) associa uma etiqueta a um registro (alfanumérico) exclusivo, permitindo que um item específico,

Tabela 3 – Classes de Tags

Banco	Descrição	Infomação
00	Reservado	Kill Password
00	Reservado	Access Password
01	ECP	CRC-16
01	ECP	Protocol Control (PC)
01	ECP	Eletronic Product Code (ECP)
10	TID	Tag Identification
11	USER Memory	User

local de origem, data de produção e outras informações sejam atreladas a esse registro.No caso da rede PNRD esse registro é o que identifica a tag no banco de dados distribuído.

Um microchip em uma etiqueta passiva básica pode conter 96 bits de informação (Figura 3). Cada microchip possui quatro bancos de informações: Banco 0, Banco 1, Banco 2 e Banco 3

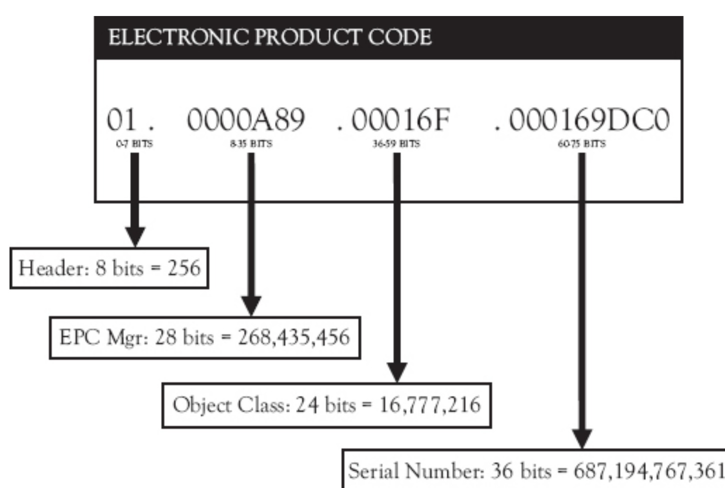

Figura 3 – Representação interna de um EPC

Fonte: RFID for Supply Chain and operations professional (ZELBST; SOWER, 2021)

Além de chips com 96 bits muitos fabricantes desenvolveram chips de tags com memórias EPC de 128, 240 e 496 bits. Alguns chips UHF fornecem memória de até 512 e 640 bits, além da memória EPC para o armazenamento de informações adicionais como data, hora e identificação do produto. Porém é evidente que essa capacidade de armazenamento não é suficiente para o armazenamento de dados históricos da PNRD justificando nossa alternativa de um banco de dados distribuído.

2.1.2 ANTENAS DOS LEITORES

A antena do leitor RFID é a responsável por enviar as onda de rádio que ativam a antena de uma tag passiva e fornecer energia para a tag transmitir dados. As antenas podem ser controladas por software e pelo leitor, sendo configuradas para coletar informações em um determinado intervalo de leitura. (ZELBST; SOWER, 2021) Ao escolher uma antena de leitor, existem várias questões que devem ser consideradas:

1. Polarização
2. Tipo de Antena
3. Posição

2.1.2.1 Polarização

(ZELBST; SOWER, 2021) Polarização se refere à direção das ondas eletromagnéticas. As antenas para RFID são polarizadas linearmente ou circularmente. Para antenas polarizadas linearmente, a leitura só é possível quando a antena e a etique estão alinhadas.Já para antenas polarizadas circularmente, a etiqueta pode ser colocada em quase qualquer lugar do campo eletromagnético e ainda assim ser lida.

2.1.2.2 Tipos de Antenas

(ZELBST; SOWER, 2021) Uma antena mono-estática, possui uma única antena para enviar e receber sinais. A antena mono-estática requer apenas um conector ligado ao leitor e é a mais utilizada do mercado. Já nas antenas bi-estáticas, possuem 2 antenas e portanto, dois conectores ligador ao leitor, sendo uma para recebe dados e outra para enviar.

2.1.2.3 Posição

O posicionamento correto da antena é um dos principais aspectos para se evitar o sombreamento das etiquetas. (ZELBST; SOWER, 2021) O sombreamento ocorre quando uma etiqueta passiva não é ativada porque outras etiquetas estão empilhadas umas sobre as outras, evitando que a antena da tag receba o sinal para energizá-la. Um bom posicionamento leva em consideração o tipo de antena e a forma como ela será utilizada para energizar as tags.

2.1.2.4 Leitor

O leitor é um sistema eletrônico que gera e interpreta os sinais recebidos pela antena. Ele embute algoritmos contra colisões e pode, por vezes, operar com mais de uma frequência de sinal (AHSAN; SHAH; KINGSTON, 2010).Em outras palavras ele é

um componente responsável pela modulação e demodução dos sinais trafegados pela etiqueta.

2.1.3 NFC

(IGOE; COLEMAN; JEPSON, 2014) O NFC pode ser considerado uma extensão do RFID, possuindo o mesmo princípio de funcionamento e operando na alta frequência (HF) da tecnologia RFID (frequência de 13,56 MHz). No entanto, ele pode fazer mais do que apenas ler ids e gravar dados no destino.

A diferença mais interessante entre RFID e NFC é que as tags NFC costumam estar atreladas a dispositivos programáveis, como telefones e celulares e podem possuir uma comunicação bilateral. Isso significa que, em vez de apenas entregar dados estáticos da memória, uma tag NFC poderia gerar conteúdo exclusivo para cada troca de informação e devolvê-lo ao iniciador.(IGOE; COLEMAN; JEPSON, 2014)

Os dispositivos NFC têm dois modos de comunicação que são idênticos ao RFID, com tags passivas e ativas, porém no NFC existem três modos de operação, conforme abaixo:

- Sendo leitores/gravadores para leitura e gravação de dados do destino.
- Emuladores de tag, agindo como etiquetas RFID quando estão no campo de outro dispositivo NFC ou RFID.
- Modo ponto a ponto , no qual as trocam dados ocorrem em ambas as direções.

2.1.3.1 NDEF

A NDEF define uma troca de dados em mensagens, que são compostas de registros NDEF, o que possibilita que o código em uma camada de aplicação Figura 4 lide com dados de leitura/gravação de tags NFC, comunicação ponto-a-ponto e emulação e etiquetas emuladas. Os dados trafegados entre dispositivos e tags NFC são formatados usando o padrão *NFC Data Exchange Format (NDEF)*. O NDEF foi um dos principais avanços que o NFC adicionou ao RFID. (IGOE; COLEMAN; JEPSON, 2014) Cada mensagem NDEF contém um ou mais registros NDEF e cada registro possui um tipo específico, um ID exclusivo, um comprimento e uma carga útil de dados. Os tipos mais conhecidos do padrão NDEF são:

- Registros de texto simples (contém uma *string* de texto)
- URIs (Contém um endereço web)
- Cartazer Inteligentes (urls e dados como mensagem sms)

- Assinaturas (Fornece informações de maneira confiável)

O tipo PNRD do Padrão NDEF detalhado por (SILVA, 2017) para tornar possível a troca de informações entre leitores e tags foi o tipo adotado nesse projeto para inserir dados em tags PNRD e iPNRD.

Ambos os protocolos NFC e muitos dos protocolos RFID operam em 13,56 MHz, mas as etiquetas RFID não precisam formatar seus dados no formato NDEF. Os vários protocolos RFID não compartilham um formato de dados comum.

A arquitetura NFC é composta por uma camada física, outra de empacotamento e transporte e mais acima a camada de aplicação, conforme (Figura 4). Na camada física estão a CPU e os dispositivos de comunicação, na de empacotamento e transporte estão as especificações de comunicação, empacotamento de dados e modelos de tags, já na camada de aplicação estão os tipos de mensagens NDEF e o código que será executado.

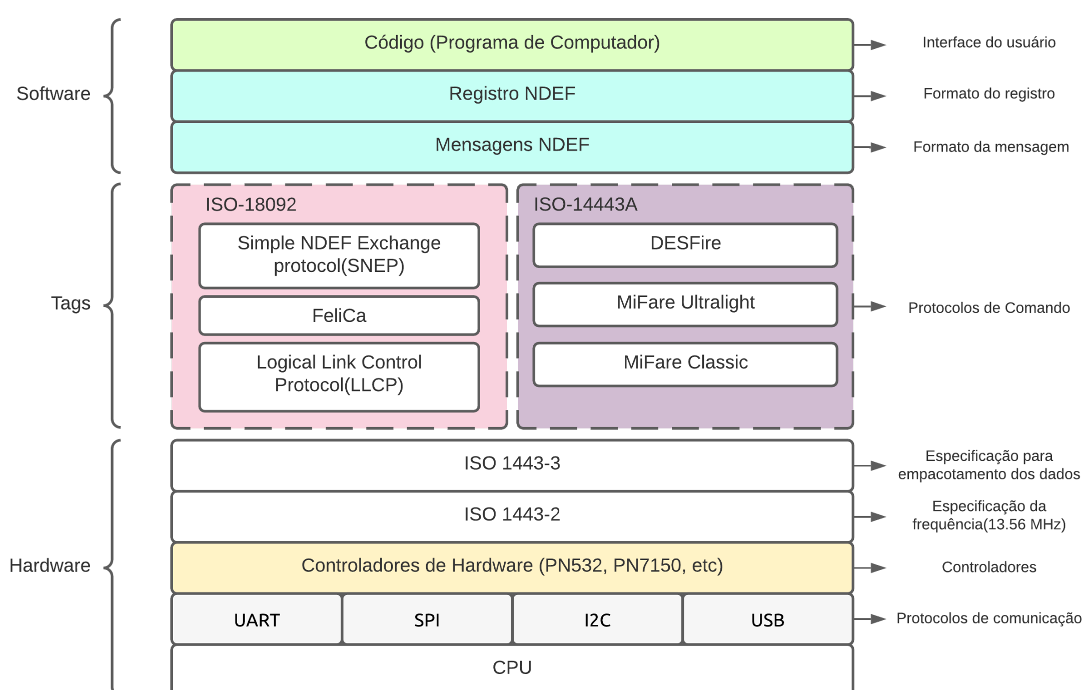

Figura 4 – Estrutura do protocolo NFC (IGOE; COLEMAN; JEPSON, 2014)

Na camada física, o NFC opera com uma frequência de 13,56 MHz (frequência intermediária no RFID) seguindo a ISO-14443-2. Em seguida, vem uma camada que descreve o enquadramento dos bytes de dados enviados por radiofrequência, ISO-14443-3. Depois existe a comunicação serial com os protocolos:

- Recepção e transmissão assíncrona universal (UART)
- Interface periférica serial (SPI)
- Comunicação de circuito integrado (I2C)
- Barramento serial universal (USB).

Já na camada de empacotamento (Tag) conforme Figura 4, existe a especificação de leitura e gravação de tags NFC construída sobre a ISO-14443A. Os protocolos Philips/NXP Semiconductors Mifare Classic e Mifare Ultralight e NXP DESFire são compatíveis com ISO-14443A. A comunicação NFC de ponto-a-ponto é baseada no protocolo de controle ISO-18092 e é uma das diferenças entre RFID e NFC, a outra diferença está no formato de transferência de dados (NDEF).

2.2 REDES DE PETRI

As redes de Petri (PN) são um modelo conceitual gráfico e matemático introduzido por Carl Adam Petri em 1962.As Redes de Petri têm sido usadas para modelagem, simulação e análise de desempenho de sistemas de eventos discretos. A representação gráfica e matemática permite análises completas de sistemas (como análise de espaço de estado, gargalos de desempenho e prevenção de inter-bloqueio (*deadlock*) (SILVA, 2017).

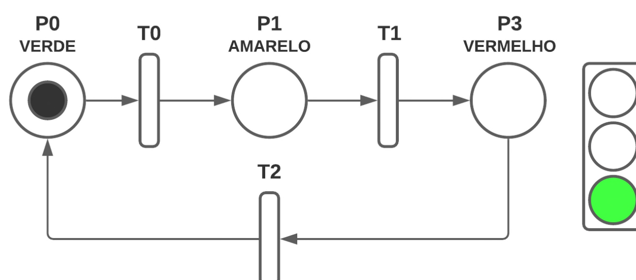

Figura 5 – Representação de um semáforo em redes de petri no estado verde

Um rede de petri possui quatro elementos, os Lugares (P), as Transições (T), os Arcos (A) e as fichas (tokens). Os lugares são geralmente representados por círculos ou elipses, as transições costumam ser representadas por retângulos ou quadrados e os arcos são as setas que ligam uma transição a um arco e vice-versa, já os tokens são bolinhas pretas que ficam dentro dos lugares. Na Figura 5, é exibida uma representação gráfica de uma rede de petri para um semáforo. Nela existem três lugares P0,P1,P2 e três transições T0,T1,T2, já a ficha (bolinha preta) está localizado dentro do lugar P0.

2.2.1 DISPARO

(SILVA, 2017) Um disparo acontece quando uma transição ou um conjunto de transições habilitadas é acionada de forma a alterar a marcação de uma RP. No exemplo do semáforo, o disparo da transição T0 altera o estado do semáforo de verde para amarelo Figura 6.Porém esse disparo só pode ser realizado caso existam token na entrada da transição T0 (veja **definição 2.2.3**)

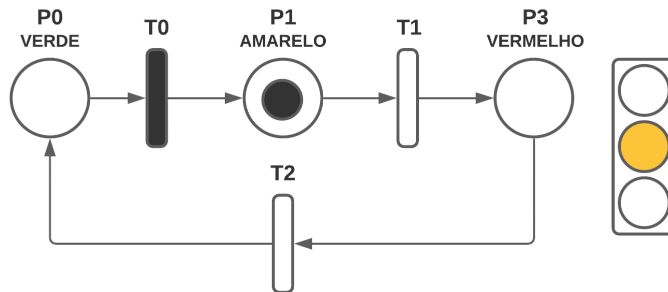

Figura 6 – Disparo da transição T0 movimentando o token de P0 para P1

2.2.2 DEFINIÇÃO FORMAL

2.2.2.1 Redes de Petri Lugares/Transições

Definição 2.2.1. Os Lugares e Transições de uma rede de petri são definidos por uma tupla de quatro elementos:

$$PTN = (P, T, A, M_0) \tag{2.1}$$

onde,

- P é um conjunto finito de lugares, $P = \left\{p_1, p_2, ..., p_{n_p}\right\}$
- T é um conjunto finito de transições, $T = \{t_1, t_2, ..., t_{n_t}\}.P \cap T = \emptyset$
- A é o conjunto de arcos (Dos lugares para as transições e das transições para os lugares).
- $A \subseteq (P \times T) \cup (T \times P)$ por padrão W (pesos dos arcos) de a_{ij} é 1, a não ser que seja especificado o contrário. ($a_{ij} \in A$ um arco vindo de p_i para t_j ou de p_j para t_i)
- M é o vetor marcações (tokens) no conjunto de Lugares.
- $M = [M(p_1), M(p_2), ...M(p_{n_p})] \in N^{n_p}$, M_0 é o vetor de marcação inicial. Devido as marcações, a $PTN = (P, T, A, M)$ é também chamada de **Rede de petri de P/T Marcada**

Definição 2.2.2. A transição t_i é habilitada no vetor de Marcação M se,

$$\forall p \in P = W(p, t_i) \leq M(p) \tag{2.2}$$

ou seja, a transição t_i só será habilitada apenas se todos os locais de entrada dela (lugares), possuírem uma quantidade de tokens que seja pelo menos igual ao respectivo peso do arco.

Definição 2.2.3. (Entradas e saídas de Lugares) Os elementos p de entrada de uma transição t, denotado por $\bullet t$, são o conjunto de todos os lugares que estão diretamente na entrada da transição t:

$$\forall p \in \bullet t = W(p, t) \neq 0 \tag{2.3}$$

Da mesma forma os elementos p de saída de uma transição t, denotada por $t\bullet$ são o conjunto de todos os lugares que estão diretamente ligados a saída de t

$$\forall p \in t\bullet = W(t, p) \neq 0 \tag{2.4}$$

De forma similar os conjuntos de transições de entrada dos lugares e os conjuntos de transições de saída do lugares, podem ser denotados por $\bullet p$ e $p\bullet$ respectivamente.

Definição 2.2.4. (Novo vetor De marcação) Quando uma transição habilitada t_i dispara, o novo vetor de marcação M_{j+1} é obtido do vetor de marcação anterior M_j da seguinte forma:

$$\forall p_i \in P, M_{j+1}(p_i) = M_j(p_i) - W(p_i, t_i) + W(t_i, p_i). \tag{2.5}$$

O disparo de uma transição t_i é denominada por $\overrightarrow{t_i}$.

2.2.2.2 Equação de Estado e Matriz de incidência

Definição 2.2.5. A matriz de incidência (A) de uma rede de petri com n_p lugares e n_t transições é dada por:

$$A_{(n_p \times n_t)} = [a_{ij}] \tag{2.6}$$

onde, $a_{i_j} = W(t_j, p_i) - W(p_i, t_j)$, ou seja elemento da matriz de incidência a_{i_j} tem um valor -1 quando o lugar p_i for uma entrada da transição t_j e tem um valor 1 quando o lugar p_i for uma saída da transição da transição t_j. Caso não exista uma relação entre lugar e transição, o valor é 0.

Definição 2.2.6. A equação de estado de uma rede de petri P/T Marcada $PTN = (P, T, A, M_0)$ apresenta um novo vetor de marcação devido ao disparo do vetor f(t) (vetor de disparo)

$$M^| = M_0 + A^T \times f(t) \tag{2.7}$$

Onde A^T é a matriz de incidência transposta. O vetor coluna de estado $M^|$ gerado pela equação de estado devido a uma série de disparos de f(t) no estado inicial M_0 é denotado pelo operador $M_0[f(t) >> M^|$

2.2.3 REPRESENTAÇÃO GRÁFICA

A grande vantagem da rede de Petri é a sua capacidade de representar sistemas caracterizados como sendo concorrentes, assíncronos, distribuídos, paralelos, não determinísticos e/ou estocásticos (SILVA, 2017). para ilustrar essa capacidade vamos representar um processo de teste de peça usinada, em redes de petri, conforme Figura 7

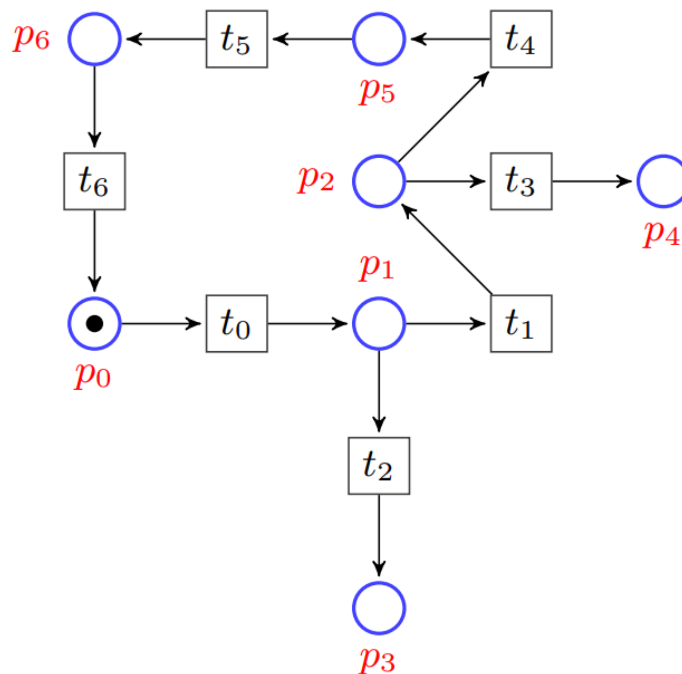

Figura 7 – Esquema de teste de peça usinada

Fonte: Improving Exception Analysis in PNRD systems using Labelled Free Choice Nets (TAVARES; PHAWADE, 2010)

Na representação acima, existem 7 transições $T = \{t0, t1, t2, t3, t4, t5, t6\}$, 7 lugares, $P = \{p0, p1, p2, p3, p4, p5, p6\}$ e um token em $p0$ conforme descrito em M_0:

$$M_0 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$$

A matriz de incidência transposta conforme definição 2.2.5 pode ser representada por:

$$A^T = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

Para encontrar o próximo vetor de marcação M_k é necessário efetuar o disparo em t0 e conforme definição 2.2.2 t0 é uma transição habilitada, portanto, seguindo a equação 2.8:

$$M_k = M_0 + A^T \times f(t) \tag{2.8}$$

sendo $f(t0) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$ foi obtido:

$$M_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

portanto, $M_1 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^T$, ou seja o token saiu de p0 e foi para p1. Semelhante ao caso do semáforo na Figura 6.

2.3 PNRD

Redes de Petri Elementares inseridas em base de dados RFID é uma estrutura de dados formal a ser armazenada na etiqueta RFID de forma a integrá-la com sistema de controle. (TAVARES; SARAIVA, 2010)

Na PNRD ocorre a união do mapeamento do processo seguindo a modelagem em redes de petri e o padrão NFC para aplicação do modelo matemática em uma estrutura física composta por tag e leitor/antena.A matriz de incidência transposta A^T e o vetor de marcação inicial M_0 são enviados para a tag através do tipo PNRD (protocolo NDEF) e o leitor/antena recebe o vetor de disparo seguindo a padronização RFID de controladores como o PN532, PN7150 e etc (Figura 4). Todo esse processo funciona da seguinte forma:

A Tag envia a informação ao leitor, que recebe os dados, decodifica e realiza os cálculos da rede de petri, criando um novo vetor de marcação M_k e um status padronizado, caso esse status seja `NO_ERROR` decorrente da rede de petri, é feita uma gravação na tag desse vetor M_k, caso não exista nenhum problema no processo de gravação o leitor retorna o status `NO_ERROR` para a aplicação conectada ao leitor. Esse processo se repete até o fim de todas as etapas mapeadas.

