Universidade Federal Fluminense – UFF

Instituto de Computação – IC

Departamento de Ciência da Computação - TCC

Curso de graduação em Ciência da Computação

**Trabalho de Contratos Inteligentes**

Leonardo Pereira

Gabriel Ferreira

Lucas Fauster

Gustavo Lopes

Brendo Costa



**Professor:** Igor Machado dos Santos

**Disciplina:** Tópicos em Redes de Computadores III

**Semestre:** 2021.2

Niterói

2022

**SUMÁRIO**

[**INTRODUÇÃO**](#_gjdgxs) **3**

**MOTIVAÇÃO 4**

**METODOLOGIA 5**

# INTRODUÇÃO

Neste trabalho iremos apresentar nossa proposta de contrato inteligente utilizando conceitos vistos em sala de aula, buscando resolver um problema presente no cotidiano de grandes empresas e baseado em uma aplicação de mercado que busca resolver este problema.

Porém, antes de partirmos para a especificação do problema e proposta de solução através da utilização de contratos inteligentes, se faz necessária a introdução do conceito de blockchain e contratos inteligentes para melhor compreensão do trabalho.

* 1. **Blockchain**

O conceito da Blockchain foi criado por Satoshi Nakamoto em seu artigo “*Bitcoin: A Peer-to-Peer Electronic Cash System”* [1]. O problema que Nakamoto resolveu com a blockchain foi o de estabelecer confiança em um sistema distribuído. Mais especificamente, o problema de criar um armazenamento distribuído de documentos com *timestamps* em que nenhuma parte pudesse alterar o conteúdo dos dados sem detecção.

Dessa forma, a blockchain é um livro-razão compartilhado, distribuído e em crescimento contínuo de blocos, que são selados criptograficamente com uma “impressão digital” gerada por uma função de hash. Cada bloco está ligado à anterior referindo-se ao seu valor de hash. Os computadores, ou nós, que se conectam a blockchain verificam se uma transação é válida de acordo com as regras do contrato inteligente.

A Blockchain possui características interessantes para serviços bancários, financeiros e outros serviços que precisam manter um histórico de registros Isso se dá pois a blockchain é resiliente, operando de forma descentralizada não requerindo um servidor central e não possuindo ponto único de falha e por ser transparente para todos os usuários da rede, uma vez que todos podem ver todas as operações realizadas nela.

* 1. **Contratos Inteligentes**

Definido por Nick Szabo em 1994 [2], um contrato inteligente é um código que é executado na blockchain para facilitar, executar e impor os termos de um acordo. O principal objetivo de um contrato inteligente é executar automaticamente os termos de um acordo uma vez que as condições especificadas sejam atendidas. Como vantagens, os contratos inteligentes prometem baixas taxas de transação em comparação com sistemas tradicionais que exigem um terceiro confiável para fazer cumprir e executar os termos de um acordo.

1. **MOTIVAÇÃO**

Grandes empresas, principalmente do ramo tecnológico, usam um vasto número de máquinas para executar suas rotinas diárias. Essas diversas máquinas podem conter bases de dados com dados sensíveis de clientes, podem ser responsáveis por rodar aplicações críticas para o funcionamento da empresa ou podem ser máquinas utilizadas para desenvolvimento, com menor criticidade.

Em ambientes como esse, existe o problema de controlar o acesso a essas máquinas, mantendo-as protegidas por aprovações de diferentes níveis e por um determinado período de tempo. Dessa forma, uma pessoa só conseguiria acessar uma máquina crítica por um determinado período se fosse aprovada por um grupo seleto de aprovadores.

Algumas soluções de mercado buscam resolver esse problema, como o caso da solução da empresa [Senha Segura](https://senhasegura.com/pt-br/), que foi utilizada de inspiração para esse trabalho. Esta empresa possui uma plataforma responsável por segregar o acesso às diversas máquinas com base em seu nível de criticidade. As máquinas são divididas em diferentes níveis, como *low, medium* e *high* e seus critérios para aprovação são definidos pelo cliente. Para acessar as máquinas, é preciso emitir uma solicitação que é enviada para os devidos aprovadores. Essa solicitação é formada por uma hora de início e uma hora de fim para o acesso e uma justificativa para o acesso. Após emitida a solicitação, um email é enviado para os aprovadores que podem aceitar ou rejeitar a solicitação. Uma vez aprovada a solicitação, é enviado um email para o solicitante informando que seu acesso foi concedido. Dessa forma, torna-se possível acessar a máquina através do software do Senha Segura, que grava a sessão do usuário para fins de auditoria.