2.3.1 EXCEÇÕES

Na PNRD, todos os disparos são padronizados seguindo as biblioteca de (SILVA, 2017).

Para o módulo *PetriNet*:

- `NO_ERROR` - No contexto de verificação de disparo, significa que o disparo descrito não gera erros. No contexto de realização do disparo, significa que o disparo foi feito e o vetor de marcação foi atualizado.
- `CONDITIONS_ARE_NOT_APPLIED` - Indica que as condições associadas às transições do disparo não estão satisfeitas;
- `PRODUCE_EXCEPTION` - Indica que o disparo resulta em exceção, ou seja, não existem marcações suficientes nos lugares de entrada para efetuar as transições
- `NOT_KNOWN` - Quando o programa segue por um fluxo inesperado.

Para o módulo PNRD:

- `NO_ERROR` - A operação de escrita ou leitura foi efetuada com sucesso;
- `TAG_NOT_PRESENT` - Não existe nenhuma etiqueta RFID compatível com o leitor na sua proximidade;
- `INFORMATION_NOT_SAVED` - Exclusivo no caso de escrita. Indica que ocorreu um erro no ato de gravação dos dados na etiqueta;
- `INFORMATION_NOT_PRESENT` - Exclusivo no caso de configuração inicial, indica que nem todas as informações configuradas estão presentes na etiqueta lida.
- `DATA_SIZE_NOT_COMPATIBLE` - Exclusivo no caso de leitura. Significa que o objeto Pnrd criado não alocou uma quantidade de memória adequada para representar as informações contidas na etiqueta;
- `NOT_ENOUGH_SPACE` - Exclusivo no caso de gravação. Indica que a etiqueta não tem memória interna suficiente para gravar os dados;

- `NOT_AUTORIZED` - significa que o ato de escrita ou leitura não foi autorizado pela etiqueta;
- `VERSION_NOT_SUPPORTED` - quer dizer que o leitor não suporta a versão de dados presentes na etiqueta;
- `ERROR_UNKNOWN` - significa a ocorrência de um erro genérico não especificado.

Com essas condições é possível saber o que ocorreu no processo de gravação/leitura da tag e com isso, definir a melhor abordagem. Por exemplo, o retorno `NO_ERROR` é o cenário ideal do processo principal, já no caso `PRODUCE_EXCEPTION` existe um problema matemático, que indica que aquela tag está em um posição incorreta e no `CONDITIONS_ARE_NOT_APPLIED` ocorre uma condição externa ao processo.

2.4 SISTEMA DISTRIBUÍDO

No campo da computação, um sistema distribuído é aquele em que o processamento não é feito apenas em um computador. Em vez disso, ele é distribuído ao longo de vários nós. Esses nós se comunicam entre si por meio de alguma forma de mensagem.(LANTZ; CAWREY, 2020) Um sistema distribuído possui características de descentralização, em que a falha de uma única entidade (ou nó) não significa a falha de toda a rede.

O objetivo comum de um sistema distribuído é usar o poder de processamento para realizar coletivamente uma tarefa, distribuindo a responsabilidade por muitos computadores. No entanto, a descentralização muda o conceito de objetivo comum e mensagem. Em um sistema totalmente descentralizado, um determinado nó não colabora necessariamente com todos os outros nós para atingir seu objetivo, além disso a tomada de decisão é feita por meio de alguma forma de consenso, em vez de atribuí-la a uma única entidade.

2.5 BLOCKCHAIN

Blockchain é uma banco de dados de transações, distribuído e que realiza somente inserção de dados, pode também ser definido como uma cadeia de blocos de informação distribuída, funcionando como um sistema de registro compartilhado, seguro e em constante crescimento, no qual cada consumidor dos dados mantém uma cópia dos registros, que só podem ser atualizados se todas as partes envolvidas em uma transação concordarem. Vale ressaltar que essa atualização não significa modificar os dados inseridos e sim uma nova inserção de dados. De forma mais técnica,

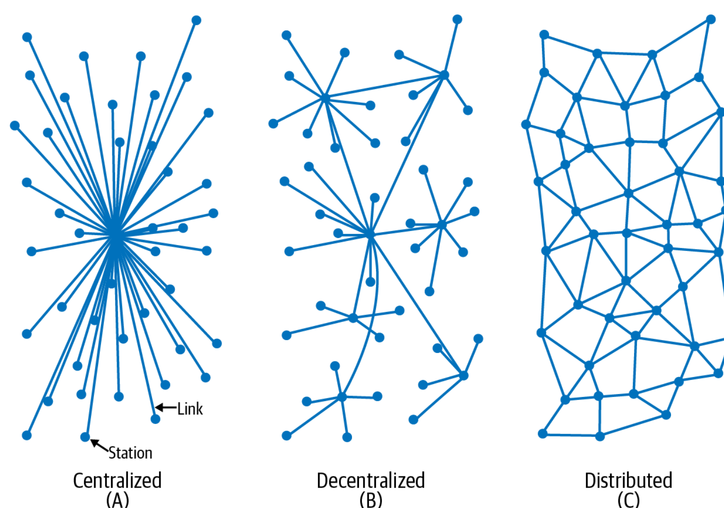

Figura 8 – Comparativo de diferentes redes, em (A) uma rede centralizada, em (B) uma rede descentraliza e em (C) uma rede distribuída

Fonte: Mastering Blockchain (LANTZ; CAWREY, 2020)

(BASHIR, 2020) blockchain é um livro-razão distribuído de ponta-a-ponta (P2P), criptografado e imutável (Extremamente difícil de ser alterado).

Camada	Componentes
Aplicações	Contratos Inteligêntes Aplicativos Descentralizados Organizações Autônomas e Descentralizadas Agentes Autônomos
Execução	Maquina Virtual Blocos Transações
Consenso	Replicação de Máquina de Estados Prova baseada em consenso Protocolo de tolerância a falhas bizantinas
Criptografia	Criptografia de chave pública Assinatura Digital Funções HASH
P2P	Protocolos de roteamento Protocolos de Flooding
Rede	Intenet TCP/IP

Figura 9 – Representação genérica de um *Blockchain* (LANTZ; CAWREY, 2020)

O esquema acima Figura 9 exibe um blockchain genérico com uma estrutura simplificada dos principais componentes que o integram, começando pela rede, camada mais baixo nível do sistema, sendo responsável pela comunicação do blockchain de acordo com uma rede P2P. Depois existe a camada de criptografia, responsável por

toda a segurança do *blockchain*, em seguida os mecanismos de consenso, que garantem a descentralização por meio de votos e sistemas de tolerâncias a falhas. Já a camada de execução é responsável por fornecer serviços ao *blockchain*, como máquinas virtuais e pra o processamento de dados como criação de bloco e transações. Por fim, a camada de aplicativos, que é a ponte entre o usuário e o *blockchain*, é ela que irá se comunicar diretamente com o blockchain, enviando transações e solicitando autenticação do usuário.

2.5.1 CADEIA DE BLOCOS

Conforme a definição de blockchain, sua construção ocorre por meio de uma série de blocos interconectados, partindo do bloco gênesis. Conforme Figura 10

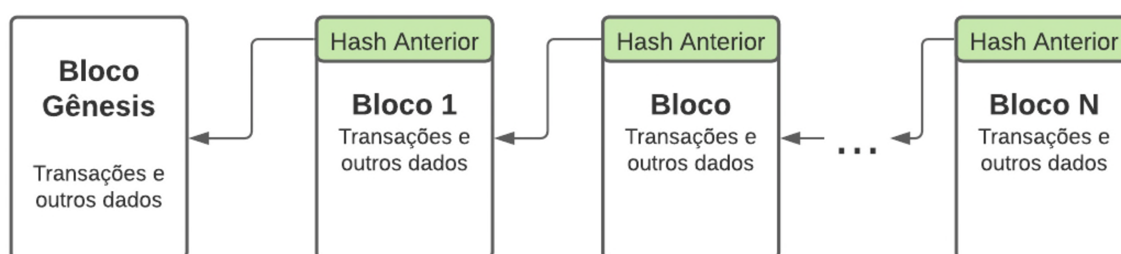

Figura 10 – Representação genérica de um *Blockchain*

2.5.1.1 Blocos

Um bloco é composto por várias transações e outros elementos (Figura 11), como a hash do bloco anterior (ponteiro para a hash), um registro de hora/data e um *nonce*. Um bloco é composto de um cabeçalho de bloco(*block Header*) e uma seleção de transações agrupadas e organizadas logicamente. Um bloco contém referência a um bloco anterior(hash do cabeçalho do bloco anterior).

O bloco gênesis é o primeiro bloco do *blockchain*, criado no momento em que o *blockchain* é iniciado pela primeira vez.

Um *nonce* é um número gerado e usado apenas uma vez, ele é usado em operações criptográficas para fornecer proteção de reprodução, autenticação e criptografia. No *blockchain*, é utilizado um algoritmos de consenso como o PoW para a proteção das transações.

Um hash combinado de todas as transações do bloco (chamado de hash da árvore Merkle (*Merkle root*) é usado para permitir a verificação eficiente das transações, esse hash está presente na seção de cabeçalho do bloco, com ele é necessário verificar apenas o hash Merkle do bloco em vez de verificar todas as transações uma por

uma. Além do cabeçalho do bloco, o bloco contém as transações, compondo assim o corpo do bloco. Uma transação é o registro de um evento, por exemplo, o evento de transferência de dinheiro da conta de um remetente para a conta de um beneficiário.

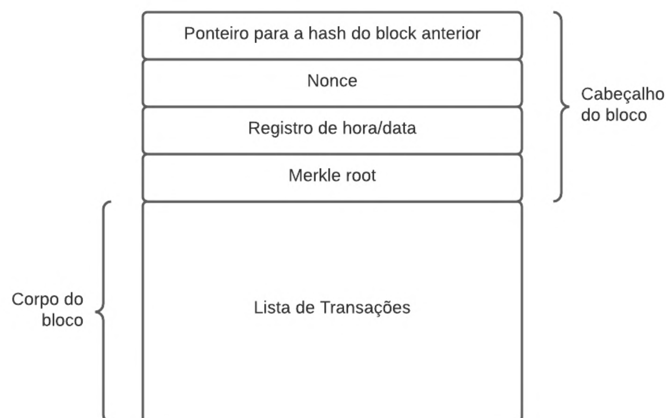

Figura 11 – Representação genérica de um bloco

```
{
    "data": {
        "header": {
            "signer_public_key": "03cf85....",
            "transaction_ids": [
                "9997920e0908e09165a41b32f551d...."
            ]
        },
        "header_signature": "c3c6aad39c61642b97....",
        "trace": false,
        "transactions": []
    },
    "link": "https://n1.pnrd.net/sawtooth/batches/c3c..."
}
```

Código 2.1 – Exemplo de um bloco gerado da rede PNRD

2.5.1.2 Transação

Uma transação é a unidade fundamental de um *blockchain* e representa a inserção de dados definida por um endereço de identificação do ator dessa transação, essa inserção pode ser:

- A transferência de um ativo, como comprar usando uma criptomoeda ou ganhar ponto de benefício.
- Mudança de localização de um produto quando rastreado em uma cadeia de suprimentos.
- Atualização de dados pessoais.

```
{
    "header": {
        "batcher_public_key": "03cf85....",
        "dependencies": [],
        "family_name": "pnrd_net",
        "family_version": "0.1",
        "inputs": [
            "d451ac0020c53a23e1ca...."
        ],
        "nonce": "",
        "outputs": [
            "d451ac0020c53a23e1ca...."
        ],
        "payload_sha512": "d13d23...",
        "signer_public_key": "0317174...."
    },
    "header_signature": "99979....",
    "payload": "Eg8KDVJvZ2VyIENhcnJpam8wnaPGiwY="
}
```

Código 2.2 – Exemplo de uma transação gerada pela rede PNRD

2.5.1.3 Endereço

É um identificador único, usado para definir quem envia/recebe transações ou participa de forma indireta, como por exemplo o endereço de um token (criptomoeda).

2.6 HYPERLEDGER SAWTOOTH

Sawtooth é um blockchain de nível empresarial que pode ser executado em modo permissionado ou não-permissionado. Um ambiente permissionado é todo aquele que necessita de permissão para entrar, no caso do blockchain essa permissão pode ocorrer por meio de votação ou autorização de uma entidade superior.

Seus diferenciais são, estrutura modular, processamento paralelo de transações e separação entre Aplicação e Sistema Central.

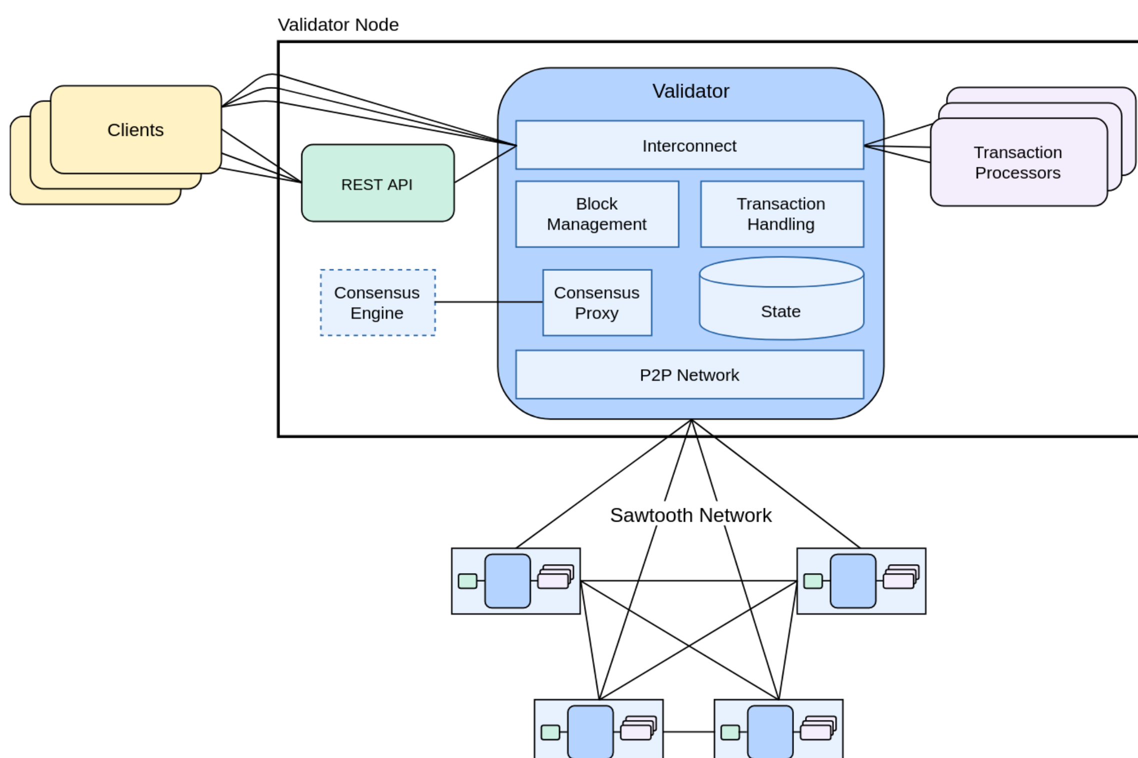

Figura 12 – Arquitetura do Blockchain Hyperledger Sawtooth

Fonte: Sowtooth Documentation <https://sawtooth.hyperledger.org/docs/core/releases/latest/>

2.6.1 FUNCIONAMENTO

Assim como a grande maioria dos blockchain o Sawtooth é composto por uma Árvore Merkle para o endereçamento das camadas de blocos, mecanismo de consenso, blocos, transações e uma camada de aplicação conforme Figura 12.

2.6.1.1 Merkle-Radix

Além de armazenar seus estados seguindo uma árvore Merkle, o sawtooth integrou o conceito de árvore Radix, para que as famílias e sub-elementos das transações pudessem ser facilmente endereçáveis. Conforme Figura 13 existem 35 bytes para o

endereço, sendo os 3 primeiros bytes o prefixo do nome da família geradora da transação e os outros 32 bytes restantes para o endereçamento de todos os sub-elementos da família. Por exemplo, na família `pnrd_net` os 3 primeiros bytes são reservados para o nome `pnrd_net`, seguido pelo tipo de transação, que no caso pode ser record, para gravação de informações das tags ou *owner*, para definir um proprietário, o restante dos bytes é definido pela transação.

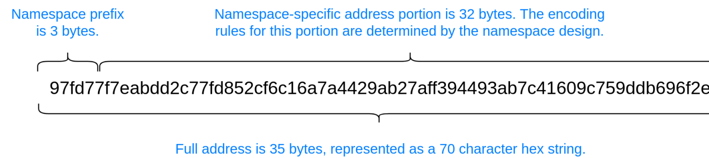

Figura 13 – Arquitetura do Blockchain Hyperledger Sawtooth

Fonte: Sowtooth Documentation <https://sawtooth.hyperledger.org/docs/core/releases/latest/>

2.6.1.2 Transações

As transações são as responsáveis pelas modificações de estado do sistema (Figura 14). Um cliente se comunica com a aplicação inserindo sua chave privada e os dados que desejar, em seguida essa ou essas transações criadas são encapsuladas em um lote (*batch*) e enviadas para o validador da família, esse validador extrai os dados, realiza um processo de validação e em seguida envia para o validador do nó, que será o responsável pelo processo de inclusão no próximo bloco disponível.

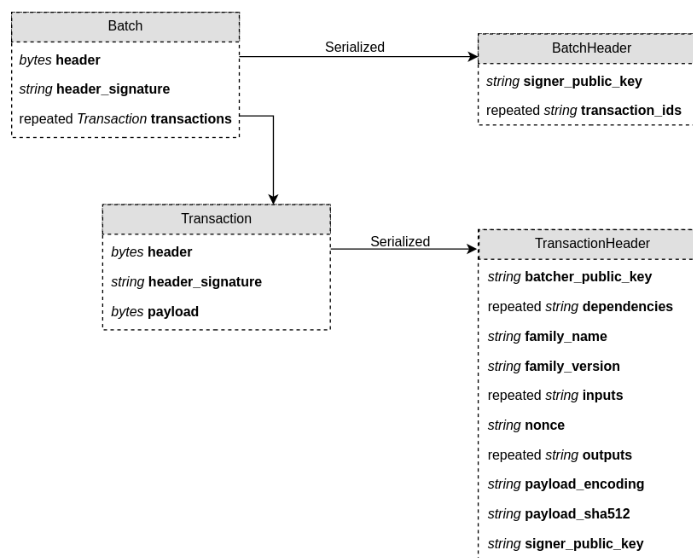

Figura 14 – Estrutura básica de um lote (batch) em protobuf

Fonte: Sowtooth Documentation <https://sawtooth.hyperledger.org/docs/core/releases/latest/>

2.6.1.3 Consenso

Atualmente o *blockchain Sawtooth* possui dois mecanismos de consenso disponível para ambiente de produção o PBFT e o PoET. A principal forma de utilização desses mecanismos é de forma separada.

No caso do PBFT (Tolerância a falhas bizantinas prático) é um mecanismo para rede permissionada baseado em votação. Ou seja para a inclusão de uma nova funcionalidade é necessário que os nós membros do *blockchain* votem. Esse mecanismo é utilizado quando é necessário ter uma rede que defina manualmente quais devem ser os nós participantes e quem deve ter direito ao voto.

Já *PoET* (Prova por Tempo decorrido), permite que qualquer nó entre na rede e valide blocos (não-permissionado), nesse mecanismo um nó é selecionado aleatoriamente com base no tempo que esperou antes de propor um bloco. Ele é utilizado para criar grandes redes com vários nós.

2.6.1.4 Camada de Aplicação

Para que uma aplicação se comunique com o *blockchain* é necessário que exista uma infraestrutura previamente configurada, com uma *REST API* pública e um processador de transação inserido no core (sistema principal) do *Sawtooth*. Todo o processo de configuração é detalhado no apêndice Infraestrutura *Sawtooth* (Apêndice A).

Toda aplicação *sawtooth* depende da família de transação, que é a estrutura que se comunica com o validador de blocos, nela o processador de transação validar as requisições executadas pelo cliente, que em seguida é despachado para o processador pelo despachante de lotes, componente da camada de aplicação que irá serializar os dados, incluir cabeçalhos, endereços proprietários das transação e encapsular o conjunto de transações em um ou mais lotes.

Após o processamento dos dados o processador envia os dados para o validador que os insere no banco e retorna o status da inserção ao processador, que retorna ao despachante.

2.6.2 MÓDULOS PRINCIPAIS

2.6.2.1 Validador

O validador tem dois componentes principais, o controlador da cadeia e o editor de blocos, ele usa agendadores para calcular as mudanças de estado e *hashes Merkle* resultantes com base no processamento da transação.

O controlador da cadeia (Cadeia de blocos) é responsável por manter o cabeçalho da cadeia atualizado, sempre apontando para o último bloco criado. Ele recebe os

candidatos a blocos e define quem será o próximo bloco publicado de acordo com um calendário.

O Editar de blocos é o responsável por criar novos candidatos a blocos, de acordo a fila de lotes (*baches*)despachados do processador.

2.6.2.2 Processador de Transação

A separação do sistema central (core) com a aplicação ocorre por meio de uma família, chamada de Família de Transação. Ela é definida por um estrutura em *proto-buffer* que é a base para toda a aplicação. Para que essa família seja integrada ao *sawtooth* é necessário que ela tenha um processador, um programa responsável pela validação das transações e manipulação dos dados para serem incluídos em uma fila para a inserção no *blockchain*.

2.6.2.3 REST API

REST API é uma interface de programação de aplicações e no caso do sawtooth serve como ponte de comunicação entre a aplicação cliente e o *blockchain*, por meio de um endereço web.

2.6.2.4 Cliente

O cliente é um software construído para a interação do usuário final com o *block-chain (REST API)* por meio de chaves públicas e privadas. Ele precisa possuir um despachante (descrito na sessão sobre a Camada de Aplicação), os dados da família de transação e as regras de negócio da aplicação.

3 METODOLOGIA E DESENVOLVIMENTO

Esse trabalho aborda a criação de um banco de dados distribuído seguindo uma análise de requisitos de projeto, comparando o desempenho de diferentes tecnologias. Posteriormente, foi feita uma análise experimental de um problema em redes de petri e a criação de uma família de transação seguindo uma abordagem incremental de melhorias, partindo do modelo mais básico e rudimentar até uma solução completa, dividida em módulos. No final foram realizados teste de integração e de implementação, por fim, o ambiente foi disponibilizado publicamente. A metodologia adotada para o desenvolvimento do trabalho foi a de pesquisa aplicada focando na conversão de conceitos teóricos em aplicações práticas e funcionais.

3.1 REQUISITOS DE PROJETO

Partindo da necessidade de salvar dados históricos de um processo que utilize a PNRD, foram definidos os seguinte requisitos funcionais:

- Obter os dados históricos a qualquer momento.
- Entender o fluxo do processo de um ponto de vista observador, sem interferência no processo.
- Identificar quais *tags* sofreram exceção.
- Filtrar informações do processo por *tag*.
- Disponibilizar os dados publicamente com o objetivo de promover a transparência.
- Criar uma forma de transferir a propriedade de um objeto (*tag*) para outro agente, (venda de produto)
- Ser democrática, com o objetivo de permitir que qualquer agente crie suas próprias soluções
- Possuir um ambiente decentralizado, para evitar conflitos de interesse.

Já os requisitos não funcionais podem ser definidos por:

- Possuir uma arquitetura de alta disponibilidade.
- Ser altamente segura.
- Ter uma estrutura de fácil integração com periféricos

Analisando os requisitos foi percebido que eles podem ser convertidos em 3 necessidades.

Aquisição de dados com informações da *tag*, do processo (rede de petri), da situação atual (data/hora), do status (`NO_ERROR`, etc) e para isso é necessário obter esses dados e enviá-los.

Transferência de propriedade, ou seja vender/doar um produto para outra pessoa.

Ser descentralizado, público, seguro e imutável (Não permitir que modificações sejam apagadas). Para isso seria necessário mapear quais bancos de dados atendem a esses requisitos.

3.1.1 COMPARAÇÃO DE DESEMPENHO ENTRE SOLUÇÕES DE BANCOS DE DADOS

Seguindo um comparativo entre diferentes soluções, levando em consideração além dos requisitos, conceitos como, ser um software livre, ter uma comunidade ativa, ser uma solução já consolidada, flexibilidade e baixo custo de implementação. Foram identificadas 2 soluções que atenderam aos requisitos (descentralizado, público e seguro). Uma delas é o mongoDB integrado ao Bigchain, uma solução de código aberto para sistemas que precisem ser Descentralizados e possuam dados Imutáveis. A outras foi o Hyperledger Sawtooth, solução também de código aberto, mas com ambiente além de descentralizado, distribuído e com dados Imutáveis.

3.1.1.1 MongoDB

A princípio a solução para o problema foi desenvolver uma estrutura distribuída e permissionada de bancos de dados MongoDB, pois devido ao mecanismo de replicação de *cluster* em escala horizontal, seria possível provisionar dezenas/centenas de nós para as aplicações, ou seja uma estrutura altamente escalável e distribuída. Porém a imutabilidade do sistema ficaria por conta da camada de software que permitiria ou não o acesso ao sistema, isso poderia ocasionar uma falha nos dados caso a segurança dos nós fosse comprometida e como a segurança dos nós foi pensada para ser provisionada pelos usuários do sistema, existiria uma vulnerabilidade caso ela fosse negligenciada.

A solução foi utilizar o Bigchain como camada intermediária, por ser de fácil integração e manipulação dos dados, mas após uma análise do sistema foi identificado que sua versão 1.3.0 possui um processo semelhante a de replicação de *cluster*, pois possui apenas um banco de dados lógico e expô-lo a vários nós traria o mesmo problema de vulnerabilidade, porém a partir da versão 2.0.0 cada nó possui um *cluster* mongoDB isolado e a conexão dos nós seria por meio do protocolo Tendermint, no entanto, após essa atualização do *Bigchain* para a versão v2.0.0, o protocolo Tendermint sofreu várias mudanças e melhorias que não foram acompanhadas pelo Bigchain.

3.1.1.2 Hyperledger Sawtooth

Dentre os blockchain analisados, os que melhor se adequaram aos requisitos de maneira geral foram o Hyperledger Fabric e o Hyperledger Sawtooth, porém devido as camadas de permissionamento e criação de entidades separadas dentro do blockchain Hyperledger Fabric, foi decido pela escolha do Hyperledger Sawtooth como o blockchain mais adequado para a aplicação, devido a sua simplicidade e a criação de redes não permissionadas.

Para a escolha final, entre o Sawtooth e o Bigchain, foi decidido pelo Sawtooth, por ser uma solução distribuída além de descentralizada e pelo apoio da comunidade.

3.2 PROJETO DO SOFTWARE

Para solucionar os problemas propostos de forma completa foi necessário dividir o projeto em dois grupos, o de infraestrutura e o de aplicações.

Para a infraestrutura, o objetivo foi criar uma infraestrutura em *blockchain* para um ambiente de produção, atuando desde a configuração dos domínios até a alta disponibilidade do servidor em nuvem.

Para o grupo de aplicações, foi proposta a criação de 3 programas, conforme abaixo:

1. Programa de interface com o usuário Final (Enviar os dados para o programa Cliente), chamado de Interface.
2. Programa Cliente para a comunicação com a REST API do blockchain, chamado de Cliente.
3. Programa Processador de transações da Família `pnrd_net`, chamado de Processador de Transações.

Um demonstrativo de todos os sistemas conectados é exibido na Figura 15 abaixo:

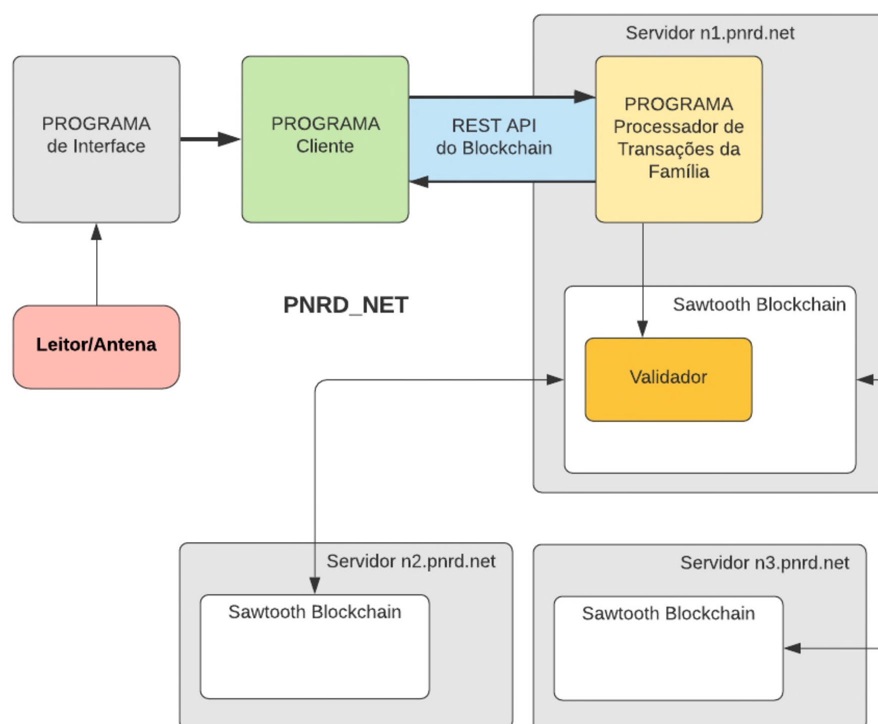

Figura 15 – Esquema de funcionamento dos sistema criados

3.3 GRUPO DE INFRAESTRUTURA

As configurações de infraestrutura para o blockchain Sawtooth seguem a documentação para sistemas não permissionados e que utilizem o mecanismo de consenso PoET (Prova por tempo de decorrido) da intel. Para entender aos requisitos desse tipo de sistema foram necessários 3 servidores Linux Ubuntu 18.04 LTS AMD64 em nuvem. Cada servidor com seu próprio subdomínio, para facilitar a manutenção do sistemas foi definido o seguinte padrão para subdomínios. `Node1 = n1.pnrd.net`, `Node2 = n2.pnrd.net` e `Node3 = n3.pnrd.net`. Todos os subdomínio possuem as mesmas configurações e seguem o mesmo passo-a-passo definido no Apêndice A.

O Node1 por ser o criador do bloco Gênesis possui outras configurações, portanto, é o mais importante dos nós. Além de ser através dele que a *API REST* será consumida, mas vale ressaltar que essa importância é apenas subjetiva, visto que todos possuem as mesmas capacidades e podem facilmente serem os substitutos do Node1.

O Fluxo básico de configuração do Node1 é o seguinte:

1. Criação do servidor (foi utilizada instâncias EC2 da AWS em todos os nós).

2. Roteamento do subdomínio n1.pnrd.net para o endereço do servidor criado.

3. Instalação de pacotes básicos de configuração e atualização do servidor, como bibliotecas etc.
4. Instalação do Sawtooth
5. Instalação dos mecanismos de consenso, motores do mecanismo de consenso e famílias (No caso o PoET)
6. Criação de uma chave privada para ser possível criar o bloco gênesis e se conetar com o blockchain
7. Criação de um chave para o Validador da rede. É atraves do validador que os blocos serão inseridos na rede.
8. Criação do bloco Gênesis e configuração do mecanismo de consenso.
9. Criação da proposta do mecanismo de consenso.
10. Unificação da proposta com o bloco Gênesis .
11. Configuração do validador (Portas, Ips e a forma de processamento de transações).
12. Configuração da *API REST* (ip do sistema, porta e etc).
13. Configuração de um servidor web para utilizar a *API REST* por meio de um *proxy* reverso.
14. Configuração do certificado SSL para a segurança da *API REST*
15. Abertura das portas de acesso da API e nó validador no servidor (No Caso no grupo de controle por ser uma instância EC2 AWS)
16. Testes de conexão em ambiente externo ao servidor (Internet)

Após ter o Node1 funcionando, foi realizada a configuração dos outros nós, Node2 e Node3. Todos seguiram basicamente os mesmo passos, com exceção da criação do bloco gênesis e do mecanismo de consenso. Mas para que esses nós se integrem à rede é necessário fornecer os *ip* e porta do validador do Node1 bem com as chaves privadas e públicas da rede.Sendo o necessário para a integração e replicação dos nós.

Após todas essas etapas a rede passa a contar com 3 nós, disponíveis globalmente e a permissão para adicionar qualquer outro nó. Uma observação importante é que o protocolo utilizado é *PoET* de forma simulada e por isso não é tolerante a falhas bizantinas. Uma alternativa seria tornar a rede provisionada alterando o mecanismo de consenso ou configurar um *PoET* real, como Intel SGX.

3.4 GRUPO APLICAÇÕES

Como foi mencionado anteriormente, para que o *blockchain* Hyperledger Sawtooth receba as transações para serem aprovadas no bloco (próximo bloco da cadeia de blocos) é necessário que exista uma família de transação para a validação e processamento dos dados, essa família é criada seguindo o formato *Protobuf*. Uma forma declarativa de dados fortemente tipada. Seguindo a documentação da linguagem, os *Protobuffers* são um mecanismo extensível para serialização de dados, sendo linguagem-neutra e plataforma-neutra, ou seja se aplicam a qualquer linguagem ou plataforma. Os códigos criados utilizando *protobuf* podem ser compilados para várias linguagens de programação.

3.4.1 FAMÍLIA DE TRANSAÇÕES PNRD NET

A família de transação PNRD NET é composta por duas entidades, os Owner (proprietários) e os Records (gravações). Os Owner são uma estrutura que permite criar novos proprietários de *tags* PNRD, sendo que cada *tag* criada é atrelada a um proprietário.

```
1  syntax = "proto3";
2
3
4  message Owner {
5      // The agent's unique public key
6      string public_key = 1;
7
8      // A human-readable name identifying the reader
9      string name = 2;
10
11     // Approximately when the reader was registered, as a Unix UTC
       timestamp
12     uint64 timestamp = 3;
13 }
14
15
16 message OwnerContainer {
17     repeated Owner entries = 1;
18 }
```

Código 3.1 – Entidade Owner da Família PNRD NET

Já entidade Record é a estruturada de dados da PNRD no blockchain, ela possui a definição, dos vetores de marcação (tokens), matriz de incidência, leitores, proprietário da tag e o status do disparo ocorrido, sendo os campos provenientes da PNRD inseridos em uma lista chamada history, incrementada a cada novo disparo na respec-

tiva TAG. Um aspecto importante na definição das entidades é que elas obedescem uma ordem fixa, isso possibilita por exemplo a atualização da família e desativação de um campo. Elas também são previamente tipadas, isso possibiliza a validação dos dados antes de chegarem ao processador da transação. Ela é demonstrada abaixo em protobuf.

```
1  syntax = "proto3";
2
3
4  message Record {
5      message Owner {
6          // Public key of the owner who owns the record
7          string owner_id = 1;
8
9          // Approximately when the owner was updated, as a Unix UTC
   timestamp
10         uint64 timestamp = 2;
11     }
12
13     message History {
14         // Firing process specifications
15         string reader_id = 1;
16         string ant_id = 2;
17         string situation = 3;
18         int32 places = 4;
19         int32 transitions = 5;
20         repeated sint32 token = 6 [packed=true];
21         repeated sint32 incidenceMatrix = 7 [packed=true];
22         // Approximately when the record is updated, as a Unix UTC
   timestamp
23         uint64 timestamp = 8;
24     }
25
26     // The user-defined natural key which identifies the object in the
27     // real world (for example a serial number)
28     string record_id = 1;
29     string tag_id = 2;
30
31     // Ordered oldest to newest by timestamp
32     repeated Owner owners = 3;
33     repeated History history = 4;
34 }
35
36
37 message RecordContainer {
38     repeated Record entries = 1;
```

```
39 }
```

Código 3.2 – Entidade Record da Família PNRD NET

Além das entidades também foi necessário definir quais ações a família irá executar, somente as ações definidas no protobuffer Payload são permitidas para a inserção de dados. Qualquer outra ação, provoca uma excessão. As ações foram definidas como Criar Proprietário, Criar Registro, Atualizar Registro e Transferir Propriedade do Registo conforme codigo a seguir.

```
1  syntax = "proto3";
2
3
4  message PnrdPayload{
5      enum Action {
6          CREATE_OWNER = 0;
7          CREATE_RECORD = 1;
8          UPDATE_RECORD = 2;
9          TRANSFER_RECORD = 3;
10     }
11
12     // Whether the payload contains a create reader, create record,
13     // update record, or transfer record action
14     Action action = 1;
15
16     // The transaction handler will read from just one of these fields
17     // according to the action
18     CreateOwnerAction create_owner = 2;
19     CreateRecordAction create_record = 3;
20     UpdateRecordAction update_record = 4;
21     TransferRecordAction transfer_record = 5;
22
23     // Approximately when transaction was submitted, as a Unix UTC
       timestamp
24     uint64 timestamp = 6;
25 }
26
27
28 message CreateOwnerAction {
29     // A human-readable name identifying the owner
30     string name = 1;
31 }
32
33
34 message CreateRecordAction {
35     // The user-defined natural key which identifies the object in the
36     // real world (for example a serial number)
37     string record_id = 1;
```

```
38      string tag_id = 2;
39
40      string reader_id = 3;
41      string ant_id = 4;
42      string situation = 5;
43      int32 places = 6;
44      int32 transitions = 7;
45      repeated sint32 token = 8 [packed=true];
46      repeated sint32 incidenceMatrix = 9 [packed=true];
47  }
48
49
50  message UpdateRecordAction {
51      // The id of the record being updated
52      string record_id = 1;
53
54      string reader_id = 2;
55      string ant_id = 3;
56      string situation = 4;
57      int32 places = 5;
58      int32 transitions = 6;
59      repeated sint32 token = 7 [packed=true];
60      repeated sint32 incidenceMatrix = 8 [packed=true];
61
62  }
63
64
65  message TransferRecordAction {
66      // The id of the record for the ownership transfer
67      string record_id = 1;
68
69      // The public key of the owner to which the record will be transferred
70      string receiving_owner = 2;
71  }
```

Código 3.3 – Payload da Família PNRD NET

Com isso, o modelo de dados está completo, permitindo a execução de ações que irão salvar os dados definidos através das entidades da família.

3.4.2 PROGRAMA DE INTERFACE

Para o programa de interface foi pensado em um painel de configuração dos leitores e leitura dos dados em tempo real, bem como o envio de cada leitura da *tag* para o programa cliente conforme Figura 16 abaixo:

Esse sistema supervisor, foi desenvolvido por (PAULA, 2019) com o objetivo de

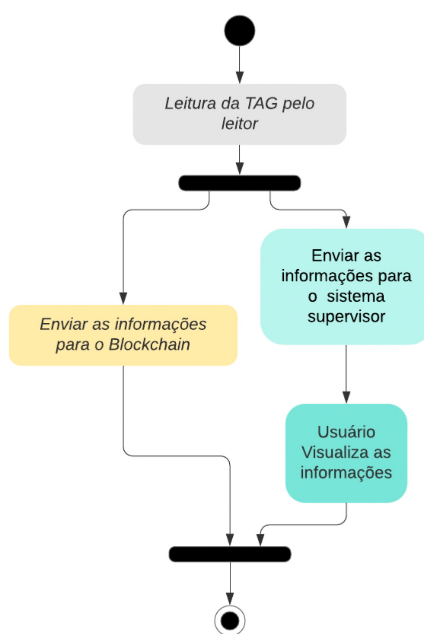

Figura 16 – Programa de interface com o programa Cliente

converter arquivos PNML que seriam representações gráficas de redes de petri em formato XML para arquivos de configuração da PNRD e leitura de *tags* em tempo real, porém para atender a necessidade de enviar dados para o programa cliente, foi feita a modificação do programa da seguinte forma:

1. Criação de uma interface web para envios dos dados recolhidos.
2. Criação de processamento paralelo para não comprometer o fluxo de atualização do sistema supervisor.
3. Melhorias no processo de leitura de arquivos PNML possibilitando a leitura de arquivos criados pelo software RENEW.

O programa possui 2 modos de funcionamento, conforme Figura 17, um de configurações, onde os arquivos PNML são convertidos em uma matriz de incidência transposta e vetor de marcação inicial, sendo posteriormente salvos como arquivos de configurações do Arduino de acordo com a quantidade leitores. Já no modo de execução o programa monitora as portas seriais previamente configuradas para receber as requisições dos leitores decorrente da leitura das Tags.