Este trabalho busca elaborar uma solução para o problema de controle de acessos via aprovações com inspiração na solução da empresa Senha Segura. Porém, diferencia-se ao utilizar a abordagem de blockchain e contratos inteligentes para gerir as solicitações e aprovações.

**3. METODOLOGIA**

**3.1 RESUMO**

Para este trabalho, realizamos um contrato inteligente utilizando do ecossistema da blockchain do Ethereum. Essa escolha foi devido ao fato da blockchain do Ethereum permitir a criação de carteiras multisig, bem como pela experiência prévia de membros do grupo trabalhando de forma profissional com Ethereum. Implementamos o contrato inteligente na linguagem Solidity, uma linguagem de programação orientada a objetos e de tipagem estática designada para o desenvolvimento de contratos inteligentes. Solidity é utilizada por várias plataformas de blockchain, sendo que seu uso mais notável ocorre dentro do Ethereum. Para compilar e testar nosso contrato, selecionamos o projeto Remix, que é composto de uma IDE distribuída como uma aplicação web juntamente com uma série de bibliotecas que a compõem. No Remix, podemos escrever, compilar, debugar e testar nosso contrato através de uma interface interativa.

**3.2 O CONTRATO**

Nosso contrato foi desenvolvido de forma simples e sua legibilidade é clara e objetiva. A primeira parte do código é a definição de dois grupos de constantes inteiras, ou *enums*. Cada um dos *enums* é composto por três constantes:

O primeiro grupo, AccountRole, define as constantes que representam os possíveis cargos de um usuário do sistema hipotético. Esses cargos são:

* Estagiário
* Regular
* Gerente

Já o segundo grupo, permissionLevel, define os possíveis níveis de permissão que um usuário pode obter dentro do sistema. São eles:

* Baixo
* Médio
* Alto

A próxima parte do código define o funcionamento do nosso contrato em si, disposto dentro de um escopo de acordo com a linguagem Solidity. Em seguida, definimos uma constante pública do tipo “uint256” (inteiro não-assinado de 256 bits) denominada “defaultDeltaDateTimeExpiration” e a definimos como 30 dias. Essa constante representa o tempo de validade de uma conta referente ao contrato. Em seguida, definimos uma variável pública do tipo “address” denominada “owner”, que guardará o endereço da conta multisig a ser utilizada. Em seguida, definimos também um evento, denominado “solicitarPermissao”, que pega como argumentos o endereço do solicitando, o endereço da conta de destino e o nível de permissão requisitado.

O próximo passo foi declarar um struct "Funcionario", composto pelas variáveis “state”, que representa um valor dentro os possíveis do *enum* “permissionLevel” previamente definido, bem como “role”, que representa um valor dentro dos possíveis do *enum* “AccountRole”. Além disso, para armazenar o período de validade de toda entrada, definimos uma variável do tipo “uint256” denominada “dateTimeExpiration”.

Em seguida, utilizamos o recurso de mapeamento da sintaxe da linguagem Solidity, o *mapping*. Com esse recurso, mapeamos entradas de endereços (*addresses*) para um struct “Funcionario” correspondente.

Além disso, definimos um conjunto de funções para o funcionamento do contrato:

* Função add(): toma como argumentos o endereço de uma conta e uma função, mapeando esses argumentos para uma instância do struct “Funcionario” associada ao endereço da conta.
* Funções isMedio() e isAlto(): retornam um valor booleano que indica o respectivo nível de permissão da conta associada ao endereço passado como argumento. Importante notar que o valor booleano retornado não está atrelado somente ao tipo do nível de permissão da conta, mas também ao período de validade da permissão. Dito isso, a função isMedio(), por exemplo, só retornará “true” no caso do nível de permissão da conta ser “Medio” e a validade da permissão não tiver expirado levando em conta o *timestamp* do bloco atual.
* Funções setBaixo(), setMedio() e setAlto(): modificam o nível de permissão de uma conta associada a um endereço passada como argumento. setBaixo() e setMedio() podem ser chamadas por um solicitante com o cargo “Gerente”, enquanto setAlto() só pode ser executada pelo endereço da carteira multisig. Quando chamadas, as três funções também definem a data de expiração da permissão como o *timestamp* do bloco atual somados ao valor de “defaultDeltaDateTimeExpiration”, previamente definido como 30 dias.
* solicitarPermissaoAlta(): essa função só pode ser executada por um solicitante com o cargo “Gerente”, e emite um evento “solicitarPermissao” com o endereço passado como argumento da função.

# REFERÊNCIAS

NAKAMOTO, Satoshi. Bitcoin: A peer-to-peer electronic cash system. **Decentralized Business Review**, p. 21260, 2008.

SZABO, Nick. Formalizing and securing relationships on public networks. **First Monday**, 2(9), 1997.