3.4.3 PROGRAMA CLIENTE

É o responsável pela comunicação da Interface com o Processador de Transações, é nele que ocorre a validação dos dados iniciais, a criação de chaves privadas

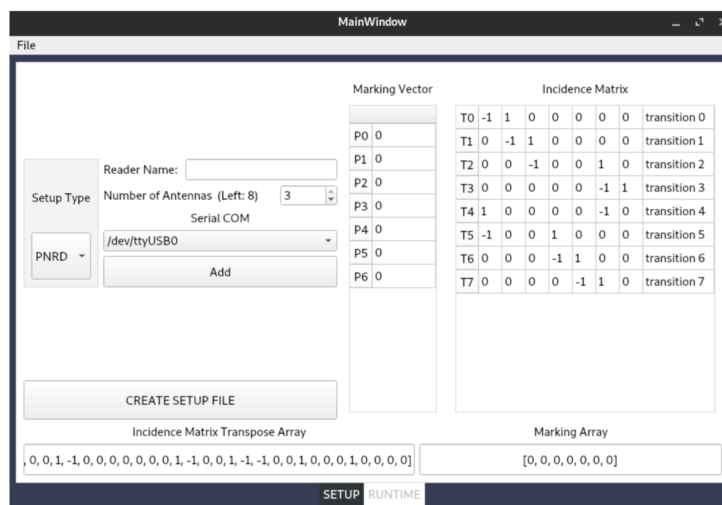

Figura 17 – Programa de interface (PALMS)

e públicas, a criação das transações, inclusão de cabeçalhos, criação de lotes, serialização dos dados e envio de solicitação ao Processador de Transações.Um esquema simplificado do funcionamento do programa é mostrado abaixo (Figura 18):

O Programa cliente foi criado utilizando o framework Flask e sua estrutura é composta por rotas, que em outras palavras são *endpoints* seguindo o Padrão REST API. Cada rota é responsável por manipular um determinado conjunto de dados. Por exemplo a rota `POST: createOwner -> /owner/create` (Figura 19) faz um pedido para criar um novo proprietário, quando a solicitação é atendida ocorre a criação de uma chave privada e pública para o solicitante e uma transação conforme dados inseridos, essa transação é então encapsulada em um lote, serializada e enviada ao Processador da transação. Lembrando que ela só é anexada ao bloco caso todos os processos de validação ocorram com sucesso. Assim que ela é enviada, seu estado é dado com `PENDING` e após a confirmação no bloco é atualizada para `COMMITED`

De forma similar foram definidas rotas para todas as ações na FAMÍLIA DE TRANSAÇÕES PNRD NET conforme o código abaixo, utilizando o pacote `curl`

```
curl --location --request POST '/owner/create' \
--data-raw '{
    "name": "Nikola Tesla"
}'

curl --location --request POST '/record/transfer' \
--data-raw '{
    "record_id": "2450",
    "receiving_owner_pubkey": "037b20d0922a...",
    "private_key": "26c6fd8914b0e85a4975be5...."
}'

```

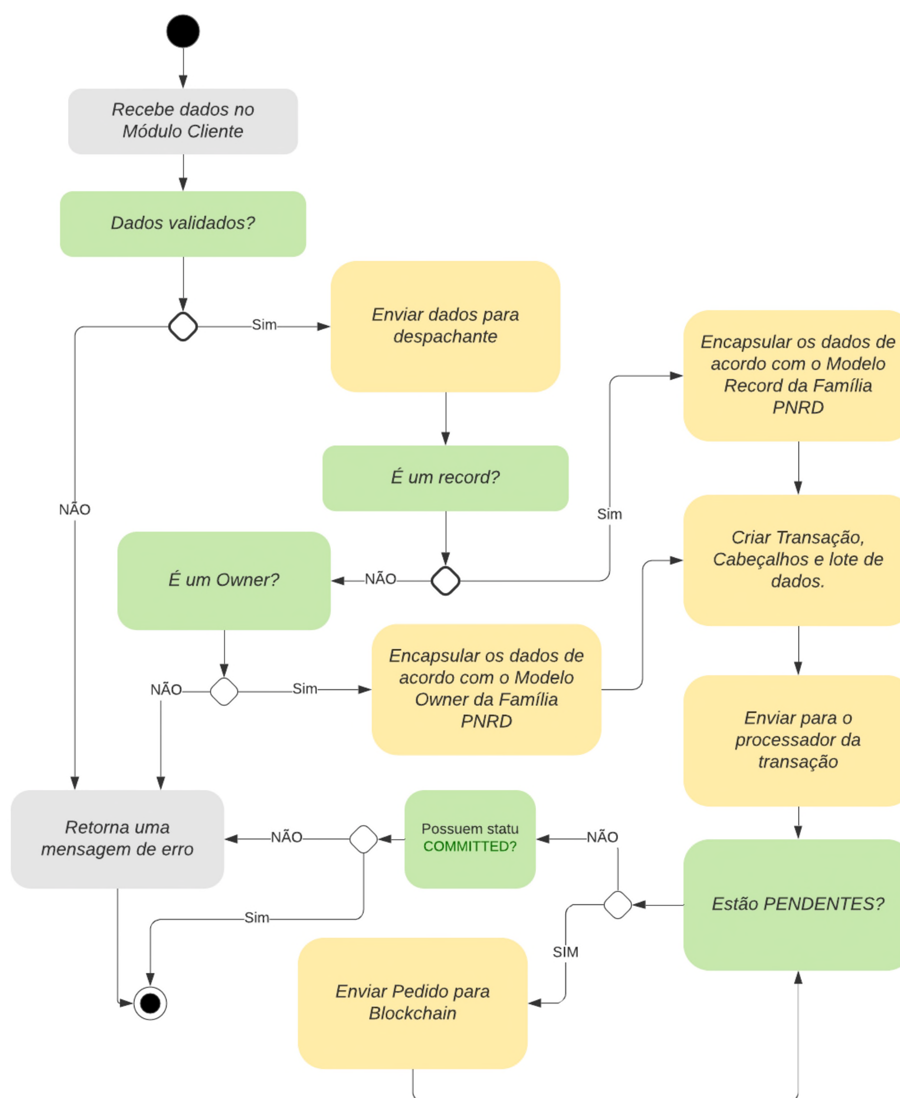

Figura 18 – Diagrama representando o funcionamento do Programa Cliente

POST **createOwner-> /owner/create**

/owner/create

BODY raw

```
{
    "name":"Roger Carrijo"
}
```

Figura 19 – Rota para a criação de um proprietário

```
13 curl --location --request POST '/record/create' \
14 --data-raw '{
15     "private_key":"26c6fd8914b0e85a499860e...",
16     "record_id": "2461",
```

```
17     "reader_id": "3",
18     "ant_id": "10",
19     "situation": "NO_ERROR",
20     "places":6,
21     "transitions":5,
22     "token":[1,0,0,0,0,0],
23     "incidenceMatrix":[0,0,-1,0,0,0,1,1,...,0,,0,1,0,0],
24     "tag_id": "ANR-546"
25 }'
26
27
28
29 curl --location --request POST '/record/update' \
30 --data-raw '{
31     "private_key":"26c6fd8914b0e85a499860e...",
32     "record_id": "2461",
33     "reader_id": "3",
34     "ant_id": "10",
35     "situation": "NO_ERROR",
36     "places":6,
37     "transitions":5,
38     "token":[1,0,0,0,0,0],
39     "incidenceMatrix":[0,0,-1,0,0,0,1,1,...,0,,0,1,0,0]
40 }'
```

Código 3.4 – Endpoints de ação da família PNRD NET

Também foram criadas rotas complementares para o obtenção de dados (Sem inserir transações) Conforme abaixo:

```
1 curl --location --request POST '/owner/detail' \
2 --data-raw '{
3     "public_key":"037
    b20d0922a419f98d30396dad78d3bb827d8fc4262fdb8c67f08295e8876a6f9"
4 }'
5
6 curl --location --request GET '/core/network'
7
8
9 curl --location --request POST '/record/detail' \
10 --data-raw '{
11         "record_id": "e2a223d9"
12 }'
```

Código 3.5 – Endpoints complementares da Família PNRD NET

3.4.3.1 Testes

Para exemplificar o funcionamento desses programas, foi criada uma transação responsável por inserir uma nova *tag* no *blockchain*. Porém antes de fazer uma requisição para criar uma gravação, primeiro é necessário criar um Owner e para isso foi executado o seguinte comando, conforme imagem a seguir:

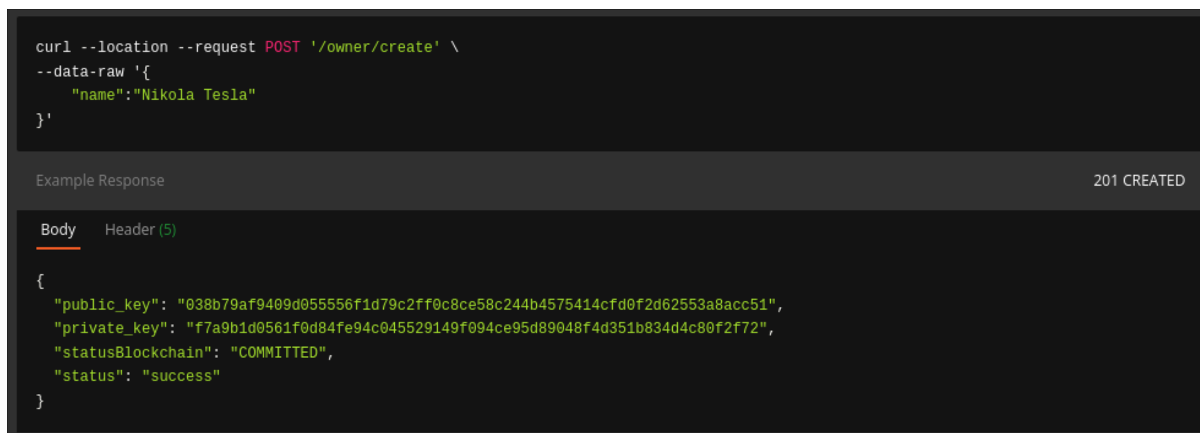

Figura 20 – Criação do proprietário Nikola Tesla

É possível perceber no retorno da solicitação o status `COMMITED` informando que os dados foram inseridos no *blockchain*, conforme requisitos funcionais, os dados publicados no banco de dados devem ser públicos, portanto a transação realizada pode ser acessada por meio do endereço (devido ao tamanho o restante do endereço foi ocultado):

`https://n1.pnrd.net/sawtooth/batch_statuses?id=7ffac92a2107....`

Ao consultar esse endereço foi obtido:

```
1  {
2      "data": [
3          {
4              "id": "7ffe9689c1fc7c0266010b8ac92a2...",
5              "invalid_transactions": [],
6              "status": "COMMITTED"
7          }
8      ],
9      "link": "https://n1.pnrd.net/sawto..."
10 }
```

Código 3.6 – Consulta de Lote ao criar o proprietário Nikola

Com o proprietário Nikola criado é possível criar um disparo simulado com o status `NO_ERROR`, para isso vamos utilizar o *endpoint*: `/record/create` com o verbo HTTP POST com o seguinte body:

```
{
    "private_key":"f7a9b1d0561f0d84fe94c045529149f094ce95d89048
  f4d351b834d4c80f2f72",
    "record_id": "2461",
    "reader_id": "3",
    "ant_id": "10",
    "situation": "NO_ERROR",
    "places":6,
    "transitions":5,
    "token":[1,0,0,0,0,0],
    "incidenceMatrix":[0,0,-1,0,0,0,1,1,-1,0,0,0,0,1,-1,-1,0,0,
  0,1,1,-1,0,0,1,-1,0,1,0,0],
    "tag_id": "ANR-546"
}
```

Código 3.7 – Body para criar um novo registro

Pode-se notar que o *private key* é o mesmo obtido ao criar o *owner* Nikola e necessário para atribuir a *tag* ao seu devido proprietário. Ao executar o comando foi obtido o seguinte resultado (Figura 21): O status `COMMITED` indica que o registro foi gravado

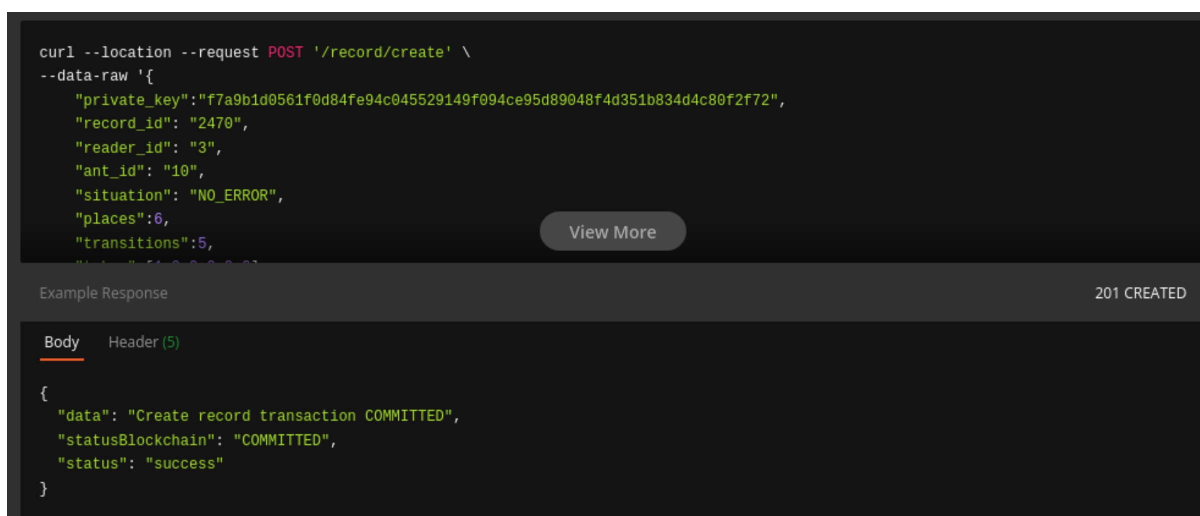

Figura 21 – Criação da gravação (Record)

com sucesso. Consultado o id do lote que deu origem a essa transação foram obtidos os seguintes dados vindo do *blockchain*:

```
{
```

```
2     "data": {
3         "header": {
4             "signer_public_key": "02cd4192b7018574d1443e1baca82
      839b0c0ad36acf0b88e428b33eeaa0e254796",
5             "transaction_ids": [
6                 "bfab5486375167e0dec8f1a21d24f702ec7c2661115548
      863a0ac102f4781b4d3df45916811ecf503e1038b01b86169d18cb12d5cf
      29f64d3ddfef30eb877cf1"
7             ]
8         },
9         "header_signature": "b3255eba2db36b9f6be2e3d36e869ea03
      dedf93a0ac65f2b422ecb2acbd3c24f7f8f43ef29f035be2bcaf85525322
      8a92afaf5dd94213f7fe0eff8b96196a4f0",
10        "trace": false,
11        "transactions": [
12            {
13                "header": {
14                    "batcher_public_key": "02cd4192b7018574d144
      3e1baca82839b0c0ad36acf0b88e428b33eeaa0e254796",
15                    "dependencies": [],
16                    "family_name": "pnrd_net",
17                    "family_version": "0.1",
18                    "inputs": [
19                        "d451ac003c415e893a4e94e8e4152ffa934a68
      e06bcdb3a6569d7f1ba2f29078744a66",
20                        "d451ac0165455e82449cb18531d35ae450c17
      fb3e81fb3b9dc1e72810a1359b99fdd58"
21                    ],
22                    "nonce": "",
23                    "outputs": [
24                        "d451ac0165455e82449cb18531d35ae450c17
      fb3e81fb3b9dc1e72810a1359b99fdd58"
25                    ],
26                    "payload_sha512": "dd0ff051aa5bf50dad43491c
      361081a1590f5e7a0df4365cbfa6a7f7545fbaa2bff80c675115808344
      dcfb4568b34af1de8f8fe967b9faf0b59825dd89e0ffc4",
27                    "signer_public_key": "038b79af9409d055556f1
      d79c2ff0c8ce58c244b4575414cfd0f2d62553a8acc51"
28                },
```

```
                "header_signature": "bfab5486375167e0dec8f1a21d24f702ec7c26611155488863a0ac102f4781b4d3df45916811ecf503e1038b01b86169d18cb12d5cf29f64d3ddfef30eb877cf1",
                "payload": "CAEaTAoEMjQ3MBIHQU5SLTU0NhoBMyICMTAqCE5PX0VSUk9SMAY4BUIGAgAAAAAASh4AAAEAAAACAgEAAAAAAgEBAAAAAgIBAAACAQACAAAwh0zaiwY="
            }
        ]
    },
    "link": "https://n1.pnrd.net/sawtooth/batches/b3255eba2db36b9f6be2e3d36e869ea03dedf93a0ac65f2b422ecb2acbd3c24f7f8f43ef29f035be2bcaf855253228a92afaf5dd94213f7fe0eff8b96196a4f0"
}
```

Código 3.8 – Body para criar um novo registro

Com o *blockchain* público, todas as transações podem ser consultadas pelo endereço `https://n1.pnrd.net/sawtooth/blocks` onde n1 representa 1 de 3 nós disponíveis na rede.

Conforme requisito funcional, era necessário que, além do registro público dos dados, fosse possível obter as informações do registo, isso é alcançado consultado os *endpoint* complementares definidos anteriormente, no caso é necessário acessar a rota `/record/detail` com o seguinte *body*:

```
{
    "record_id": "2470"
}
```

Código 3.9 – Body para a busca de uma tag específica

Com isso foi obtido o seguinte resultado:

```
{
    "address": "d451ac0165455e82449cb18531d35ae450c17fb3e81fb3b9dc1e72810a1359b99fdd58",
    "data": {
        "tag_id": "ANR-546",
        "record_id": "2470",
        "owners": [
            {
                "owner_id": "038b79af9409d055556f1d79c2ff0c8ce58c244b4575414cfd0f2d62553a8acc51",
```

```
                "timestamp": 1635169796
            }
        ],
        "history": [
            {
                "reader_id": "3",
                "ant_id": "10",
                "situation": "NO_ERROR",
                "places": 6,
                "transitions": 5,
                "incidenceMatrix": [
                    0,
                    0,
                    -1,
                    0,
                    0,
                    0,
                    1,
                    1,
                    -1,
                    0,
                    0,
                    0,
                    0,
                    1,
                    -1,
                    -1,
                    0,
                    0,
                    0,
                    1,
                    1,
                    -1,
                    0,
                    0,
                    1,
                    -1,
                    0,
                    1,
```

```
48                     0,
49                     0
50                 ],
51                 "token": [
52                     1,
53                     0,
54                     0,
55                     0,
56                     0,
57                     0
58                 ]
59             }
60         ]
61     },
62     "status": "success"
63 }
```

Código 3.10 – Busca de uma tag específica

Após a consulta dos *endpoints* percebeu-se que o Programa Cliente funciona como o esperado, atendendo aos requisitos de disponibilidade global dos dados, busca do histórico e informações das *tags*, porém vale ressaltar que não foram mostrados nesse teste as análises de todos os *endpoints*, mas todos eles foram verificados seguindo a mesma metodologia do exemplo acima.

3.4.4 PROGRAMA PROCESSADOR DE TRANSAÇÕES

O programa processador de transações tem como base a família de transação `PNRD_NET`, ele tem o objetivo de validar os dados antes de serem enviados ao bloco do *blockchain*, ele também servem de ponte para a *REST API*, pois só é permitido o envio de lotes para famílias que estão disponíveis ao Validador.

Conforme informações no modelo de arquitetura, o Validador escuta os pedidos de processadores de transações de famílias por meio de um endereço pré-definido, geralmente criando no início de vida do *blockchain*. No caso da `PNRD_NET` o endereço do validador é definido por `tcp://localhost:4004`. Como medida de segurança o validador do *blockchain* não pode ser exposto publicamente, pois isso permitiria que qualquer família externa ao *blockchain* pudesse solicitar entrada como Processadora de transações. Por essa razão, tanto o Validador quando o Processador de Transações se encontram no mesmo servidor, no caso o `n1.pnrd.net`

3.4.4.1 Classes para o Processamento de Transações

O Processador `PNRD_NET` possui apenas 3 classes principais:

1. PnrdNetHandler, filha da classe TransactionHandler e lida com a atualização de estados
2. PnrdNetPayload, responsável por mapear as ações da família. (Criar Proprietário, Criar Registro, Atualizar Registro e Transferir Propriedade)
3. PnrdNetState, um conjunto de funções que estruturam como os dados devem ser inseridos, seguindo as entidades Owner e Record

O código completo do programa processador encontra-se em anexo. Devido ao seu mecanismo de funcionamento (Por linha de comando), a exemplificação de seu funcionamento de maneira visual é complexa, porém sua execução de forma esperada pôde ser vista quando enviamos requisições de criação do proprietário Nikola e criação da tag, pois essas transações não seriam possíveis caso o Processador de transações não estivesse em pleno funcionamento.

Um aspecto importante do Processador é como ele lida com as informações passadas pelo Executor(Elemento do Blockchain Sawtooth), sendo recebidas pelo TransactionHandler com método `Apply`

3.4.4.2 Método Apply

Toda classe processadora de transações é filha da classe TransactionHandler, chamada também de classe Manipuladora, ela possui 2 argumentos, o `context` e o `transaction`.

O argumento `transaction` é uma instância da classe Transaction (transação) que é criada a partir dos *protobufs* (Família de Transação). Ele contém o *payload* em bytes e as especificações da família da transação, sendo decodificado utilizando uma classe de descodificação que depende da aplicação, no nosso caso a PnrdNetPayload, de forma resumida ele contém o conteúdo e as ações da aplicação.

Já o `context` é uma instância da classe Context vinda do SDK (No caso do projeto o SKD Python).Ele contém as informações a respeito da estrutura da aplicação, como estados e elementos. Por convenção a classe da aplicação que recebe o Context é chamada de estado da aplicação no caso do projeto ela foi chamada de PnrdNetState.

3.4.5 TESTE FUNCIONAL

Para realizar os testes funcionais, foi pensando pelo ponto de vista do usuário final como o sistema deveria funcionar, então foi adotada a seguinte abordagem:

1. Modelar um problema utilizando a forma gráfica das redes de petri
2. Salvar a modelo em arquivo pnml utilizando o software Renew.
3. Enviar o arquivo PNML para o Programa de interface, com o objetivo de mapear e configurar todos as placas e leitores envolvidos no processo.
4. Com o arquivo gerado pelo Programa de interface, realizar a configuração inicial de todas as tags e leitores.
5. Conectar os leitores configurados em suas respectivas portas seriais.
6. Efetua a leitura das Tags
7. Conferir o resultado do envio bem sucedido das informações no Programa Cliente.

3.4.5.1 Modelagem da Rede de Petri

Para a modelagem foi utilizado o mesmo modelo da Figura 7 um esquema de teste de peça usinada. Segue abaixo a modelagem da estrutura em redes de petri utilizando o programa RENEW. Foi criado também o seu respectivo arquivo pnml e

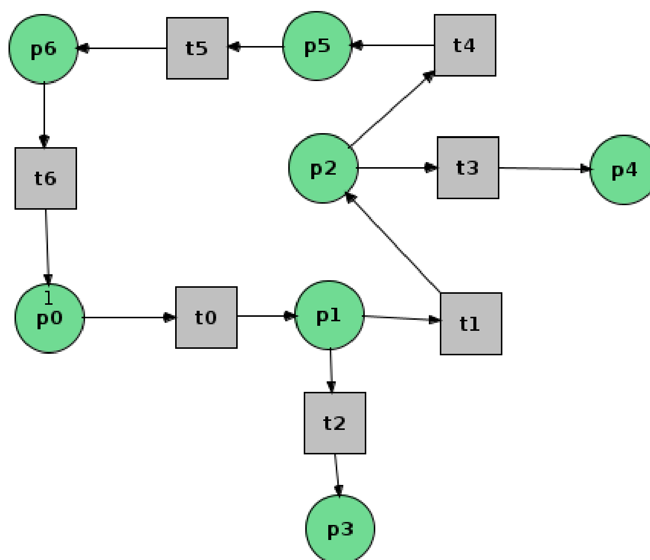

Figura 22 – Representação em redes de petri de um esquema de teste de peça usinada

utilizando o comando Ctrl+O dentro do programa de interface foi possível abrí-lo, o resultado é mostrado abaixo (Figura 23).

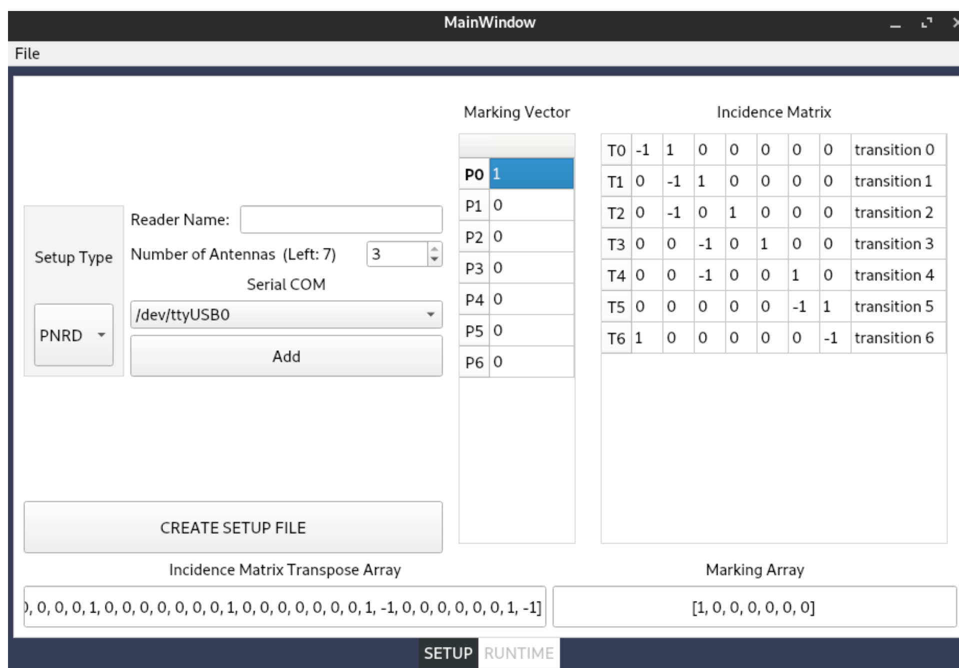

Figura 23 – Representação em redes de petri de uma máquina de teste de usinagem no Programa de Interface

Também foram configurados todos os leitores em suas respectivas portas e antenas, o resultado foi a criação de um arquivo *.palms que representa todas as configurações inseridas. Também são criados arquivos em Arduino de acordo com a quantidade de leitores, de forma automática pelo programa de Interface, tudo isso inserido em uma pasta chamada palmsSetup localizada no mesmo endereço onde se encontra o arquivo PNML.

A configuração das placas de Arduino (Figura 24) foi a seguinte:

- Leitor0 com 1 antena para o disparto de t0
- Leitor1 com 1 antena para o disparo de t1
- Leitor2 com 3 antenas para o disparo de t2, t3, t4
- Leitor3 com 2 antenas para o disparo de t5, t6

```
1 {
2     "pnmlFile": "/home/roger/Documentos/TCC/pnrd-net/docs/tcc/
   code/TESTE_UNITARIO2.pnml",
3     "qtdReaders": 4,
4     "readerListConfig": [
5         {
6             "qtdAntenna": 1,
```

```
7                "readerName": "Leitor0",
8                "serialPort": "/dev/ttyUSB0"
9            },
10           {
11               "qtdAntenna": 1,
12               "readerName": "Leitor1",
13               "serialPort": "/dev/ttyS3"
14           },
15           {
16               "qtdAntenna": 3,
17               "readerName": "Leitor2",
18               "serialPort": "/dev/ttyS2"
19           },
20           {
21               "qtdAntenna": 2,
22               "readerName": "Leitor3",
23               "serialPort": "/dev/ttyS1"
24           }
25       ],
26       "transitionNames": [
27           "transition 0",
28           "transition 1",
29           "transition 2",
30           "transition 3",
31           "transition 4",
32           "transition 5",
33           "transition 6"
34       ],
35       "type": "PNRD"
36   }
```

Código 3.11 – Arquivo de configuração *.palms para teste funcional

Para iniciar o modo em tempo real foi necessário configurar pelo menos 1 leitor, responsável pelo disparo em t0 (Figura 25).

Para a integração do programa de Interface com o programa Cliente foi necessário realizar a inclusão de 2 arquivos na pasta palmsSetup, um para autorizar a programa de Interface a realizar transações em nome do proprietário e outro para definir qual o endereço completo de acesso ao programa cliente.

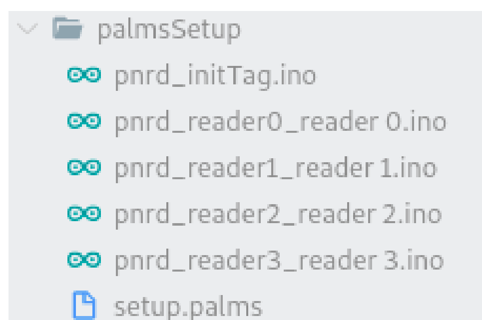

Figura 24 – Pasta de arquivos gerados para a configuração do teste funcional

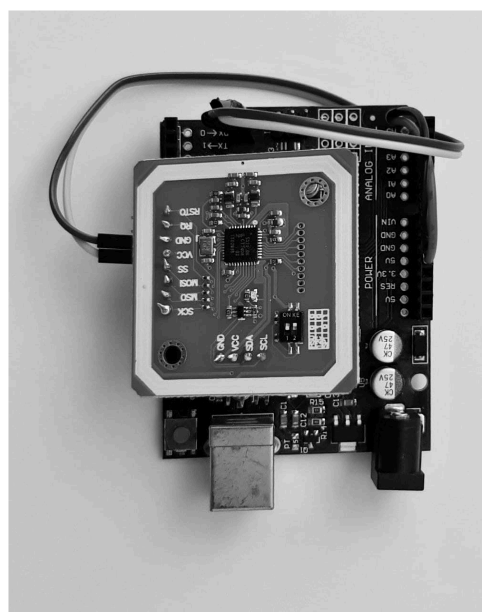

Figura 25 – Arduino com placa NFC para leitura e tags

A autorização ocorre por meio da busca de um arquivo chamado secret.txt, nele existe apenas a chave privada criada ao criar o usuário Nikola.

O endereço completo para acesso ao programa cliente tem que ser especificado por meio do arquivo `rest_api_url.txt`, ele contém apenas a url base para acesso ao programa cliente. Ambos os arquivos foram configurados conforme modelos abaixo:

```
1 f7a9b1d0561f0d84fe94c045529149f094ce95d89048f4d351b834d4c80f2f7
  2
```

Código 3.12 – Arquivo secret.txt

```
1 http://localhost:5000
```

Código 3.13 – Arquivo rest api url.txt

Após essas configurações, o programa de interface foi iniciado e utilizando o comando *Open Setup file (*.palms)* a aba para o monitoramento do processo em tempo

real foi habilitada, conforme (Figura 26) a seguir. Por simplicidade foram incluídos apenas os leitores leitor0 e leitor1 nesse teste, mas todos os leitores funcionam de forma similar.

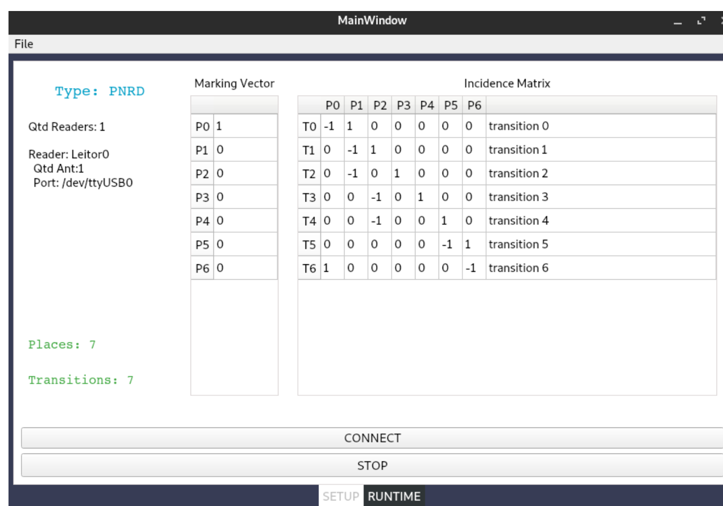

Figura 26 – Programa de interface na aba runtime

Em seguida o botão *CONNECT* foi pressionado para que o programa monitorasse a porta serial `/dev/tty/USB0` (Figura 27)

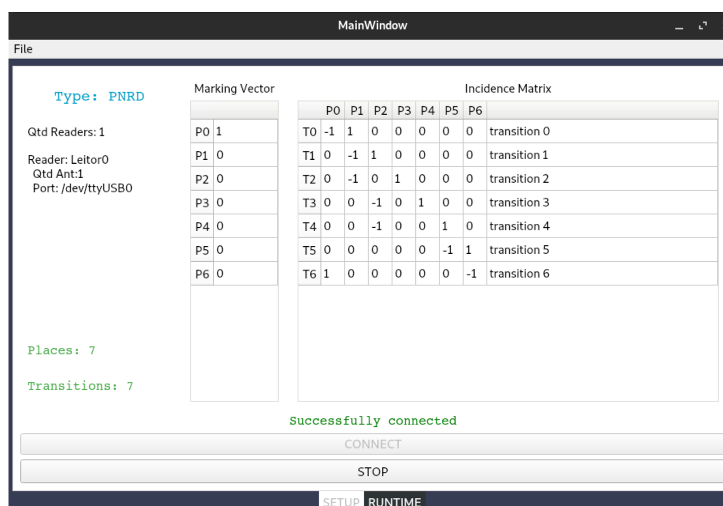

Figura 27 – Programa de interface na aba runtime - Conectado

Após ter a conexão estabelecida, a *tag* `80fa2243` foi aproximada do leitor para obter o disparo de t0 referente ao leitor0 com o status (Figura 28) que significa que nenhum erro foi encontrado.

De forma similar realizamos o disparo em t1 para a mesma tag `80fa2243` e também nenhum erro foi encontrado. (Figura 29)

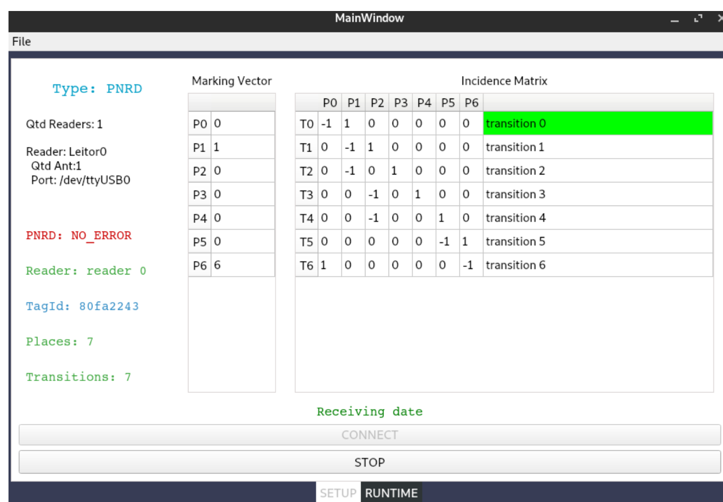

Figura 28 – Programa de interface na aba runtime - Disparo de t0

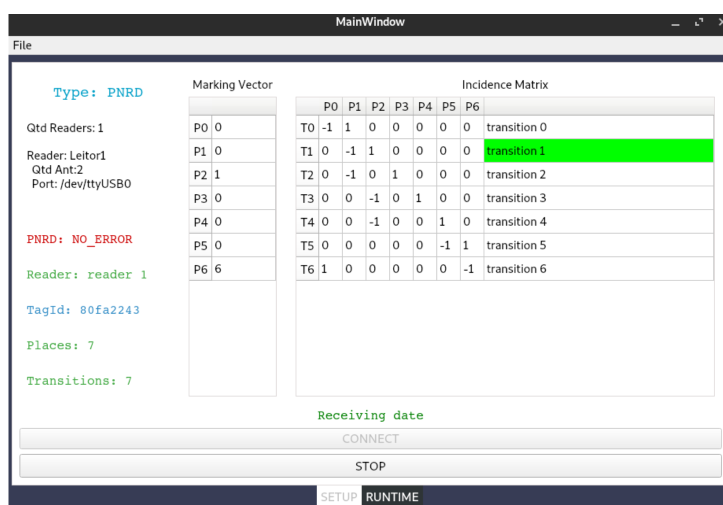

Figura 29 – Programa de interface na aba runtime - Disparo de t1

Para verificar a ocorrência de problemas com o fluxo normal da rede de petri a tag `80fa2243` foi retornada ao leitor0, onde não existe uma transição habilitada e por isso ocorre o erro da Figura 30.

Todos as transações mostradas anteriormente foram enviadas ao programa Cliente de acordo com o endereço informado no arquivo `rest_api_url.txt`, para obter esses dados é necessário realizar uma busca no endpoint complementar `/record/detail` com o body:

```
1 {
2     "record_id": "80fa2243"
3 }
```

Código 3.14 – Body para a busca de uma tag 80fa2243

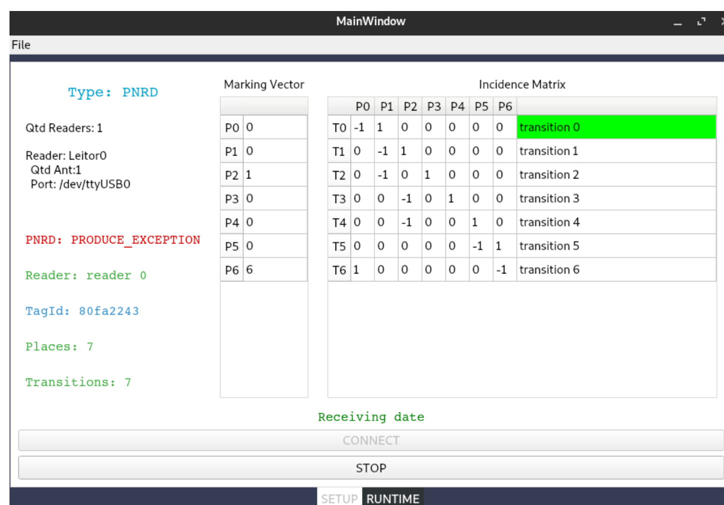

Figura 30 – Programa de interface na aba runtime - Disparo de t0

Esse id `80fa2243` é o identificador único da tag e é através dele que todos as informações da *tag* são obtidas.


```
{
    "address": "d451ac01cacde38318c2c8ab7de99a3e6096c66e10bc3a065f597645f8bb51f1ca73b0",
    "data": {
        "tag_id": "80fa2243",
        "record_id": "80fa2243",
        "owners": [
            {
                "owner_id": "038b79af9409d055556f1d79c2ff0c8ce58c244b4575414cfd0f2d62553a8acc51",
                "timestamp": 1635205982
            }
        ],
        "history": [
            {
                "reader_id": "reader 0",
                "ant_id": "0",
                "situation": "NO_ERROR",
                "places": 7,
                "transitions": 7,
                "incidenceMatrix": [ -1,0,0,0,0,0,1,1,-1,-1,0,0,0,0,0,1,0,-1,-1,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,-1,0,0,0,0,0,0,1,-1],
                "token": [0,1,0,0,0,0,6]
```

```
            },
            {
                "reader_id": "reader 1",
                "ant_id": "1",
                "situation": "NO_ERROR",
                "places": 7,
                "transitions": 7,
                "incidenceMatrix": [ -1,0,0,0,0,0,1,1,-1,-1,0,0
   ,0,0,0,1,0,-1,-1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,-
   1,0,0,0,0,0,0,1,-1
                ],
                "token": [ 0, 0, 1, 0, 0, 0, 6 ]
            },
            {
                "reader_id": "reader 0",
                "ant_id": "0",
                "situation": "PRODUCE_EXCEPTION",
                "places": 7,
                "transitions": 7,
                "incidenceMatrix": [-1,0,0,0,0,0,1,1,-1,-1,0,0,
   0,0,0,1,0,-1,-1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,-1
   ,0,0,0,0,0,0,1,-1],
                 "token": [0,0,1,0,0,0,6 ]
            }
        ]
    },
    "status": "success"
}
```

Código 3.15 – Resultado da busca pela tag 80fa2243

Consultado o endereço acima, diretamente no *blockchain (REST API)*, o seguinte resultado foi alcançado:

```
\\https://n1.pnrd.net/sawtooth/state?address=d451ac01cacde38318
   c2c8ab7de99a3e6096c66e10bc3a065f597645f8bb51f1ca73b0

{
    "data": [
    {
    "address": "d451ac01cacde38318c2c8ab7de99a3e6096c66e10bc3a0
```

```
65f597645f8bb51f1ca73b0",
 "data": "CoYDCgg4MGZhMjI...."}
 ],
 "head": "0190d0532543f2e0db895ff648821da2aa6c3cdd3b0ab0bf9
cefd0",
 "link": "https://n1.pnrd.net/sawtooth/state?head=0190d05325
43f2e0dbc6ca1b4d6864330b75db54d6eef42f1b21cdb772d7a3de3edf48
0818dce22def3f5e4107fd895ff648821da2aa6c3cdd3b0ab0bf9cefd0&
start=d451ac01cacde38318c2c8ab7de99a3e6096c66e10bc3a065f5976
45f8bb51f1ca73b0&limit=100&address=d451ac01cacde38318c2c8ab7
de99a3e6096c66e10bc3a065f597645f8bb51f1ca73b0",
 "paging": {
 "limit": null,
 "start": null
 }
 }
```


Código 3.16 – Resultado da busca pela tag 80fa2243 no *blockchain*

3.5 RESULTADOS

A criação de dois programas para a comunicação com o blockchain e a atualização do programa de interface proporcionaram a integração de um ambiente local, modelado por redes de petri com um ambiente distribuído de dados com disponibilidade global, seguro e imutável, atendendo assim aos requisitos de projeto informados:

Possuir um ambiente decentralizado. Disponibilizar os dados publicamente. Permitir que qualquer agente crie suas próprias soluções. A criação da PNRD NET inserida no blockchain Sawtooth atendeu a esses requisitos, pois o blockchain proporciona um ambiente descentralizado e não permissionado, onde qualquer entidade possa criar seu próprio registro proprietário, inserir tags ou ingressar na rede por meio de um nó.

*Obter os dados históricos a qualquer momento. Entender o fluxo da processo de um ponto de vista observador, sem interferência no processo.Filtrar informações do processo por tag.*Alcançados pela criação do endpoint `getRecordDetail-> /record/detail`

Mapear todas as tags ativas de acordo com o tempo. Identificar quais tags sofreram exceção. Por meio do `getNetworkDetail-> /core/network` foi possível obter todos os dados das rede.

Criar uma forma de transferir a propriedade de um objeto (tag) para outro agente, (venda de produto). A transfência de propriedade pode ser alcançada por meio do

endpoint `transferRecord -> /record/transfer` , porém para não entrar em conflito com a Transparência dos dados, qualquer alteração de propriedade deve manter um histórico de antigos donos.

4 CONCLUSÃO

Esse projeto apresentou o conceito de Famílias de Transações, estruturas de dados criadas em *protobuf* para a comunicação e validação de dados. Criou-se uma nova família de transações através da `PNRD_NET`,uma família para o envio de dados históricos da PNRD e criação de proprietários.

Demonstrou-se de forma prática o desempenho da rede e da família, através da infraestrutura de ponta-a-ponta composta por 3 programas, a Interface de comunicação, o Cliente e Processador de transações, realizaram-se testes funcionais para a validação dos requisitos necessário.

Provisionou-se uma arquitetura distribuída em 3 nós, essencial para o bom desempenho das aplicações e execução dos testes.

4.1 TRABALHOS FUTUROS

Visando a melhoria e escalabilidade da solução, os trabalhos futuros poderiam ser divididos em duas vertentes: Arquitetura e Aplicações.

Para a arquitetura:

- Modificar o protocolo de consenso para PoET com o SGX Intel.
- Criar um script de criação de nós para agilizar o processo de configuração.
- Pensar na REST API como sistema local, isso reduziria a carga sobre o Node1.
- Condicionar a utilização da solução por ambiente local, isso iria incentivar a criação de mais nós para o sistema, melhorando a escalabilidade, disponibilidade e velocidade de processamento.
- Desenvolver um leitor com PNRD em hardware.
- Padronizar os dados da tag PNRD.
- Desenvolver um workflow para a implementação prática da solução PNRD e `PNRD_NET` em ambiente empresarial.

Aplicações:

- Criação de uma nova entidade para mapeamento dos processos em PNRD, funcionando como um catálogo global de soluções pré-fabricadas.

- Incluir na entidade Record a informação referente a rede de petri utilizada, como por exemplo qual a rede de petri configurada inicialmente.
- Melhorar o processo de envio dos dados do programa de Interface. Adicionando avisos de envio bem sucedido dos dados.

REFERÊNCIAS

AHSAN, K.; SHAH, H.; KINGSTON, P. *RFID Applications: An Introductory and Exploratory Study*. 2010. Citado na página 18.

BASHIR, I. *Mastering Blockchain*. 3. ed. [S.l.]: Packt Publishing, 2020. Citado na página 28.

COULOURIS, G. et al. *Sistemas distribuídos - Conceitos e projeto*. 5. ed. [S.l.]: Bookman, 2013. Citado na página 12.

IGOE, T.; COLEMAN, D.; JEPSON, B. *Beginning NFC*. 1. ed. [S.l.]: O'Reilly Media, Inc, 2014. Citado 3 vezes nas páginas 5, 19 e 20.

LANTZ, L.; CAWREY, D. *Mastering Blockchain*. 1. ed. [S.l.]: O'Reilly Media, Inc., 2020. Citado 3 vezes nas páginas 5, 27 e 28.

MADUMIDHA, S. et al. Transparency and traceability: In food supply chain system using blockchain technology with internet of things. In: *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*. [S.l.: s.n.], 2019. p. 983–987. Citado na página 13.

PAULA, R. H. C. D. Sistema configuração e executação de pnrds. 2019. Citado na página 44.

SCHUH, G. et al. Hypotheses for a theory of production in the context of industrie 4.0. *Advances in Production Technology*, v. 1, p. 21–23, 2015. Citado na página 13.

SILVA, C. E. A. D. Desenvolvimento de biblioteca para aplicações de pnrd e pnrd invertida embarcadas em arduino. *Repositório Universidade Federal de Uberlândia*, p. 46–54, 2017. Citado 7 vezes nas páginas 12, 13, 20, 21, 22, 24 e 26.

TANENBAUM, A. S.; STEEN, M. V. *Sistemas distribuídos - Princípios e paradígmas*. 2. ed. [S.l.]: Person, 2013. Citado na página 12.

TAVARES, J.; SARAIVA, T. Elementary petri net inside rfid distributed database (pnrd). *International Journal of Production Research*, v. 48, p. 2563–2582, 05 2010. Citado na página 25.

TAVARES, J. J. P. Z. de S.; PHAWADE, R. Improving exception analysis in pnrd systems using labelled free choice nets. 2010. Citado na página 24.

ZELBST, D. P. J.; SOWER, D. V. E. *RFID for the Supply Chain and Operations Professional*. 3. ed. [S.l.]: Business Expert Press, 2021. Citado 3 vezes nas páginas 16, 17 e 18.

Apêndices

APÊNDICE A – INFRAESTRUTURA SAWTOOTH

Administração rede hyperledger Sawtooth

Instalação Sawtooth

Para instalar o sawtooth conforme recomendação, foi criada uma instância da AWS com Ubuntu 18.04 LTS (Bionic) para arquitetura AMD64, seguindo os seguintes passos:

1. Atualização da lista de repositório:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv-keys
8AA7AF1F1091A5FD

sudo add-apt-repository 'deb [arch=amd64]
http://repo.sawtooth.me/ubuntu/chime/stable bionic universe'

sudo apt-get update
```

2. Instalação do Sawtooth

```
sudo apt-get install -y sawtooth
```

3. Instalação do mecanismo de consenso PoET (Proof of Elapse Time)

```
sudo apt-get install -y sawtooth \
python3-sawtooth-poet-cli \
python3-sawtooth-poet-engine \
python3-sawtooth-poet-families
```

4. Visão Geral de todos os pacotes instalados

```
dpkg -l '_sawtooth_'
```

```
ubuntu@ip-172-31-30-176:~$ dpkg -l '*sawtooth*'
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name                Version        Architecture   Description
+++-===================-==============-==============-==========================================
ii  python3-sawtooth-c 1.2.6-1        all            Sawtooth CLI
ii  python3-sawtooth-i 1.2.3-1        all            Sawtooth Intkey Python Example
ii  python3-sawtooth-p 1.1.3-1        all            Sawtooth PoET CLI
ii  python3-sawtooth-p 1.1.3-1        all            Sawtooth PoET Common Modules
ii  python3-sawtooth-p 1.1.3-1        all            Sawtooth Core Consensus Module
ii  python3-sawtooth-p 1.1.3-1        all            Sawtooth PoET Consensus Engine
ii  python3-sawtooth-p 1.1.3-1        all            Sawtooth Transaction Processor Families
ii  python3-sawtooth-p 1.1.3-1        all            Sawtooth PoET Simulator Enclave
ii  python3-sawtooth-r 1.2.6-1        all            Sawtooth REST API
ii  python3-sawtooth-s 1.2.3-1        all            Sawtooth Python SDK
ii  python3-sawtooth-v 1.2.6-1        all            Sawtooth Validator
ii  python3-sawtooth-x 1.2.3-1        all            Sawtooth XO Example
ii  sawtooth           1.2.6          all            Hyperledger Sawtooth Distributed Ledger
ii  sawtooth-identity- 1.2.6          amd64          The Sawtooth Identity TP for validating o
ii  sawtooth-settings- 1.2.6          amd64          The Sawtooth Settings transaction process
```

Configuração

A configuração da rede consiste em gerar as chaves para acesso ao blockchain, definir mecanismo de consenso e criação da bloco Genesis(Inicio do blockchain)

1. Geração da chave privada

```
sawtooth keygen my_key
```

As chaves são armazenadas na pasta local do usuário em **.sawtooth/keys**

2. Geração das chaves do Validador. O Validador é um componente da rede Sawtooth responsável por permitir o que entra ou não no livro-razão (ledger)

```
sudo sawadm keygen
```

3. Criação do bloco Gênesis O Primeiro nó da rede Sawtooth é o responsável pela criação do block genesis. Todos os outros nós que ingressarem na rede utilizarão as configurações definidas nesse block.

   Somente o criador do block genesis tem permissão para alterá-lo a não ser que na configuração do block seja definido quais outros itegrantes terão esse privilégio.

```
 cd /tmp

 sawset genesis --key $HOME/.sawtooth/keys/pnrd.priv -o config-genesis.batch

 sawset proposal create --key $HOME/.sawtooth/keys/pnrd.priv \
 -o config-consensus.batch \
 sawtooth.consensus.algorithm.name=PoET \
```

```
 sawtooth.consensus.algorithm.version=0.1 \
 sawtooth.poet.report_public_key_pem="$(cat
/etc/sawtooth/simulator_rk_pub.pem)" \
 sawtooth.poet.valid_enclave_measurements=$(poet enclave measurement)
\
 sawtooth.poet.valid_enclave_basenames=$(poet enclave basename)
```

```
poet registration create --key /etc/sawtooth/keys/validator.priv -o
poet.batch

sawset proposal create --key $HOME/.sawtooth/keys/pnrd.priv \
-o poet-settings.batch \
sawtooth.poet.target_wait_time=5 \
sawtooth.poet.initial_wait_time=25 \
sawtooth.publisher.max_batches_per_block=100
```

Unificação de todos os batches para a criação do block genesis

```
sudo -u sawtooth sawadm genesis \
config-genesis.batch config-consensus.batch poet.batch poet-
settings.batch
```

Bloco genesis criado.

Configuração do Validador

AS configurações aqui definidas são chamadas de Off-chain(fora do blockchain)

Elas são importantes para que os nós consigam se comunicar e para que a Rest API e o validador funcionem de maneira correta.

Vamos começar com um modelo base e alterar as configurações necessário. No arquivo validator.toml definimos qual será a forma de busca dos nós ip/host e porta de conexão do nó fundador e chaves privadas para garantir a segurança da rede.

```
sudo cp -a /etc/sawtooth/validator.toml.example
/etc/sawtooth/validator.toml

sudo vim /etc/sawtooth/validator.toml
```

Nesse arquivo foram feitas as seguintes modificações:

```
endpoint = "tcp://n1.pnrd.net:8800"
peering = "dynamic"
seeds = ["tcp://n1.pnrd.net:8800"]
peers = ["tcp://n1.pnrd.net:8800"]
network_public_key = '<secreto>'
network_private_key = '<secreto>'
```

```
sudo chown root:sawtooth /etc/sawtooth/validator.toml
sudo chmod 640 /etc/sawtooth/validator.toml
sudo systemctl restart sawtooth-validator.service
```

Configuração da REST API

A REST API é responsável pela comunicação do cliente com o validador e por isso ela tambem foi colocada como pública seguindo as configurações a seguir:

```
sudo cp -a /etc/sawtooth/rest_api.toml.example /etc/sawtooth/rest_api.toml
sudo vi /etc/sawtooth/rest_api.toml
```

```
bind = ["127.0.0.1:8008"]
```

```
<VirtualHost *:443>
    ServerName n1.pnrd.net
    ServerAdmin rogerhcp@gmail.com
    DocumentRoot /var/www/html

    SSLEngine on
    SSLCertificateFile /etc/apache2/keys/.ssl.crt
    SSLCertificateKeyFile /etc/apache2/keys/.ssl.key
    RequestHeader set X-Forwarded-Proto "https"

    <Location />
        Options Indexes FollowSymLinks
        AllowOverride None
        AuthType Basic
        AuthName sucesso
        AuthUserFile "/etc/apache2/.htpassword"
        Require user sawtooth
        Require all denied
    </Location>
</VirtualHost>
```

```
ProxyPass /sawtooth http://localhost:8008
ProxyPassReverse /sawtooth http://localhost:8008
RequestHeader set X-Forwarded-Path "/sawtooth"
```

Certificado SSL

```
sudo apt install certbot python3-certbot-apache
sudo certbot --apache
```

Siga o passo a passo no terminal e adicione o domínio n1.pnrd.net

Online

Com isso temos um nó da rede sawtooth online configurado para o subdomínio n1.pnrd.net

Nós da rede

Para a configuração dos outros nós é necessário seguir os mesmo passos de instalação com excessão da criação do bloco gênesis. O arquivo validator deve ser copiado do nó primário com a adição dos ips específicos e inclusão do endereço em seeds e peers.

APÊNDICE B – FAMÍLIA DE TRANSAÇÕES

Owner

```
syntax = "proto3";


message Owner {
    // The agent's unique public key
    string public_key = 1;

    // A human-readable name identifying the reader
    string name = 2;

    // Approximately when the reader was registered, as a Unix UTC
timestamp
    uint64 timestamp = 3;
}


message OwnerContainer {
    repeated Owner entries = 1;
}
```

Record

```
syntax = "proto3";


message Record {
    message Owner {
        // Public key of the owner who owns the record
        string owner_id = 1;

        // Approximately when the owner was updated, as a Unix UTC
timestamp
        uint64 timestamp = 2;
    }

    message History {
        // Firing process specifications
        string reader_id = 1;
        string ant_id = 2;
        string situation = 3;
        int32 places = 4;
        int32 transitions = 5;
        repeated sint32 token = 6 [packed=true];
```

```
        repeated sint32 incidenceMatrix = 7 [packed=true];
        // Approximately when the record is updated, as a Unix UTC
timestamp
        uint64 timestamp = 8;
    }

    // The user-defined natural key which identifies the object in the
    // real world (for example a serial number)
    string record_id = 1;
    string tag_id = 2;

    // Ordered oldest to newest by timestamp
    repeated Owner owners = 3;
    repeated History history = 4;
}


message RecordContainer {
    repeated Record entries = 1;
}
```

```
syntax = "proto3";


message PnrdPayload{
    enum Action {
        CREATE_OWNER = 0;
        CREATE_RECORD = 1;
        UPDATE_RECORD = 2;
        TRANSFER_RECORD = 3;
    }

    // Whether the payload contains a create reader, create record,
    // update record, or transfer record action
    Action action = 1;

    // The transaction handler will read from just one of these fields
    // according to the action
    CreateOwnerAction create_owner = 2;
    CreateRecordAction create_record = 3;
    UpdateRecordAction update_record = 4;
    TransferRecordAction transfer_record = 5;

    // Approximately when transaction was submitted, as a Unix UTC
timestamp
    uint64 timestamp = 6;
}
```

```
message CreateOwnerAction {
    // A human-readable name identifying the owner
    string name = 1;
}


message CreateRecordAction {
    // The user-defined natural key which identifies the object in the
    // real world (for example a serial number)
    string record_id = 1;
    string tag_id = 2;

    string reader_id = 3;
    string ant_id = 4;
    string situation = 5;
    int32 places = 6;
    int32 transitions = 7;
    repeated sint32 token = 8 [packed=true];
    repeated sint32 incidenceMatrix = 9 [packed=true];
}


message UpdateRecordAction {
    // The id of the record being updated
    string record_id = 1;

    string reader_id = 2;
    string ant_id = 3;
    string situation = 4;
    int32 places = 5;
    int32 transitions = 6;
    repeated sint32 token = 7 [packed=true];
    repeated sint32 incidenceMatrix = 8 [packed=true];

}


message TransferRecordAction {
    // The id of the record for the ownership transfer
    string record_id = 1;

    // The public key of the owner to which the record will be transferred
    string receiving_owner = 2;
}
```

APÊNDICE C – PROCESSADOR DE TRANSAÇÕES

main_processor.py

```
import os
import argparse
import sys

from sawtooth_sdk.processor.core import TransactionProcessor
from sawtooth_sdk.processor.log import init_console_logging

from processor.handler import PnrdNetHandler


TOP_DIR = os.path.dirname(os.path.dirname(os.path.realpath(__file__)))
sys.path.insert(0, os.path.join(TOP_DIR, 'pnrdnet_addressing'))
sys.path.insert(0, os.path.join(TOP_DIR, 'processor'))
sys.path.insert(0, os.path.join(TOP_DIR, 'pnrdnet_protobuf'))


def parse_args(args):
    parser = argparse.ArgumentParser(
        formatter_class=argparse.RawTextHelpFormatter)

    parser.add_argument(
        '-C', '--connect',
        default='tcp://localhost:4004',
        help='Endpoint for the validator connection')

    parser.add_argument(
        '-v', '--verbose',
        action='count',
        default=0,
        help='Increase output sent to stderr')

    return parser.parse_args(args)


def main(args=None):
    if args is None:
        args = sys.argv[1:]
    opts = parse_args(args)
    processor = None
    try:
        init_console_logging(verbose_level=opts.verbose)

        processor = TransactionProcessor(url=opts.connect)
        handler = PnrdNetHandler()
        processor.add_handler(handler)
        print("Startou!")
        processor.start()
    except KeyboardInterrupt:
        pass
```

```
    except Exception as err:  # pylint: disable=broad-except
        print("Error: {}".format(err))
    finally:
        if processor is not None:
            processor.stop()


if __name__ == '__main__':
    main()
```

handler.py

```
import datetime
import time

from sawtooth_sdk.processor.handler import TransactionHandler
from sawtooth_sdk.processor.exceptions import InvalidTransaction

from pnrdnet_addressing import addresser
from pnrdnet_protobuf import payload_pb2

from processor.payload import PnrdNetPayload
from processor.state import PnrdNetState


SYNC_TOLERANCE = 60 * 5


class PnrdNetHandler(TransactionHandler):

    @property
    def family_name(self):
        return addresser.FAMILY_NAME

    @property
    def family_versions(self):
        return [addresser.FAMILY_VERSION]

    @property
    def namespaces(self):
        return [addresser.NAMESPACE]

    def apply(self, transaction, context):
        header = transaction.header
        payload = PnrdNetPayload(transaction.payload)
        state = PnrdNetState(context)

        _validate_timestamp(payload.timestamp)
```

```
        if payload.action == payload_pb2.PnrdPayload.CREATE_OWNER:
            _create_owner(
                state=state,
                public_key=header.signer_public_key,
                payload=payload)
        elif payload.action == payload_pb2.PnrdPayload.CREATE_RECORD:
            _create_record(
                state=state,
                public_key=header.signer_public_key,
                payload=payload)
        elif payload.action == payload_pb2.PnrdPayload.TRANSFER_RECORD:
            _transfer_record(
                state=state,
                public_key=header.signer_public_key,
                payload=payload)
        elif payload.action == payload_pb2.PnrdPayload.UPDATE_RECORD:
            _update_record(
                state=state,
                public_key=header.signer_public_key,
                payload=payload)
        else:
            raise InvalidTransaction('Unhandled action')


def _create_owner(state, public_key, payload):
    if state.get_owner(public_key):
        raise InvalidTransaction('Owner with the public key {} already '
                                 'exists'.format(public_key))
    state.set_owner(
        public_key=public_key,
        name=payload.data.name,
        timestamp=payload.timestamp)


def _create_record(state, public_key, payload):
    if state.get_owner(public_key) is None:
        raise InvalidTransaction('Owner with the public key {} does '
                                 'not exist'.format(public_key))

    if payload.data.record_id == '':
        raise InvalidTransaction('No record ID provided')

    if state.get_record(payload.data.record_id):
        raise InvalidTransaction('Identifier {} belongs to an existing '
                                 'record'.format(payload.data.record_id))

    _validate_tag(payload.data.tag_id)

    state.set_record(
        public_key=public_key,
        record_id=payload.data.record_id,
        tag_id=payload.data.tag_id,
        reader_id=payload.data.reader_id,
```

```
        ant_id=payload.data.ant_id,
        situation=payload.data.situation,
        places=payload.data.places,
        transitions=payload.data.transitions,
        incidenceMatrix=payload.data.incidenceMatrix,
        token=payload.data.token,
        timestamp=payload.timestamp)


def _transfer_record(state, public_key, payload):
    if state.get_owner(payload.data.receiving_owner) is None:
        raise InvalidTransaction(
            'Owner with the public key {} does '
            'not exist'.format(payload.data.receiving_owner))

    record = state.get_record(payload.data.record_id)
    if record is None:
        raise InvalidTransaction('Record with the record id {} does not '
                                 'exist'.format(payload.data.record_id))

    if not _validate_record_owner(signer_public_key=public_key,
                                  record=record):
        raise InvalidTransaction(
            'Transaction signer is not the owner of the record')

    state.transfer_record(
        receiving_owner=payload.data.receiving_owner,
        record_id=payload.data.record_id,
        timestamp=payload.timestamp)


def _update_record(state, public_key, payload):
    record = state.get_record(payload.data.record_id)
    if record is None:
        raise InvalidTransaction('Record with the record id {} does not '
                                 'exist'.format(payload.data.record_id))

    if not _validate_record_owner(signer_public_key=public_key,
                                  record=record):
        raise InvalidTransaction(
            'Transaction signer is not the owner of the record')

    state.update_record(
        record_id=payload.data.record_id,
        reader_id=payload.data.reader_id,
        ant_id=payload.data.ant_id,
        situation=payload.data.situation,
        places=payload.data.places,
        transitions=payload.data.transitions,
        incidenceMatrix=payload.data.incidenceMatrix,
        token=payload.data.token,
        timestamp=payload.timestamp)
```

```
def _validate_record_owner(signer_public_key, record):
    """Validates that the public key of the signer is the latest (i.e.
    current) owner of the record
    """
    latest_owner = max(record.owners, key=lambda obj:
obj.timestamp).owner_id
    return latest_owner == signer_public_key


def _validate_tag(tag_id):
    if tag_id is None or tag_id == '':
        raise InvalidTransaction('Incorrect TAG')


def _validate_timestamp(timestamp):
    """Validates that the client submitted timestamp for a transaction is
not
    greater than current time, within a tolerance defined by SYNC_TOLERANCE

    NOTE: Timestamp validation can be challenging since the machines that
are
    submitting and validating transactions may have different system times
    """
    dts = datetime.datetime.utcnow()
    current_time = round(time.mktime(dts.timetuple()) +
dts.microsecond/1e6)
    if (timestamp - current_time) > SYNC_TOLERANCE:
        raise InvalidTransaction(
            'Timestamp must be less than local time.'
            ' Expected {0} in ({1}-{2}, {1}+{2})'.format(
                timestamp, current_time, SYNC_TOLERANCE))
```

payload.py

```
from sawtooth_sdk.processor.exceptions import InvalidTransaction

from pnrdnet_protobuf import payload_pb2


class PnrdNetPayload(object):

    def __init__(self, payload):
        self._transaction = payload_pb2.PnrdPayload()
        self._transaction.ParseFromString(payload)

    @property
    def action(self):
        return self._transaction.action
```

```
    @property
    def data(self):
        if self._transaction.HasField('create_owner') and \
            self._transaction.action == \
                payload_pb2.PnrdPayload.CREATE_OWNER:
            return self._transaction.create_owner

        if self._transaction.HasField('create_record') and \
            self._transaction.action == \
                payload_pb2.PnrdPayload.CREATE_RECORD:
            return self._transaction.create_record

        if self._transaction.HasField('transfer_record') and \
            self._transaction.action == \
                payload_pb2.PnrdPayload.TRANSFER_RECORD:
            return self._transaction.transfer_record

        if self._transaction.HasField('update_record') and \
            self._transaction.action == \
                payload_pb2.PnrdPayload.UPDATE_RECORD:
            return self._transaction.update_record

        raise InvalidTransaction('Action does not match payload data')

    @property
    def timestamp(self):
        return self._transaction.timestamp
```

state.py

```
from pnrdnet_addressing import addresser

from pnrdnet_protobuf import owner_pb2
from pnrdnet_protobuf import record_pb2


class PnrdNetState(object):
    def __init__(self, context, timeout=2):
        self._context = context
        self._timeout = timeout

    def get_owner(self, public_key):
        """Gets the owner associated with the public_key

        Args:
            public_key (str): The public key of the agent

        Returns:
            owner_pb2.Owner: Agent with the provided public_key
        """
```

```
        address = addresser.get_owner_address(public_key)
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container = owner_pb2.OwnerContainer()
            container.ParseFromString(state_entries[0].data)
            for owner in container.entries:
                if owner.public_key == public_key:
                    return owner

        return None

    def set_owner(self, public_key, name, timestamp):
        """Creates a new owner in state

        Args:
            public_key (str): The public key of the agent
            name (str): The human-readable name of the agent
            timestamp (int): Unix UTC timestamp of when the agent was
created
        """
        address = addresser.get_owner_address(public_key)
        owner = owner_pb2.Owner(
            public_key=public_key, name=name, timestamp=timestamp)
        container = owner_pb2.OwnerContainer()
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container.ParseFromString(state_entries[0].data)

        container.entries.extend([owner])
        data = container.SerializeToString()

        updated_state = {}
        updated_state[address] = data
        self._context.set_state(updated_state, timeout=self._timeout)

    def get_record(self, record_id):
        """Gets the record associated with the record_id

        Args:
            record_id (str): The id of the record

        Returns:
            record_pb2.Record: Record with the provided record_id
        """
        address = addresser.get_record_address(record_id)
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container = record_pb2.RecordContainer()
            container.ParseFromString(state_entries[0].data)
            for record in container.entries:
                if record.record_id == record_id:
```

```
                    return record

        return None

    def set_record(self,
                   public_key,
                   reader_id,
                   ant_id,
                   situation,
                   places,
                   transitions,
                   incidenceMatrix,
                   token,
                   record_id,
                   tag_id,
                   timestamp):
        """Creates a new record in state
        """
        address = addresser.get_record_address(record_id)
        owner = record_pb2.Record.Owner(
            owner_id=public_key,
            timestamp=timestamp)
        history = record_pb2.Record.History(
            reader_id=reader_id,
            ant_id=ant_id,
            situation=situation,
            places=places,
            transitions=transitions,
            incidenceMatrix=incidenceMatrix,
            token=token,
            timestamp=timestamp)
        record = record_pb2.Record(
            record_id=record_id,
            tag_id=tag_id,
            owners=[owner],
            history=[history])
        container = record_pb2.RecordContainer()
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container.ParseFromString(state_entries[0].data)
        container.entries.extend([record])
        data = container.SerializeToString()

        updated_state = {}
        updated_state[address] = data
        self._context.set_state(updated_state, timeout=self._timeout)

    def transfer_record(self, receiving_owner, record_id, timestamp):
        owner = record_pb2.Record.Owner(
            owner_id=receiving_owner,
            timestamp=timestamp)
        address = addresser.get_record_address(record_id)
        container = record_pb2.RecordContainer()
```

```
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container.ParseFromString(state_entries[0].data)
            for record in container.entries:
                if record.record_id == record_id:
                    record.owners.extend([owner])
        data = container.SerializeToString()
        updated_state = {}
        updated_state[address] = data
        self._context.set_state(updated_state, timeout=self._timeout)

    def update_record(self,
                      reader_id,
                      ant_id,
                      situation,
                      places,
                      transitions,
                      incidenceMatrix,
                      token,
                      record_id,
                      timestamp):
        history = record_pb2.Record.History(
            reader_id=reader_id,
            ant_id=ant_id,
            situation=situation,
            places=places,
            transitions=transitions,
            incidenceMatrix=incidenceMatrix,
            token=token,
            timestamp=timestamp)
        address = addresser.get_record_address(record_id)
        container = record_pb2.RecordContainer()
        state_entries = self._context.get_state(
            addresses=[address], timeout=self._timeout)
        if state_entries:
            container.ParseFromString(state_entries[0].data)
            for record in container.entries:
                if record.record_id == record_id:
                    record.history.extend([history])
        data = container.SerializeToString()
        updated_state = {}
        updated_state[address] = data
        self._context.set_state(updated_state, timeout=self._timeout)
```

Esse programa contém parte de código dos repositórios:

Sawtooth Simple Supply:

- https://github.com/hyperledger/education-sawtooth-simple-supply

- codeowners: @agunde406 @chenette @danintel @dcmiddle @dplumb94 @jsmitchell @peterschwarz @vaporos

Sawtooth Core: https://github.com/hyperledger/sawtooth-core

requirements.txt

```
autopep8==1.5.7
bcrypt==3.2.0
black==21.9b0
cbor==1.0.0
certifi==2021.10.8
cffi==1.15.0
charset-normalizer==2.0.7
click==8.0.3
colorlog==6.5.0
Flask==2.0.2
idna==3.3
itsdangerous==2.0.1
Jinja2==3.0.2
MarkupSafe==2.0.1
mypy-extensions==0.4.3
pathspec==0.9.0
platformdirs==2.4.0
protobuf==3.18.1
pycodestyle==2.8.0
pycparser==2.20
pycrypto==2.6.1
python-dotenv==0.19.1
PyYAML==6.0
pyzmq==22.3.0
regex==2021.10.8
requests==2.26.0
sawtooth-sdk==1.2.3
secp256k1==0.11.1
six==1.16.0
toml==0.10.2
tomli==1.2.1
typing-extensions==3.10.0.2
urllib3==1.26.7
Werkzeug==2.0.2
```

APÊNDICE D – PROGRAMA CLIENTE

main.py

```
import os
import sys
import logging
import argparse
from flask import Flask
from pnrdnet_api.routes.core import core_routes
from pnrdnet_api.routes.owner import owner_routes
from pnrdnet_api.routes.record import record_routes
from pnrdnet_api.config import CoreConfig
from pnrdnet_api.utils.responses import response_with
import pnrdnet_api.utils.responses as resp
from dotenv import load_dotenv


load_dotenv()
LOGGER = logging.getLogger(__name__)


def parse_args(args):
    parser = argparse.ArgumentParser(
        description='Starts the PNRD NET')

    parser.add_argument(
        '-B', '--bind',
        help='identify host and port for api to run on',
        default='localhost:8000')
    parser.add_argument(
        '-C', '--connect',
        help='specify URL to connect to a running validator',
        default='tcp://localhost:4004')
    parser.add_argument(
        '-t', '--timeout',
        help='set time (in seconds) to wait for a validator response',
        default=500)
    parser.add_argument(
        '--primary-key',
        help='The authorized primary key for blockchain access',
        default='sawtooth')
    parser.add_argument(
        '--public-key',
        help='The public key for blockchain access',
        default='sawtooth')
    parser.add_argument(
        '--password',
        help="The authorized password for blockchain access",
        default='sawtooth')
    parser.add_argument(
        '-v', '--verbose',
        action='count',
```

```
        default=0,
        help='enable more verbose output to stderr')

    return parser.parse_args(args)


def create_app(config):
    """
    """
    app = Flask(__name__)
    app.config.from_object(config)

    # BLUEPRINTS
    app.register_blueprint(core_routes, url_prefix="/core")
    app.register_blueprint(owner_routes, url_prefix="/owner")
    app.register_blueprint(record_routes, url_prefix="/record")
    # START GLOBAL HTTP CONFIGURATIONS

    @app.after_request
    def add_header(response):
        return response

    @app.errorhandler(400)
    def bad_request(e):
        logging.error(e)
        return response_with((resp.BAD_REQUEST_400))

    @app.errorhandler(500)
    def server_error(e):
        logging.error(e)
        return response_with(resp.SERVER_ERROR_500)

    @app.errorhandler(404)
    def not_found(e):
        logging.error(e)
        return response_with(resp.SERVER_ERROR_404)

    logging.basicConfig(
        stream=sys.stdout,
        format="%(asctime)s|%(levelname)s|%(filename)s:%(lineno)s|%
(message)s",
        level=logging.DEBUG,
    )
    return app


app_config = CoreConfig

app = create_app(app_config)

if __name__ == "__main__":
    app.run(port=5000, host="0.0.0.0", use_reloader=False)
```

decoding.py

```
from pnrdnet_addressing.addresser import AddressSpace
from pnrdnet_addressing.addresser import get_address_type
from pnrdnet_protobuf.owner_pb2 import OwnerContainer
from pnrdnet_protobuf.record_pb2 import RecordContainer


CONTAINERS = {
    AddressSpace.OWNER: OwnerContainer,
    AddressSpace.RECORD: RecordContainer
}


def _parse_proto(proto_class, data):
    deserialized = proto_class()
    deserialized.ParseFromString(data)
    return deserialized


def _convert_proto_to_dict(proto):
    result = {}

    for field in proto.DESCRIPTOR.fields:
        key = field.name
        value = getattr(proto, key)
        if field.type == field.TYPE_MESSAGE:
            if field.label == field.LABEL_REPEATED:
                result[key] = [_convert_proto_to_dict(p) for p in value]
            else:
                result[key] = _convert_proto_to_dict(value)

        elif field.type == field.TYPE_ENUM:
            number = int(value)
            name = field.enum_type.values_by_number.get(number).name
            result[key] = name

        else:
            result[key] = value

    return result


def deserialize_data(address, data):
    """Deserializes state data by type based on the address structure and
    returns it as a dictionary with the associated data type

    Args:
        address (str): The state address of the container
        data (str): String containing the serialized state data
```

```
    """
    data_type = get_address_type(address)

    if data_type == AddressSpace.OTHER_FAMILY:
        return []

    try:
        container = CONTAINERS[data_type]
    except KeyError:
        raise TypeError('Unknown data type: {}'.format(data_type))

    entries = _parse_proto(container, data).entries
    return data_type, [_convert_proto_to_dict(pb) for pb in entries]
```

dispatcher/Dispatcher.py

```
import time
from typing import Tuple
import requests
import yaml
import base64
import cbor
from sawtooth_sdk.protobuf import client_batch_submit_pb2
from sawtooth_sdk.protobuf import validator_pb2
from sawtooth_sdk.protobuf import batch_pb2

from sawtooth_signing import create_context
from sawtooth_signing import CryptoFactory
from sawtooth_signing import secp256k1

from pnrdnet_addressing.addresser import NAMESPACE, AddressSpace,
get_owner_address, get_record_address
from pnrdnet_api.decoding import deserialize_data
from pnrdnet_protobuf.owner_pb2 import _OWNER

from .transaction_creation import make_create_owner_transaction
from .transaction_creation import make_create_record_transaction
from .transaction_creation import make_transfer_record_transaction
from .transaction_creation import make_update_record_transaction
from pnrdnet_api.utils.errors import ApiBadRequest, ApiInternalError
from pnrdnet_api.config import DEFAULT_URL_SAWTOOH_REST_API,
DEFAULT_PASS_SAWTOOH_REST_API, DEFAULT_USER_SAWTOOH_REST_API


class Dispatcher(object):
    def __init__(self, sawtooth_rest_api_url=DEFAULT_URL_SAWTOOH_REST_API):
        self.sawtooth_rest_api_url = sawtooth_rest_api_url
        self._context = create_context('secp256k1')
        self._crypto_factory = CryptoFactory(self._context)
        self._batch_signer = self._crypto_factory.new_signer(
```

```
            self._context.new_random_private_key())

    def open_validator_connection(self):
        self._connection.open()

    def close_validator_connection(self):
        self._connection.close()

    def get_new_key_pair(self):
        private_key = self._context.new_random_private_key()
        public_key = self._context.get_public_key(private_key)
        return public_key.as_hex(), private_key.as_hex()

    def _get_status(self, batch_id, wait):
        try:
            result = self._send_request(
                f'batch_statuses?id={batch_id}&wait={wait}')
            return yaml.safe_load(result)['data'][0]['status']
        except BaseException as err:
            raise ApiBadRequest(err) from err

    def _send_request(self, suffix, data=None, name=None,
http_verb='POST'):
        url = f"{self.sawtooth_rest_api_url}/{suffix}"
        headers = {
            'Content-Type': 'application/octet-stream'
        }
        try:
            if data is not None and http_verb == 'POST':
                result = requests.post(url,
                                       auth=(DEFAULT_USER_SAWTOOH_REST_API,

DEFAULT_PASS_SAWTOOH_REST_API),
                                       headers=headers, data=data)
            else:
                result = requests.get(url, auth=
(DEFAULT_USER_SAWTOOH_REST_API,

DEFAULT_PASS_SAWTOOH_REST_API), headers=headers)

            if result.status_code == 404:
                raise ApiBadRequest("No such key: {}".format(name))

            if not result.ok:
                raise ApiBadRequest("Error {}: {}".format(
                    result.status_code, result.reason))

        except requests.ConnectionError as err:
            raise ApiBadRequest(
                'Failed to connect to REST API: {}'.format(err)) from err

        except BaseException as err:
            raise ApiBadRequest(err) from err
```

```
        return result.text

    def _get_blockchain_data_net(self, address, result):
        encoded_entries = yaml.safe_load(result)["data"]
        deserialize_data_batch = []
        for n in encoded_entries:
            decode_data = base64.b64decode(n["data"])
            data_type, resources = deserialize_data(n["address"],
decode_data)
            if data_type is AddressSpace.RECORD:
                deserialize_data_batch.append(
                    {"type": 'RECORD', "record_id": resources[0]
['record_id']})
            elif data_type is AddressSpace.OWNER:
                deserialize_data_batch.append(
                    {"type": 'OWNER', "name": resources[0]['name']})
        return deserialize_data_batch

    def _get_blockchain_data(self, address, result):
        encoded_entries = yaml.safe_load(result)["data"]
        decode_data = [
            base64.b64decode(entry["data"])
            for entry in encoded_entries
        ]
        deserialized_data = []
        for entry in decode_data:
            data_type, resources = deserialize_data(address, entry)
            deserialized_data.append(resources)
        return deserialized_data

    def _transaction_signer(self, private_key):
        return self._crypto_factory.new_signer(
            secp256k1.Secp256k1PrivateKey.from_hex(private_key))

    def get_owner_data(self, public_key):
        owner_address = get_owner_address(public_key)
        result = self._send_request(
            f"state?address={owner_address}")

        try:
            deserialized_data = self._get_blockchain_data(
                owner_address, result)
            return (deserialized_data, owner_address)
        except BaseException as e:
            print(e)
            return None

    def get_record_data(self, record_id):
        record_address = get_record_address(record_id)
        result = self._send_request(
            f"state?address={record_address}")
        try:
            deserialized_data = self._get_blockchain_data(
                record_address, result)
```

```
            return (deserialized_data, record_address)
        except BaseException as e:
            print(e)
            return None

    def get_network_data(self):
        namespace_address = NAMESPACE

        result = self._send_request(
            f"state?address={namespace_address}")
        try:

            deserialized_data = self._get_blockchain_data_net(
                namespace_address, result)
            return (deserialized_data, namespace_address)
        except BaseException as e:
            print(e)
            return None

    def send_create_owner_transaction(self,
                                      private_key,
                                      name,
                                      timestamp):
        transaction_signer = self._transaction_signer(private_key)
        batch = make_create_owner_transaction(
            transaction_signer=transaction_signer,
            batch_signer=self._batch_signer,
            name=name,
            timestamp=timestamp)

        response, status = self.post_batch(
            batch=batch, transaction_name="create_owner", wait=1)
        return response, status
        # self._send_and_wait_for_commit(batch)

    def send_create_record_transaction(self,
                                       private_key,
                                       reader_id,
                                       ant_id,
                                       situation,
                                       places,
                                       transitions,
                                       incidenceMatrix,
                                       token,
                                       record_id,
                                       tag_id,
                                       timestamp):

        transaction_signer = self._transaction_signer(private_key)
        batch = make_create_record_transaction(
            transaction_signer=transaction_signer,
            batch_signer=self._batch_signer,
            reader_id=reader_id,
            ant_id=ant_id,
```

```
            situation=situation,
            places=places,
            transitions=transitions,
            incidenceMatrix=incidenceMatrix,
            token=token,
            record_id=record_id,
            tag_id=tag_id,
            timestamp=timestamp)
        response, status = self.post_batch(
            batch=batch, transaction_name="create_record", wait=1)
        return response, status

    def send_transfer_record_transaction(self,
                                         private_key,
                                         receiving_owner,
                                         record_id,
                                         timestamp):
        transaction_signer = self._transaction_signer(private_key)

        batch = make_transfer_record_transaction(
            transaction_signer=transaction_signer,
            batch_signer=self._batch_signer,
            receiving_owner=receiving_owner,
            record_id=record_id,
            timestamp=timestamp)

        response, status = self.post_batch(
            batch=batch, transaction_name="transfer_record", wait=1)
        return response, status

    def send_update_record_transaction(self,
                                       private_key,
                                       reader_id,
                                       ant_id,
                                       situation,
                                       places,
                                       transitions,
                                       incidenceMatrix,
                                       token,
                                       record_id,
                                       timestamp):
        transaction_signer = self._transaction_signer(private_key)

        batch = make_update_record_transaction(
            transaction_signer=transaction_signer,
            batch_signer=self._batch_signer,
            reader_id=reader_id,
            ant_id=ant_id,
            situation=situation,
            places=places,
            transitions=transitions,
            incidenceMatrix=incidenceMatrix,
            token=token,
            record_id=record_id,
```

```
                timestamp=timestamp)

            response, status = self.post_batch(
                batch=batch, transaction_name="update_record", wait=1)
            return response, status

        def post_batch(self, batch,  transaction_name, wait=None) -> tuple[str,
    str]:
            batch_list = batch_pb2.BatchList(batches=[batch])
            batch_id = batch.header_signature
            if wait and wait > 0:
                wait_time = 0
                start_time = time.time()
                response = self._send_request(
                    suffix="batches", data=batch_list.SerializeToString(),
    name=transaction_name)
                while wait_time < wait:
                    status = self._get_status(
                        batch_id,
                        wait - int(wait_time),
                    )
                    wait_time = time.time() - start_time
                    if status != 'PENDING':
                        return (response, status)

                return (response, "")

            rest = self._send_request(
                suffix="batches", data=batch_list.SerializeToString(),
    name=transaction_name,  http_verb="POST")
            return (rest, "")

        def _send_and_wait_for_commit(self, batch):
            # Send transaction to validator
            submit_request = client_batch_submit_pb2.ClientBatchSubmitRequest(
                batches=[batch])
            self._connection.send(
                validator_pb2.Message.CLIENT_BATCH_SUBMIT_REQUEST,
                submit_request.SerializeToString())

            # Send status request to validator
            batch_id = batch.header_signature
            status_request = client_batch_submit_pb2.ClientBatchStatusRequest(
                batch_ids=[batch_id], wait=True)
            validator_response = self._connection.send(
                validator_pb2.Message.CLIENT_BATCH_STATUS_REQUEST,
                status_request.SerializeToString())

            # Parse response
            status_response =
    client_batch_submit_pb2.ClientBatchStatusResponse()
            status_response.ParseFromString(validator_response.content)
            status = status_response.batch_statuses[0].status
            if status == client_batch_submit_pb2.ClientBatchStatus.INVALID:
```

```
            error =
status_response.batch_statuses[0].invalid_transactions[0]
            raise ApiBadRequest(error.message)
        elif status == client_batch_submit_pb2.ClientBatchStatus.PENDING:
            raise ApiInternalError('Transaction submitted but timed out')
        elif status == client_batch_submit_pb2.ClientBatchStatus.UNKNOWN:
            raise ApiInternalError('Something went wrong. Try again later')
```

dispatcher/transaction_creation.py

```
import hashlib

from sawtooth_sdk.protobuf import batch_pb2
from sawtooth_sdk.protobuf import transaction_pb2

from pnrdnet_addressing import addresser

from pnrdnet_protobuf import payload_pb2


def _make_batch(payload_bytes,
                inputs,
                outputs,
                transaction_signer,
                batch_signer):

    transaction_header = transaction_pb2.TransactionHeader(
        family_name=addresser.FAMILY_NAME,
        family_version=addresser.FAMILY_VERSION,
        inputs=inputs,
        outputs=outputs,
        signer_public_key=transaction_signer.get_public_key().as_hex(),
        batcher_public_key=batch_signer.get_public_key().as_hex(),
        dependencies=[],
        payload_sha512=hashlib.sha512(payload_bytes).hexdigest())
    transaction_header_bytes = transaction_header.SerializeToString()

    transaction = transaction_pb2.Transaction(
        header=transaction_header_bytes,
        header_signature=transaction_signer.sign(transaction_header_bytes),
        payload=payload_bytes)

    batch_header = batch_pb2.BatchHeader(
        signer_public_key=batch_signer.get_public_key().as_hex(),
        transaction_ids=[transaction.header_signature])
    batch_header_bytes = batch_header.SerializeToString()

    batch = batch_pb2.Batch(
```

```
        header=batch_header_bytes,
        header_signature=batch_signer.sign(batch_header_bytes),
        transactions=[transaction])

    return batch


def make_create_owner_transaction(transaction_signer,
                                  batch_signer,
                                  name,
                                  timestamp):
    """Make a CreateOwnerAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        name (str): The owner's name
        timestamp (int): Unix UTC timestamp of when the owner is created

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch

    """

    owner_address = addresser.get_owner_address(
        transaction_signer.get_public_key().as_hex())

    inputs = [owner_address]

    outputs = [owner_address]

    action = payload_pb2.CreateOwnerAction(name=name)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.CREATE_OWNER,
        create_owner=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)


def make_create_record_transaction(transaction_signer,
                                   batch_signer,
                                   reader_id,
                                   ant_id,
                                   situation,
                                   places,
```

```
                                        transitions,
                                        incidenceMatrix,
                                        token,
                                        record_id,
                                        tag_id,
                                        timestamp):
    """Make a CreateRecordAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        ...

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch
    """

    inputs = [
        addresser.get_owner_address(
            transaction_signer.get_public_key().as_hex()),
        addresser.get_record_address(record_id)
    ]

    outputs = [addresser.get_record_address(record_id)]

    action = payload_pb2.CreateRecordAction(
        record_id=record_id,
        reader_id=reader_id,
        ant_id=ant_id,
        situation=situation,
        places=places,
        transitions=transitions,
        incidenceMatrix=incidenceMatrix,
        token=token,
        tag_id=tag_id)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.CREATE_RECORD,
        create_record=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)


def make_transfer_record_transaction(transaction_signer,
                                     batch_signer,
                                     receiving_owner,
```

```
                                       record_id,
                                       timestamp):
    """Make a CreateRecordAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        receiving_owner (str): Public key of the agent receiving the record
        record_id (str): Unique ID of the record
        timestamp (int): Unix UTC timestamp of when the record is
transferred

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch
    """
    sending_owner_address = addresser.get_owner_address(
        transaction_signer.get_public_key().as_hex())
    receiving_owner_address = addresser.get_owner_address(receiving_owner)
    record_address = addresser.get_record_address(record_id)

    inputs = [sending_owner_address, receiving_owner_address,
record_address]

    outputs = [record_address]

    action = payload_pb2.TransferRecordAction(
        record_id=record_id,
        receiving_owner=receiving_owner)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.TRANSFER_RECORD,
        transfer_record=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)


def make_update_record_transaction(transaction_signer,
                                   batch_signer,
                                   reader_id,
                                   ant_id,
                                   situation,
                                   places,
                                   transitions,
                                   incidenceMatrix,
                                   token,
                                   record_id,
```

```
                                          timestamp):
    """Make a CreateRecordAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        timestamp (int): Unix UTC timestamp of when the record is updated

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch
    """
    owner_address = addresser.get_owner_address(
        transaction_signer.get_public_key().as_hex())
    record_address = addresser.get_record_address(record_id)

    inputs = [owner_address, record_address]

    outputs = [record_address]

    action = payload_pb2.UpdateRecordAction(
        record_id=record_id,
        reader_id=reader_id,
        ant_id=ant_id,
        situation=situation,
        places=places,
        transitions=transitions,
        incidenceMatrix=incidenceMatrix,
        token=token)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.UPDATE_RECORD,
        update_record=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)
```

routes/core.py

```
import hashlib

from sawtooth_sdk.protobuf import batch_pb2
```

```
from sawtooth_sdk.protobuf import transaction_pb2

from pnrdnet_addressing import addresser

from pnrdnet_protobuf import payload_pb2


def _make_batch(payload_bytes,
                inputs,
                outputs,
                transaction_signer,
                batch_signer):

    transaction_header = transaction_pb2.TransactionHeader(
        family_name=addresser.FAMILY_NAME,
        family_version=addresser.FAMILY_VERSION,
        inputs=inputs,
        outputs=outputs,
        signer_public_key=transaction_signer.get_public_key().as_hex(),
        batcher_public_key=batch_signer.get_public_key().as_hex(),
        dependencies=[],
        payload_sha512=hashlib.sha512(payload_bytes).hexdigest())
    transaction_header_bytes = transaction_header.SerializeToString()

    transaction = transaction_pb2.Transaction(
        header=transaction_header_bytes,
        header_signature=transaction_signer.sign(transaction_header_bytes),
        payload=payload_bytes)

    batch_header = batch_pb2.BatchHeader(
        signer_public_key=batch_signer.get_public_key().as_hex(),
        transaction_ids=[transaction.header_signature])
    batch_header_bytes = batch_header.SerializeToString()

    batch = batch_pb2.Batch(
        header=batch_header_bytes,
        header_signature=batch_signer.sign(batch_header_bytes),
        transactions=[transaction])

    return batch


def make_create_owner_transaction(transaction_signer,
                                  batch_signer,
                                  name,
                                  timestamp):
    """Make a CreateOwnerAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        name (str): The owner's name
        timestamp (int): Unix UTC timestamp of when the owner is created
```

```
    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch

    """

    owner_address = addresser.get_owner_address(
        transaction_signer.get_public_key().as_hex())

    inputs = [owner_address]

    outputs = [owner_address]

    action = payload_pb2.CreateOwnerAction(name=name)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.CREATE_OWNER,
        create_owner=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)


def make_create_record_transaction(transaction_signer,
                                   batch_signer,
                                   reader_id,
                                   ant_id,
                                   situation,
                                   places,
                                   transitions,
                                   incidenceMatrix,
                                   token,
                                   record_id,
                                   tag_id,
                                   timestamp):
    """Make a CreateRecordAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        ...

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch
    """

    inputs = [
```

```
        addresser.get_owner_address(
            transaction_signer.get_public_key().as_hex()),
        addresser.get_record_address(record_id)
    ]

    outputs = [addresser.get_record_address(record_id)]

    action = payload_pb2.CreateRecordAction(
        record_id=record_id,
        reader_id=reader_id,
        ant_id=ant_id,
        situation=situation,
        places=places,
        transitions=transitions,
        incidenceMatrix=incidenceMatrix,
        token=token,
        tag_id=tag_id)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.CREATE_RECORD,
        create_record=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)


def make_transfer_record_transaction(transaction_signer,
                                     batch_signer,
                                     receiving_owner,
                                     record_id,
                                     timestamp):
    """Make a CreateRecordAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        receiving_owner (str): Public key of the agent receiving the record
        record_id (str): Unique ID of the record
        timestamp (int): Unix UTC timestamp of when the record is
transferred

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch
    """
    sending_owner_address = addresser.get_owner_address(
        transaction_signer.get_public_key().as_hex())
    receiving_owner_address = addresser.get_owner_address(receiving_owner)
```

```
    record_address = addresser.get_record_address(record_id)

    inputs = [sending_owner_address, receiving_owner_address,
record_address]

    outputs = [record_address]

    action = payload_pb2.TransferRecordAction(
        record_id=record_id,
        receiving_owner=receiving_owner)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.TRANSFER_RECORD,
        transfer_record=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)


def make_update_record_transaction(transaction_signer,
                                   batch_signer,
                                   reader_id,
                                   ant_id,
                                   situation,
                                   places,
                                   transitions,
                                   incidenceMatrix,
                                   token,
                                   record_id,
                                   timestamp):
    """Make a CreateRecordAction transaction and wrap it in a batch

    Args:
        transaction_signer (sawtooth_signing.Signer): The transaction key
pair
        batch_signer (sawtooth_signing.Signer): The batch key pair
        timestamp (int): Unix UTC timestamp of when the record is updated

    Returns:
        batch_pb2.Batch: The transaction wrapped in a batch
    """
    owner_address = addresser.get_owner_address(
        transaction_signer.get_public_key().as_hex())
    record_address = addresser.get_record_address(record_id)

    inputs = [owner_address, record_address]

    outputs = [record_address]
```

```
    action = payload_pb2.UpdateRecordAction(
        record_id=record_id,
        reader_id=reader_id,
        ant_id=ant_id,
        situation=situation,
        places=places,
        transitions=transitions,
        incidenceMatrix=incidenceMatrix,
        token=token)

    payload = payload_pb2.PnrdPayload(
        action=payload_pb2.PnrdPayload.UPDATE_RECORD,
        update_record=action,
        timestamp=timestamp)
    payload_bytes = payload.SerializeToString()

    return _make_batch(
        payload_bytes=payload_bytes,
        inputs=inputs,
        outputs=outputs,
        transaction_signer=transaction_signer,
        batch_signer=batch_signer)
```

routes/owner.py

```
from flask import Blueprint, request
from pnrdnet_api.config import AES_KEY, APP_SECRET_KEY
from pnrdnet_api.dispatcher.Dispatcher import Dispatcher
from pnrdnet_api.utils.functions import encrypt_private_key,
generate_auth_token, get_time, hash_password, validate_fields
from pnrdnet_api.utils.responses import response_with
from pnrdnet_api.utils import responses as resp


owner_routes = Blueprint("owner_routes", __name__)


@owner_routes.route("/create", methods=["POST"])
def create_owner():
    try:
        data = request.get_json()
        required_fields = ['name']
        validate_fields(required_fields, data)
        dispatch = Dispatcher()
        public_key, private_key = dispatch.get_new_key_pair()

        result, status = dispatch.send_create_owner_transaction(
            private_key=private_key,
```

```
            name=data.get('name'),
            timestamp=get_time())

        encrypted_private_key = encrypt_private_key(
            AES_KEY, public_key, private_key)

        token = generate_auth_token(encrypted_private_key, public_key)
        return response_with(
            resp.SUCCESS_201,
            value={'public_key': public_key,
                   'private_key': private_key,
                   "statusBlockchain": status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)


@owner_routes.route("/detail", methods=["POST"])
def get_owner_details():
    try:
        data = request.get_json()
        required_fields = ['public_key']
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        owner_data, owner_address = dispatch.get_owner_data(
            public_key=data.get('public_key'))

        return response_with(
            resp.SUCCESS_201,
            value={'address': owner_address, 'data': owner_data}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)
```

routes/record.py

```
from flask import Blueprint, request
from pnrdnet_api.config import AES_KEY, APP_SECRET_KEY
from pnrdnet_api.dispatcher.Dispatcher import Dispatcher
from pnrdnet_api.utils.functions import get_time, validate_fields
from pnrdnet_api.utils.responses import response_with
from pnrdnet_api.utils import responses as resp
from google.protobuf.json_format import MessageToJson


record_routes = Blueprint("record_routes", __name__)
```

```python
@record_routes.route("/create", methods=["POST"])
def create_record():
    try:
        data = request.get_json()
        required_fields = [
            'private_key',
            'record_id',
            'reader_id',
            'ant_id',
            'situation',
            'places',
            'transitions',
            'incidenceMatrix',
            'token',
            'tag_id'
        ]
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        result, status = dispatch.send_create_record_transaction(
            private_key=data['private_key'],
            record_id=data['record_id'],
            reader_id=data['reader_id'],
            ant_id=data['ant_id'],
            situation=data['situation'],
            places=data['places'],
            transitions=data['transitions'],
            incidenceMatrix=data['incidenceMatrix'],
            token=data['token'],
            tag_id=data['tag_id'],
            timestamp=get_time())

        return response_with(
            resp.SUCCESS_201,
            value={
                'data': f'Create record transaction {status}',
                'statusBlockchain': status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)


@record_routes.route("/detail", methods=["POST"])
def get_record_details():
    try:
        data = request.get_json()
        required_fields = ['record_id']
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        record_data, record_address = dispatch.get_record_data(
```

```
            record_id=data.get('record_id'))
        record = record_data[0][0]
        record_decoded = {
            'tag_id': record['tag_id'],
            'record_id': record['record_id'],
            'owners': record['owners'],
            'history': list()
        }
        for history in record['history']:
            record_decoded['history'].append(
                {
                    'reader_id': history['reader_id'],
                    'ant_id': history['ant_id'],
                    'situation': history['situation'],
                    'places': history['places'],
                    'transitions': history['transitions'],
                    'incidenceMatrix': list(history['incidenceMatrix']),
                    'token': list(history['token']),
                }
            )
        return response_with(
            resp.SUCCESS_201,
            value={'address': record_address, 'data': record_decoded}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)


@record_routes.route("/transfer", methods=["POST"])
def transfer_record():
    try:
        data = request.get_json()
        required_fields = ['receiving_owner_pubkey',
                           'record_id', 'private_key']
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        result, status = dispatch.send_transfer_record_transaction(
            private_key=data['private_key'],
            receiving_owner=data['receiving_owner_pubkey'],
            record_id=data['record_id'],
            timestamp=get_time())
        return response_with(
            resp.SUCCESS_201,
            value={
                'data': f'Create transfer transaction {status}',
                'statusBlockchain': status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)
```

```
@record_routes.route("/update", methods=["POST"])
def update_record():
    try:
        data = request.get_json()
        required_fields = required_fields = [
            'private_key',
            'record_id',
            'reader_id',
            'ant_id',
            'situation',
            'places',
            'transitions',
            'incidenceMatrix',
            'token',
        ]
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        result, status = dispatch.send_update_record_transaction(
            private_key=data['private_key'],
            record_id=data['record_id'],
            reader_id=data['reader_id'],
            ant_id=data['ant_id'],
            situation=data['situation'],
            places=data['places'],
            transitions=data['transitions'],
            incidenceMatrix=data['incidenceMatrix'],
            token=data['token'],
            timestamp=get_time())
        return response_with(
            resp.SUCCESS_201,
            value={
                'data': f'Create update transaction {status}',
                'statusBlockchain': status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)
```

utils/functions.py

```
from flask import Blueprint, request
from pnrdnet_api.config import AES_KEY, APP_SECRET_KEY
from pnrdnet_api.dispatcher.Dispatcher import Dispatcher
from pnrdnet_api.utils.functions import get_time, validate_fields
from pnrdnet_api.utils.responses import response_with
from pnrdnet_api.utils import responses as resp
from google.protobuf.json_format import MessageToJson
```

```
record_routes = Blueprint("record_routes", __name__)


@record_routes.route("/create", methods=["POST"])
def create_record():
    try:
        data = request.get_json()
        required_fields = [
            'private_key',
            'record_id',
            'reader_id',
            'ant_id',
            'situation',
            'places',
            'transitions',
            'incidenceMatrix',
            'token',
            'tag_id'
        ]
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        result, status = dispatch.send_create_record_transaction(
            private_key=data['private_key'],
            record_id=data['record_id'],
            reader_id=data['reader_id'],
            ant_id=data['ant_id'],
            situation=data['situation'],
            places=data['places'],
            transitions=data['transitions'],
            incidenceMatrix=data['incidenceMatrix'],
            token=data['token'],
            tag_id=data['tag_id'],
            timestamp=get_time())

        return response_with(
            resp.SUCCESS_201,
            value={
                'data': f'Create record transaction {status}',
                'statusBlockchain': status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)


@record_routes.route("/detail", methods=["POST"])
def get_record_details():
    try:
        data = request.get_json()
        required_fields = ['record_id']
        validate_fields(required_fields, data)
        dispatch = Dispatcher()
```

```
        record_data, record_address = dispatch.get_record_data(
            record_id=data.get('record_id'))
        record = record_data[0][0]
        record_decoded = {
            'tag_id': record['tag_id'],
            'record_id': record['record_id'],
            'owners': record['owners'],
            'history': list()
        }
        for history in record['history']:
            record_decoded['history'].append(
                {
                    'reader_id': history['reader_id'],
                    'ant_id': history['ant_id'],
                    'situation': history['situation'],
                    'places': history['places'],
                    'transitions': history['transitions'],
                    'incidenceMatrix': list(history['incidenceMatrix']),
                    'token': list(history['token']),
                }
            )
        return response_with(
            resp.SUCCESS_201,
            value={'address': record_address, 'data': record_decoded}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)


@record_routes.route("/transfer", methods=["POST"])
def transfer_record():
    try:
        data = request.get_json()
        required_fields = ['receiving_owner_pubkey',
                           'record_id', 'private_key']
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        result, status = dispatch.send_transfer_record_transaction(
            private_key=data['private_key'],
            receiving_owner=data['receiving_owner_pubkey'],
            record_id=data['record_id'],
            timestamp=get_time())
        return response_with(
            resp.SUCCESS_201,
            value={
                'data': f'Create transfer transaction {status}',
                'statusBlockchain': status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)
```

```python
@record_routes.route("/update", methods=["POST"])
def update_record():
    try:
        data = request.get_json()
        required_fields = required_fields = [
            'private_key',
            'record_id',
            'reader_id',
            'ant_id',
            'situation',
            'places',
            'transitions',
            'incidenceMatrix',
            'token',
        ]
        validate_fields(required_fields, data)
        dispatch = Dispatcher()

        result, status = dispatch.send_update_record_transaction(
            private_key=data['private_key'],
            record_id=data['record_id'],
            reader_id=data['reader_id'],
            ant_id=data['ant_id'],
            situation=data['situation'],
            places=data['places'],
            transitions=data['transitions'],
            incidenceMatrix=data['incidenceMatrix'],
            token=data['token'],
            timestamp=get_time())
        return response_with(
            resp.SUCCESS_201,
            value={
                'data': f'Create update transaction {status}',
                'statusBlockchain': status}
        )
    except Exception as e:
        print(e)
        return response_with(resp.INVALID_INPUT_422)
```

utils/response.py

```python
from flask import make_response, jsonify

INVALID_FIELD_NAME_SENT_422 = {
    "http_code": 422,
    "status": "invalidField",
    "message": "Invalid Field",
```

```
}
INVALID_INPUT_422 = {
    "http_code": 422,
    "status": "invalidInput",
    "message": "Invalid Input",
}
INVALID_OR_KEY_422 = {
    "http_code": 422,
    "status": "duplicatedOrInvalidKey",
    "message": "Duplicated or Invalid Key",
}
MAX_SIZE_LIMIT_422 = {
    "http_code": 422,
    "status": "maxSizeLimit",
    "message": "The request exceeded the maximum size limit",
}
MISSING_PARAMETERS_422 = {
    "http_code": 422,
    "status": "missingParameter",
    "message": "Missing Parameter",
}
BAD_REQUEST_400 = {
    "http_code": 400,
    "status": "badRequest",
    "message": "Dar Request",
}
SERVER_ERROR_500 = {
    "http_code": 500,
    "status": "serverError",
    "message": "Server Error",
}
SERVER_ERROR_404 = {
    "http_code": 404,
    "status": "notFound",
    "message": "Resource not found",
}
UNAUTHORIZED_403 = {
    "http_code": 403,
    "status": "notAuthorized",
    "message": "You are not authorized to executed this action",
}
SUCCESS_200 = {"http_code": 200, "status": "success"}
SUCCESS_201 = {"http_code": 201, "status": "success"}
SUCCESS_204 = {"http_code": 204, "status": "success"}


def response_with(
    response, value=None, message=None, error=None, headers={},
pagination=None
):
    """
    Cria as respostas de retorno da API
        request:
            any
```

```
        response:
            status: String
            any
    """

    result = {}
    if value is not None:
        result.update(value)

    if response.get("message", None) is not None:
        result.update({"message": response["message"]})

    result.update({"status": response["status"]})

    if error is not None:
        result.update({"errors": error})

    if pagination is not None:
        result.update({"pagination": pagination})

    headers.update({"Access-Control-Allow-Origin": "*"})
    headers.update({"server": "PNRD NET"})

    return make_response(jsonify(result), response["http_code"], headers)
```

utils/errors.py

```
import json


class ApiBadRequest(Exception):
    def __init__(self, message):
        self.status_code = 400
        self.message = 'Bad Request: ' + message
        super().__init__(message)


class ApiInternalError(Exception):
    def __init__(self, message):
        self.status_code = 500
        self.message = 'Internal Error: ' + message
        super().__init__(message)


class ApiNotFound(Exception):
    def __init__(self, message):
        self.status_code = 404
        self.message = 'Not Found: ' + message
        super().__init__(message)
```

```
class ApiUnauthorized(Exception):
    def __init__(self, message):
        self.status_code = 401
        self.message = 'Unauthorized: ' + message
        super().__init__(message)
```

Esse programa contém parte de código dos repositórios:

Sawtooth Simple Supply:

- https://github.com/hyperledger/education-sawtooth-simple-supply
- codeowners: @agunde406 @chenette @danintel @dcmiddle @dplumb94 @jsmitchell @peterschwarz @vaporos

Sawtooth Core:

- https://github.com/hyperledger/sawtooth-core

requirements.txt

```
autopep8==1.5.7
bcrypt==3.2.0
black==21.9b0
cbor==1.0.0
certifi==2021.10.8
cffi==1.15.0
charset-normalizer==2.0.7
click==8.0.3
colorlog==6.5.0
Flask==2.0.2
idna==3.3
itsdangerous==2.0.1
Jinja2==3.0.2
MarkupSafe==2.0.1
mypy-extensions==0.4.3
pathspec==0.9.0
platformdirs==2.4.0
protobuf==3.18.1
pycodestyle==2.8.0
pycparser==2.20
pycrypto==2.6.1
python-dotenv==0.19.1
PyYAML==6.0
pyzmq==22.3.0
regex==2021.10.8
requests==2.26.0
```

```
sawtooth-sdk==1.2.3
secp256k1==0.11.1
six==1.16.0
toml==0.10.2
tomli==1.2.1
typing-extensions==3.10.0.2
urllib3==1.26.7
Werkzeug==2.0.2
```